# gb_master's /dev/null

## … and I said, "Hello, Satan. I believe it's time to go."

WRITTEN BY GB_MASTERAUGUST 24, 2014AUGUST 3, 2015

# x86 Exploitation 101: this is the first witchy house

So, history goes on with Phantasmal Phantasmagoria publishing a groundbreaking article in 2005 (right after the one-line-of-code *unlink* fix) called Malloc Maleficarum (http://packetstorm.foofus.com/papers/attack/MallocMaleficarum.txt) proposing five new ways of attacking the Linux heap implementation. If it took four years to fix the *unlink* vulnerability with just one line of code, so things looked pretty interesting in 2005. The article proposed by Phantasmal Phatasmagoria actually didn't include any example of exploit and was more or less purely theoretical, pointing the finger to new directions. We had to wait two years for the first incarnation of one of these techniques and two more years to see an article describing all the others (I'm not saying that in the meanwhile there were no exploits based on these new vulnerabilities). Anyway the hacker K-sPecial published his own article on .aware called The House of Mind (http://www.awarenetwork.org/etc/alpha/?x=4) and a most-probably spanish hacker called *blackngel* (yes, without the "a") published another article on Phrack 66 on 2009 called Malloc Des-Maleficarum (http://phrack.org/issues/66/10.html#article), both explaining, with examples, how to use the techniques previously mentioned.

Phantasmal Phantasmagorias named these techniques with names of Houses (who knows why), so we have:

- The House of Prime
- The House of Mind
- The House of Force
- The House of Lore
- The House of Spirit

I will try to explain how do they work (some of them, again, not working anymore) one by one. In order to study these techniques it's strongly advisable to give a look at the glibc's source code: the version available at the time when the articles were written was 2.3.5 (https://sourceware.org/git/?p=glibc.git;a=blob;f=malloc/malloc.c;h=e3ccbde7b5b84affbf6ff2387a5151310235f0a3;hb=1afdd17390f6febdfe559e16dfc5c5718f8934aa). Anyway, before doing this, it's mandatory to understand the concept of *fastbin* (somebody remembers the *fastbinsY[NFASTBIN]* array in the *malloc_state* structure?). Fastbins are different from the standard bins:

1. Chunks included in the fastbin are small (by default, their size can be up to 64+4+4 bytes, but the maximum can be tuned up to 80+4+4 bytes by setting the DEFAULT_MXFAST variable)
2. Chunks handled by fastbins keep are not coalesced with the neighbors when it's free() time, as they keep their *inuse* bit set
3. The list data structure has only a singular link

They are removed in a LIFO fashion, while the classic bin is organized in FIFO

Handling these chunks in fastbins allows a faster access, even if the price to pay is a higher fragmentation due to the missing coalescence.

# THE HOUSE OF PRIME

Ingredients:

- Two free()'s of chunks under the exploiter's control (the exploiter MUST be able to modify the size of these two chunks)
- Those two free()'s must be followed by a call to malloc() in which the exploiter can write data

Hypothetical scenario:

```
char *overflowed_ptr = (char *)malloc(256);
char *ptr1 = (char *)malloc(256);
char *ptr2 = (char *)malloc(256);
char *ptr3;
[...]
/* overflow on overflowed_ptr */
[...]
free(ptr1);
[...]
free(ptr2);
[...]
ptr3 = (char *)malloc(256);
```

The first goal of this technique is to overwrite the maximum chunk size allowed to be handled by fastbins. This information is stored at initialization time in the *max_fast* of the *malloc_state* structure: this step is done at line #2295 thanks to the *set_max_fast* macro.

The whole trip starts with *free(ptr1)*, so it's cool to give a look at the beginning of this function:

```
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);

  /* Little security check which won't hurt performance: the
     allocator never wrapps around at the end of the address space.
     Therefore we can exclude some size values which might appear
     here by accident or by "design" from some intruder.  */
  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;
    }
```

It is, of course, of crucial importance to be able to pass the "little security check": *-size* must be greater than the value of *p* and that *p* is well aligned. Even if the exploiter doesn't actually control the value of *p*, he (or she) can control its size: the smallest size possible is our aim. So, if the three LSB of the size are reserved for flags, this means that the smallest chunk is 8 byte large.

Going on, at line #4244 we find this other piece of code:

```
  /*
    If eligible, place chunk on a fastbin so it can be found
    and used quickly in malloc.
  */

  if ((unsigned long)(size) <= (unsigned long)(av->max_fast)

#if TRIM_FASTBINS
      /*
        If TRIM_FASTBINS set, don't place chunks
        bordering top into fastbins
      */
      && (chunk_at_offset(p, size) != av->top)
#endif
      ) {

    if (__builtin_expect (chunk_at_offset (p, size)->size <= 2 * SIZE_
        || __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)]);
    /* Another simple check: make sure the top of the bin is not the
       record we are going to add (i.e., double free).  */
    if (__builtin_expect (*fb == p, 0))
      {
        errstr = "double free or corruption (fasttop)";
        goto errout;
      }
    p->fd = *fb;
    *fb = p;
  }
```

Ah, here's the check for the size of a chunk: if it's smaller than the maximum size for a fastbin, then the chunk goes into a fastbin. The first *if* is easily passed, as we used the smallest size possible, and anyway *av->max_fast* is equal to 72 by default (*av* is the arena pointer). Then, woooops, another check on the size: this time is on the **next** chunk. The next chunk's size must be greater than 2*SIZE_SZ (2*4=8): again this is not a big issue, as the exploiter can control the size of the second chunk as well (it's easier if the overflow allows NULL characters, otherwise it gets tricky). Also the next chunk's size must be less than *av->system_mem* (of course).

Anyway, this size won't be even set in the real *size* field of *ptr2*, as the next chunk's address is computed by adding *ptr1*'s size to *ptr1* address (as usual): fact is that the exploiter changed the size to 0, so, according to *free(ptr1)*, *ptr2*'s chunk is located at *ptr1+0* (after *ptr1*'s headers), while instead it's

located at *ptr1+256*. The layout of the exploit starts taking a shape:

```
0x41 0x41 0x41 0x41            ptr1 prev_size
0x09 0x00 0x00 0x00            ptr1 size + PREV_INUSE bit
0x41 0x41 0x41 0x41            ptr2 (according to free(ptr1)) prev_siz
0x10 0x00 0x00 0x00            ptr2 (according to free(ptr1)) size
0x41 0x41 0x41 0x41            -\
[...]                           | still part of ptr1's space (248 * 0x
0x41 0x41 0x41 0x41            -/
0x41 0x41 0x41 0x41            ptr2 prev_size
```

Important thing to keep in mind, before going on to the next step, is how the fields of the *malloc_state* are organized:

```
/* The maximum chunk size to be eligible for fastbin */
INTERNAL_SIZE_T  max_fast;   /* low 2 bits used as flags */

/* Fastbins */
mfastbinptr      fastbins[NFASTBINS];

/* Base of the topmost chunk -- not otherwise kept in a bin */
mchunkptr        top;
```

The position of *max_fast* compared to *fastbins* (huh, it was called *fastbins* back in the days and not *fastbinsY*) is crucial: they are **contiguous**.

So, the line #4264 reads the address of the fastbin from the address, given the index. The index is computed by using the following macro:

```
/* offset 2 to use otherwise unindexable first 2 bins */
#define fastbin_index(sz)        ((((unsigned int)(sz)) >> 3) - 2)
```

What happens when 8 is the size of the chunk? Well, the result is -1. Given the layout of *malloc_state*, the returned address is the one of the *max_fast* field and it is stored in the *fb* variable. The rest is done by line #4273: the *max_fast* variable is set with the value of *p*. This means that now the maximum size is set to a value of the order of 0x080XXXXX.

The next goal is to overwrite the *arena_key* variable during *free(ptr2)*: this is a very particular one, as it's thread dependent and it's used to identify the arena for the current thread. In case of multi-threaded applications, this can become a little bit painful; in case of single-threaded applications, instead, this one can be treated like a standard variable. The *get_arena* macro uses the value stored in the *arena_key* variable to retrieve the arena and set the mutex on it: in case it fails, it creates a new arena:

```
/* arena_get() acquires an arena and locks the corresponding mutex.
   First, try the one last locked successfully by this thread.  (This
   is the common case and handled with a macro for speed.)  Then, loop
   once over the circularly linked list of arenas.  If no arena is
   readily available, create a new one.  In this latter case, `size'
   is just a hint as to how much memory will be required immediately
   in the new arena. */

#define arena_get(ptr, size) do { \
  Void_t *vptr = NULL; \
  ptr = (mstate)tsd_getspecific(arena_key, vptr); \
  if(ptr && !mutex_trylock(&ptr->mutex)) { \
    THREAD_STAT(++(ptr->stat_lock_direct)); \
  } else \
    ptr = arena_get2(ptr, (size)); \
} while(0)
```

The *arena_key* variable is actually stored in the *arena.c* file and not in *malloc.c*: this means that its location in memory might be higher or lower than the actual arena. The exploitation is possible only when *arena_key* is at a higher address. Overwriting this value is possible by using the fastbins again, so another call to the *free()* function is required.

In a standard 32-bit sytems, *NFASTBINS* gets set to 10: this value is computed through different macros presuming the maximum chunk size possible (*MAX_FAST_SIZE* = 80, as we said before). This means that, even if a smaller size is specified, the amount of allocated fastbins is always the same. The fastbin index is computed, then, dinamically by using the now-familiar *fastbin_index* macro. As the size for a fastbin is always smaller than the *MAX_FAST_SIZE* macro and as the number of allocated fastbins is computed on the maximum chunk size, this way of computing the index is safe. Problem is that *av->max_fast* variable now changed and the *size* variable can be DEFINITELY bigger than *MAX_FAST_SIZE*. The aim is to overwrite *arena_key* by correctly setting an index for the *fastbins* array: the index must make *fastbins[index]* pointing to *arena_key*.

In the example provided by Phantasmal Phantasmagoria, *arena_key* is 1156 bytes away from *fastbins[0]*, so I can use still this value as a valid example: this means that *fastbin_index* must return 1156/sizeof(mfastbinptr)=289. Inverting the *fastbin_index* macro is possible:

*(289 + 2) << 3 = 2328 = 0x918*

The PREV_INUSE bit needs to be set for *ptr2*, so its size must be set to 0x919. In the end, *fb* will point exactly at *arena_key*, that will be overwritten with the value of *ptr2*. The exploit layout evolves into:

```
0x41 0x41 0x41 0x41            ptr1 prev_size
0x09 0x00 0x00 0x00            ptr1 size + PREV_INUSE bit
0x41 0x41 0x41 0x41            ptr2 (according to free(ptr1)) prev_siz
0x10 0x00 0x00 0x00            ptr2 (according to free(ptr1)) size
0x41 0x41 0x41 0x41            -\
[...]                           | still part of ptr1's space (248 * 0x
0x41 0x41 0x41 0x41            -/
0x41 0x41 0x41 0x41            ptr2 prev_size
0x19 0x09 0x00 0x00            ptr2 size + PREV_INUSE bit
```

So, the second task is complete. The reason why it is important to overwrite *arena_key* gets evident when the *malloc* is called. A little analysis of first line of code of a *malloc* is mandatory here:

```
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;
}
```

First thing the malloc tries to do is to retrieve the arena and to store the address in *ar_ptr* by using the *arena_get* macro. As the *arena_key* is set to a value different from zero, the macro won't try to create a new arena. As *ar_ptr* is the current arena and points directly to the *ptr2* chunk space, the *max_fast* field corresponds to the *ptr2*'s size field. The pointer is then passed to the *_int_malloc* function, in which the following happens:

```
Void_t*
_int_malloc(mstate av, size_t bytes)
{
  INTERNAL_SIZE_T nb;               /* normalized request size */
  unsigned int    idx;              /* associated bin index */
  mbinptr         bin;              /* associated bin */
  mfastbinptr*    fb;               /* associated fastbin */

  mchunkptr       victim;           /* inspected/selected chunk */

  [...]

  /*
    Convert request size to internal form by adding SIZE_SZ bytes
    overhead plus possibly more to obtain necessary alignment and/or
    to obtain a size of at least MINSIZE, the smallest allocatable
    size. Also, checked_request2size traps (returning 0) request sizes
    that are so large that they wrap around zero when padded and
    aligned.
  */

  checked_request2size(bytes, nb);

  /*
    If the size qualifies as a fastbin, first check corresponding bin.
    This code is safe to execute even if av is not yet initialized, so
    can try it without checking, which saves some time on this fast pa
  */

  if ((unsigned long)(nb) <= (unsigned long)(av->max_fast)) {
    long int idx = fastbin_index(nb);
    fb = &(av->fastbins[idx]);
    if ( (victim = *fb) != 0) {
      if (__builtin_expect (fastbin_index (chunksize (victim)) != idx,
        malloc_printerr (check_action, "malloc(): memory corruption (f
                         chunk2mem (victim));
      *fb = victim->fd;
      check_remalloced_chunk(av, victim, nb);
      return chunk2mem(victim);
    }
  }
```

1. If the requested size is smaller than *av->max_fast,* then the fastbin index is computed on the request and the fastbin is retrieved: by setting a fake fastbin entry, it is possible to return a stack address. The problem comes with the following security check on the chunk retrieved: the fastbin index is recomputed and compared to the one previously computed. This means that the size of the chunk must be correctly set. So the exploiter can't just return an address in the stack, but **must use an address 4 bytes before an user controlled value in the stack set to the chunk size**.

Once found this address of memory, the exploit should allow to return always the same address of memory, whichever request is made:

```
0x41 0x41 0x41 0x41          ptr1 prev_size
0x09 0x00 0x00 0x00          ptr1 size + PREV_INUSE bit
0x41 0x41 0x41 0x41          ptr2 (according to free(ptr1)) prev_siz
0x10 0x00 0x00 0x00          ptr2 (according to free(ptr1)) size
0x41 0x41 0x41 0x41          -\
[...]                         | still part of ptr1's space (248 * 0x
0x41 0x41 0x41 0x41          -/
0x41 0x41 0x41 0x41          ptr2 prev_size
0x19 0x09 0x00 0x00          ptr2 size + PREV_INUSE bit
0x... CHUNK ADDRESS          the address location to overwrite
0x... CHUNK ADDRESS          set for all the possible fastbins
0x... CHUNK ADDRESS
0x... CHUNK ADDRESS
[...]
0x... CHUNK ADDRESS
0x10 0x00 0x00 0x00          after the address, only free chunks
0x10 0x00 0x00 0x00
0x10 0x00 0x00 0x00
[...]
0x10 0x00 0x00 0x00
```

Let's say that the address returned is the location of memory where the return address is stored, then the exploiter will be able to overwrite it with whatever he wants.

2. If it has been possible to allocate a chunk bigger than the second chunk, then the first *if* is obviously skipped and the following code is executed at #3925:

```
/*
  Process recently freed or remaindered chunks, taking one only if
  it is exact fit, or, if this a small request, the chunk is remainder
  the most recent non-exact fit.  Place other traversed chunks in
  bins.  Note that this step is the only place in any routine where
  chunks are placed in bins.

  The outer loop here is needed because we might not realize until
  near the end of malloc that we should have consolidated, so must
  do so and retry. This happens at most once, and only when we would
  otherwise need to expand memory to service a "small" request.
*/

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 a small request, try to use last remainder if it is the
       only chunk in unsorted bin.  This helps promote locality for
       runs of consecutive small requests. This is the only
       exception to best-fit, and applies only when there is
       no exact fit for a small chunk.
    */

    if (in_smallbin_range(nb) &&
        bck == unsorted_chunks(av) &&
        victim == av->last_remainder &&
        (unsigned long)(size) > (unsigned long)(nb + MINSIZE)) {
        [...]
      }

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

      /* Take now instead of binning if exact fit */

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

```
}
[...]
```

The *unsorted_chunks* at line #3927 will return the address of *av->bins[0]*: this address + 12 is going to be stored in the *victim* variable. *bck* will point to *victim->bk* (which means victim + 12).
If *&av->bins[0] + 16 – 12* is stored at *&av->bins[0] + 12*, then

```
victim = &av->bins[0] + 4
```

If the return address location – 8 is stored at *&av->bins[0] + 16*, then

```
bck = (&av->bins[0] + 4)->bk = av->bins[0] + 16 = &EIP - 8
```

A JMP instruction must be set at *av->bins[0]* in order to jump at *&av->bins[0] + 20*. When the execution reaches line #3965 (**unsorted_chunks(av)->bk = bck**), in the end the following will happen:

```
bck->fd = EIP = &av->bins[0]
```

A NOP slide + shellcode is then required to be stored at *&av->bins[0] + 20*. When the RET instruction will be executed, then the flow will be redirected to the JMP instruction aforementioned and the heap overflow will be fully exploited.

So far, this is the first one of the techniques described by Phantasmal Phantasmagoria. It actually didn't last that much after the publication of the article, as, two days after, this (https://sourceware.org/git/?p=glibc.git;a=commitdiff;h=bf58906631af8fe0d57625988b1d003cc09ef01d;hp=04ec80e410b4efb0576a2ffd0d2f29ed1fdac451) patch made the *free* function fail in case the size of the chunk is smaller than *MINSIZE* (set to 16). As the base of this kind of exploit is the ability to free chunks that are 8 bytes long, then this whole thing is not working anymore since glibc 2.4.
Nothing new has been introduced in this article, as it's just a reporting of what others already did. Anyway, it becomes evident how difficult heap overflows can be. And it's not over yet, as the House of Mind is coming into town…

POSTED IN EXPLOITATION.

# 4 thoughts on "x86 Exploitation 101: this is the first witchy house"

1. 

   **JACK MA** SAYS:
   DECEMBER 8, 2015 AT 8:45 PM

   if (__builtin_expect ((uintptr_t) p > (uintptr_t) -size, 0)
   || __builtin_expect ((uintptr_t) p & MALLOC_ALIGN_MASK, 0))

   why do we compare p with -size? -size is a negative number. a pointer is always positive.

   REPLY

   1. 

      **GB_MASTER** SAYS:
      DECEMBER 8, 2015 AT 9:19 PM

      The line "the allocator never wrapps around at the end of the address space" makes me guess that the check you mentioned plays on the trick of flipping the bits of a variable by negating it. For example, on a 32-bit system, if the size of the chunk is 88, this means that the pointer can't really be greater than 0xFFFFFFA8 (which is 0xFFFFFFFF – 0x58 + 1) and you can easily get that value by negating 88 (0x58 -> 0xFFFFFFA8).

      I hope I've guessed it right and that I've been clear explaining it.

      REPLY
2. Pingback: Linux堆溢出之fastbin-knowsec-安全新闻资讯
3. Pingback: 힙 오버플로우를 통한 fastbin 컨트롤