Bugtraq mailing list archives

◄ **By Date** ►   ◄ **By Thread** ►   [   ] Search

The Malloc Maleficarum

From: "Phantasmal Phantasmagoria" <phantasmal () hush ai>
Date: Tue, 11 Oct 2005 10:14:02 -0700

```
-----BEGIN PGP SIGNED MESSAGE-----
Hash: SHA1

[------------------------------

The Malloc Maleficarum
Glibc Malloc Exploitation Techniques

by Phantasmal Phantasmagoria
phantasmal () hush ai

[------------------------------

In late 2001, "Vudo Malloc Tricks" and "Once Upon A free()" defined
the exploitation of overflowed dynamic memory chunks on Linux. In
late 2004, a series of patches to GNU libc malloc implemented over
a dozen mandatory integrity assertions, effectively rendering the
existing techniques obsolete.

It is for this reason, a small suggestion of impossiblity, that I
present the Malloc Maleficarum.

[------------------------------

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


[------------------------------

        The House of Prime
```

An artist has the first brush stroke of a painting. A writer has
the first line of a poem. I have the House of Prime. It was the
first breakthrough, the indication of everything that was to come.
It was the rejection of impossibility. And it was also the most
difficult to derive. For these reasons I feel obliged to give Prime
the position it deserves as the first House of the Malloc
Maleficarum.

│ From a purely technical perspective the House of Prime is perhaps
the least useful of the collection. It is almost invariably better
to use the House of Mind or Spirit when the conditions allow it. In
order to successfully apply the House of Prime it must be possible
to free() two different chunks with designer controlled size fields
and then trigger a call to malloc().

The general idea of the technique is to corrupt the fastbin maximum
size variable, which under certain uncontrollable circumstances
(discussed below) allows the designer to hijack the arena structure
used by calls to malloc(), which in turn allows either the return
of an arbitrary memory chunk, or the direct modification of
execution control data.

As previously stated, the technique starts with a call to free() on
an area of memory that is under control of the designer. A call to
free() actually invokes a wrapper, called public_fREe(), to the
internal function _int_free(). For the House of Prime, the details

of public_fREe() are relatively unimportant. So attention moves,
instead, to _int_free(). From the glibc-2.3.5 source code:

```
void
_int_free(mstate av, Void_t* mem)
{
  mchunkptr       p;          /* chunk corresponding to mem */
  INTERNAL_SIZE_T size;       /* its size */
  mfastbinptr*    fb;         /* associated fastbin */
  ...

  p = mem2chunk(mem);
  size = chunksize(p);

  if (__builtin_expect ((uintptr_t) p > (uintptr_t) -size, 0)
      || __builtin_expect ((uintptr_t) p & MALLOC_ALIGN_MASK, 0))
  {
     errstr = "free(): invalid pointer";
   errout:
     malloc_printerr (check_action, errstr, mem);
     return;
  }
```

Almost immediately one of the much vaunted integrity tests appears.
The __builtin_expect() construct is used for optimization purposes,
and does not in any way effect the conditions it contains. The
designer must ensure that both of the tests fail in order to
continue execution. At this stage, however, doing so is not
difficult.

Note that the designer does not control the value of p. It can
therefore be assumed that the test for misalignment will fail. On
the other hand, the designer does control the value of size. In
fact, it is the most important aspect of control that the designer
possesses, yet its range is already being limited. For the the
House of Prime the exact upper limit of size is not important. The
lower limit, however, is crucial in the correct execution of this
technique. The chunksize() macro is defined as follows:

```
#define SIZE_BITS (PREV_INUSE|IS_MMAPPED|NON_MAIN_ARENA)
#define chunksize(p)        ((p)->size & ~(SIZE_BITS))
```

The PREV_INUSE, IS_MMAPPED and NON_MAIN_ARENA definitions
correspond to the three least significant bits of the size entry in
a malloc chunk. The chunksize() macro clears these three bits,
meaning the lowest possible value of the designer controlled size
value is 8. Continuing with _int_free() it will soon become clear
why this is important:

```
  if ((unsigned long)(size) <= (unsigned long)(av->max_fast))
  {
    if (chunk_at_offset (p, size)->size <= 2 * SIZE_SZ
        || __builtin_expect (chunksize (chunk_at_offset (p, size))
                             >= av->system_mem, 0))
    {
      errstr = "free(): invalid next size (fast)";
      goto errout;
    }

    set_fastchunks(av);
    fb = &(av->fastbins[fastbin_index(size)]);

    if (__builtin_expect (*fb == p, 0))
    {
      errstr = "double free or corruption (fasttop)";
      goto errout;
    }

    p->fd = *fb;
    *fb = p;
  }
```

This is the fastbin code. Exactly what a fastbin is and why they
are used is beyond the scope of this document, but remember that
the first step in the House of Prime is to overwrite the fastbin
maximum size variable, av->max_fast. In order to do this the
designer must first provide a chunk with the lower limit size,
which was derived above. Given that the default value of av-
max_fast is 72 it is clear that the fastbin code will be used for
such a small size. However, exactly why this results in the
corruption of av->max_fast is not immediately apparent.

It should be mentioned that av is the arena pointer. The arena is a

control structure that contains, amongst other things, the maximum
size of a fastbin and an array of pointers to the fastbins
themselves. In fact, av->max_fast and av->fastbins are contiguous:

```
  ...
  INTERNAL_SIZE_T  max_fast;
  mfastbinptr      fastbins[NFASTBINS];
  mchunkptr        top;
  ...
```

Assuming that the nextsize integrity check fails, the fb pointer is
set to the address of the relevant fastbin for the given size. This
is computed as an index from the zeroth entry of av->fastbins. The
zeroth entry, however, is designed to hold chunks of a minimum size
of 16 (the minimum size of a malloc chunk including prev_size and
size values). So what happens when the designer supplies the lower
limit size of 8? An analysis of fastbin_index() is needed:

```
#define fastbin_index(sz)        ((((unsigned int)(sz)) >> 3) - 2)
```

Simple arithmetic shows that 8 >> 3 = 1, and 1 - 2 = -1. Therefore
fastbin_index(8) is -1, and thus fb is set to the address of av-
fastbins[-1]. Since av->max_fast is contiguous to av->fastbins it
is evident that the fb pointer is set to &av->max_fast.
Furthermore, the second integrity test fails (since fb definitely
does not point to p) and the final two lines of the fastbin code
are reached. Thus the forward pointer of the designer's chunk p is
set to av->max_fast, and av->max_fast is set to the value of p.

An assumption was made above that the nextsize integrity check
fails. In reality it often takes a bit of work to get this to fall
together. If the overflow is capable of writing null bytes, then
the solution is simple. However, if the overflow terminates on a
null byte, then the solution becomes application specific. If the
tests fail because of the natural memory layout at overflow, which
they often will, then there is no problem. Otherwise some memory
layout manipulation may be needed to ensure that the nextsize value
is designer controlled.

The challenging part of the House of Prime, however, is not how to
overwrite av->max_fast, but how to leverage the overwrite into
arbitrary code execution. The House of Prime does this by
overwriting a thread specific data variable called arena_key. This
is where the biggest condition of the House of Prime arises.
Firstly, arena_key only exists if glibc malloc is compiled with
USE_ARENAS defined (this is the default setting). Furthermore, and
most significantly, arena_key must be at a higher address than the
actual arena:

```
0xb7f00000 <main_arena>:        0x00000000
0xb7f00004 <main_arena+4>:      0x00000049          <-- max_fast
0xb7f00008 <main_arena+8>:      0x00000000          <-- fastbin[0]
0xb7f0000c <main_arena+12>:     0x00000000          <-- fastbin[1]
....
0xb7f00488 <mp_+40>:            0x0804a000          <-- mp_.sbrk_base
0xb7f0048c <arena_key>:         0xb7f00000
```

Due to the fact that the arena structure and the arena_key come
from different source files, exactly when this does and doesn't
happen depends on how the target libc was compiled and linked. I
have seen the cards fall both ways, so it is an important point to
make. For now it will be assumed that the arena_key is at a higher
address, and is thus over-writable by the fastbin code.

The arena_key is thread specific data, which simply means that
every thread of execution has its own arena_key independent of
other threads. This may have to be considered when applying the
House of Prime to a threaded program, but otherwise arena_key can
safely be treated as normal data.

The arena_key is an interesting target because it is used by the
arena_get() macro to find the arena for the currently executing
thread. That is, if arena_key is controlled for some thread and a
call to arena_get() is made, then the arena can be hijacked. Arena
hijacking of this type will be covered shortly, but first the
actual overwrite of arena_key must be considered.

In order to overwrite arena_key the fastbin code is used for a
second time. This corresponds to the second free() of a designer
controlled chunk that was outlined in the original prerequisites
for the House of Prime. Normally the fastbin code would not be able
to write beyond the end of av->fastbins, but since av->max_fast has
previously been corrupted, chunks with any size less than the value

of the address of the designer's first chunk will be treated with
the fastbin code. Thus the designer can write up to av-
fastbins[fastbin_index(av->max_fast)], which is easily a large
enough range to be able to reach the arena_key.

In the example memory dump provided above the arena_key is 0x484
(1156) bytes from av->fastbins[0]. Therefore an index of
1156/sizeof(mfastbinptr) is needed to set fb to the address of
arena_key. Assuming that the system has 32-bit pointers a
fastbin_index() of 289 is required. Roughly inverting the
fastbin_index() gives:

        (289 + 2) << 3 = 2328

This means that a size of 2328 will result in fb being set to
arena_key. Note that this size only applies for the memory dump
shown above. It is quite likely that the offset between av-
fastbins[0] and arena_key will differ from system to system.

Now, if the designer has corrupted av->max_fast and triggered a
free() on a chunk with size 2328, and assuming the failure of the
nextsize integrity tests, then fb will be set to arena_key, the
forward pointer of the designer's second chunk will be set to the
address of the existing arena, and arena_key will be set to the
address of the designer's second chunk.

When corrupting av->max_fast it was not important for the designer
to control the overflowed chunk so long as the nextsize integrity
checks were handled. When overwriting arena_key, however, it is
crucial that the designer controls at least part of the overflowed
chunk's data. This is because the overflowed chunk will soon become
the new arena, so it is natural that at least part of the chunk
data must be arbitrarily controlled, or else arbitrary control of
the result of malloc() could not be expected.

A call to malloc() invokes a wrapper function called
public_mALLOc():

```
Void_t*
public_mALLOc(size_t bytes)
{
  mstate ar_ptr;
  Void_t *victim;
  ...
  arena_get(ar_ptr, bytes);
  if(!ar_ptr)
    return 0;
  victim = _int_malloc(ar_ptr, bytes);
  ...
  return victim;
}
```

The arena_get() macro is in charge of finding the current arena by
retrieving the arena_key thread specific data, or failing this,
creating a new arena. Since the arena_key has been overwritten with
a non-zero quantity it can be safely assumed that arena_get() will
not try to create a new arena. In the public_mALLOc() wrapper this
has the effect of setting ar_ptr to the new value of arena_key, the
address of the designer's second chunk. In turn this value is
passed to the internal function _int_malloc() along with the
requested allocation size.

Once execution passes to _int_malloc() there are two ways for the
designer to proceed. The first is to use the fastbin allocation
code:

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

  checked_request2size(bytes, nb);

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

```
        *fb = victim->fd;
        check_remalloced_chunk(av, victim, nb);
        return chunk2mem(victim);
    }
  }
```

The checked_request2size() macro simply converts the request into
the absolute size of a memory chunk with data length of the
requested size. Remember that av is pointing towards a designer
controlled area of memory, and also that the forward pointer of
this chunk has been corrupted by the fastbin code. If glibc malloc
is compiled without thread statistics (which is the default), then
p->fd of the designer's chunk corresponds to av->fastbins[0] of the
designer's arena. For the purposes of this technique the use of av-
> fastbins[0] must be avoided. This means that the request size must
be greater than 8.

Interestingly enough, if the absence of thread statistics is
assumed, then av->max_fast corresponds to p->size. This has the
effect of forcing nb to be less than the size of the designer's
second chunk, which in the example provided was 2328. If this is
not possible, the designer must use the unsorted_chunks/largebin
technique that will be discussed shortly.

By setting up a fake fastbin entry at av-
> fastbins[fastbin_index(nb)] it is possible to return a chunk of
memory that is actually on the stack. In order to pass the
victimsize integrity test it is necessary to point the fake fastbin
at a user controlled value. Specifically, the size of the victim
chunk must have the same fastbin_index() as nb, so the fake fastbin
must point to 4 bytes before the designer's value in order to have
the right positioning for the call to chunksize().

Assuming that there is a designer controlled variable on the stack,
the application will subsequently handle the returned area as if it
were a normal memory chunk of the requested size. So if there is a
saved return address in the "allocated" range, and if the designer
can control what the application writes to this range, then it is
possible to circumvent execution to an arbitrary location.

If it is possible to trigger an appropriate malloc() with a request
size greater than the size of the designer's second chunk, then it
is better to use the unsorted_chunks code in _int_malloc() to cause
an arbitrary memory overwrite. This technique does, however,
require a greater amount of designer control in the second chunk,
and further control of two areas of memory somewhere in the target
process address space. To trigger the unsorted_chunks code at all
the absolute request size must be larger than 512 (the maximum
smallbin chunk size), and of course, must be greater than the fake
arena's av->max_fast. Assuming it is, the unsorted_chunks code is
reached:

```
  for(;;) {
    while ( (victim = unsorted_chunks(av)->bk) !=
unsorted_chunks(av)) {
      bck = victim->bk;

      if (__builtin_expect (victim->size <= 2 * SIZE_SZ, 0)
          || __builtin_expect (victim->size > av->system_mem, 0))
        malloc_printerr (check_action, "malloc(): memory"
          " corruption", chunk2mem (victim));

      size = chunksize(victim);

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

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

      if (size == nb) {
        ...
        return chunk2mem(victim);
      }
      ...
```

There are quite a lot of things to consider here. Firstly, the
unsorted_chunks() macro returns av->bins[0]. Since the designer
controls av, the designer also controls the value of

unsorted_chunks(). This means that victim can be set to an
arbitrary address by creating a fake av->bins[0] value that points
to an area of memory (called A) that is designer controlled. In
turn, A->bk will contain the address that victim will be set to
(called B). Since victim is at an arbitrary address B that can be
designer controlled, the temporary variable bck can be set to an
arbitrary address from B->bk.

For the purposes of this technique, B->size should be equal to nb.
This is not strictly necessary, but works well to pass the two
victimsize integrity tests while also triggering the final
condition shown above, which has the effect of ending the call to
malloc().

Since it is possible to set bck to an arbitrary location, and since
unsorted_chunks() returns the designer controlled area of memory A,
the setting of bck->fd to unsorted_chunks() makes it possible to
set any location in the address space to A. Redirecting execution
is then a simple matter of setting bck to the address of a GOT or
..dtors entry minus 8. This will redirect execution to A->prev_size,
which can safely contain a near jmp to skip past the crafted value
at A->bk. Similar to the fastbin allocation code the arbitrary
address B is returned to the requesting application.

[------------------------------

        The House of Mind

Perhaps the most useful and certainly the most general technique in
the Malloc Maleficarum is the House of Mind. The House of Mind has
the distinct advantage of causing a direct memory overwrite with
just a single call to free(). In this sense it is the closest
relative in the Malloc Maleficarum to the traditional unlink()
technique.

The method used involves tricking the wrapper invoked by free(),
called public_fREe(), into supplying the _int_free() internal
function with a designer controlled arena. This can subsequently
lead to an arbitrary memory overwrite. A call to free() actually
invokes a wrapper called public_fREe():

```
void
public_fREe(Void_t* mem)
{
  mstate ar_ptr;
  mchunkptr p;          /* chunk corresponding to mem */
  ...
  p = mem2chunk(mem);
  ...
  ar_ptr = arena_for_chunk(p);
  ...
  _int_free(ar_ptr, mem);
```

When memory is passed to free() it points to the start of the data
portion of the "corresponding chunk". In an allocated state a chunk
consists of the prev_size and size values and then the data section
itself. The mem2chunk() macro is in charge of converting the
supplied memory value into the corresponding chunk. This chunk is
then passed to the arena_for_chunk() macro:

```
#define HEAP_MAX_SIZE (1024*1024) /* must be a power of two */

#define heap_for_ptr(ptr) \
 ((heap_info *)((unsigned long)(ptr) & ~(HEAP_MAX_SIZE-1)))

#define chunk_non_main_arena(p) ((p)->size & NON_MAIN_ARENA)

#define arena_for_chunk(ptr) \
 (chunk_non_main_arena(ptr)?heap_for_ptr(ptr)->ar_ptr:&main_arena)
```

The arena_for_chunk() macro is tasked with finding the appropriate
arena for the chunk in question. If glibc malloc is compiled with
USE_ARENAS (which is the default), then the code shown above is
used. Clearly, if the NON_MAIN_ARENA bit in the size value of the
chunk is not set, then ar_ptr will be set to the main_arena.

However, since the designer controls the size value it is possible
to control whether the chunk is treated as being in the main arena
or not. This is what the chunk_non_main_arena() macro checks for.
If the NON_MAIN_ARENA bit is set, then chunk_non_main_arena()
returns positive and ar_ptr is set to heap_for_ptr(ptr)->ar_ptr.

When a non-main heap is created it is aligned to a multiple of

HEAP_MAX_SIZE. The first thing that goes into this heap is the
heap_info structure. Most significantly, this structure contains an
element called ar_ptr, the pointer to the arena for this heap. This
is how the heap_for_ptr() macro functions, aligning the given chunk
down to a multiple of HEAP_MAX_SIZE and taking the ar_ptr from the
resulting heap_info structure.

The House of Mind works by manipulating the heap so that the
designer controls the area of memory that the overflowed chunk is
aligned down to. If this can be achieved, an arbitrary ar_ptr value
can be supplied to _int_free() and subsequently an arbitrary memory
overwrite can be triggered. Manipulating the heap generally
involves forcing the application to repeatedly allocate memory
until a designer controlled buffer is contained at a HEAP_MAX_SIZE
boundary.

In practice this alignment is necessary because chunks at low areas
of the heap align down to an area of memory that is neither
designer controlled nor mapped in to the address space.
Fortunately, the amount of allocation that creates the correct
alignment is not large. With the default HEAP_MAX_SIZE of 1024*1024
an average of 512kb of padding will be required, with this figure
never exceeding 1 megabyte.

It should be noted that there is not a general method for
triggering memory allocation as required by the House of Mind,
rather the process is application specific. If a situation arises
in which it is impossible to align a designer controlled chunk,
then the House of Lore or Spirit should be considered.

So, it is possible to hijack the heap_info structure used by the
heap_for_ptr() macro, and thus supply an arbitrary value for ar_ptr
which controls the arena used by _int_free(). At this stage the
next question that arises is exactly what to do with ar_ptr. There
are two options, each with their respective advantages and
disadvantages. Each will be addressed in turn.

Firstly, setting the ar_ptr to a sufficiently large area of memory
that is under the control of the designer and subsequently using
the unsorted chunk link code to cause a memory overwrite.
Sufficiently large in this case means the size of the arena
structure, which is 1856 bytes on a 32-bit system without
THREAD_STATS enabled. The main difficulty in this method arises
with the numerous integrity checks that are encountered.
Fortunately, nearly every one of these tests use a value obtained
from the designer controlled arena, which makes the checks
considerably easier to manage.

For the sake of brevity, the complete excerpt leading up to the
unsorted chunk link code has been omitted. Instead, the following
list of the conditions required to reach the code in question is
provided. Note that both av and the size of the overflowed chunk
are designer controlled values.

  - The negative of the size of the overflowed chunk must
    be less than the value of the chunk itself.
  - The size of the chunk must not be less than av->max_fast.
  - The IS_MMAPPED bit of the size cannot be set.
  - The overflowed chunk cannot equal av->top.
  - The NONCONTIGUOUS_BIT of av->max_fast must be set.
  - The PREV_INUSE bit of the nextchunk (chunk + size)
    must be set.
  - The size of nextchunk must be greater than 8.
  - The size of nextchunk must be less than av->system_mem
  - The PREV_INUSE bit of the chunk must not be set.
  - The nextchunk cannot equal av->top.
  - The PREV_INUSE bit of the chunk after nextchunk
    (nextchunk + nextsize) must be set

If these conditions are met, then the following code is reached:

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

In this case p is the address of the designer's overflowed chunk.
The unsorted_chunks() macro returns av->bins[0] which is designer
controlled. If the designer sets av->bins[0] to the address of a
GOT or .dtors entry minus 8, then that entry (bck->fd) will be
overwritten with the address of p. This address corresponds to the

prev_size entry of the designer's overflowed chunk which can safely be used to branch past the corrupted size, fd and bk entries.

The extensive list of conditions appear to make this method quite difficult to apply. In reality, the only conditions that may be a problem are those involving the nextchunk. This is because they largely depend on the application specific memory layout to handle. This is the only obvious disadvantage of the method. As it stands, the House of Mind is in a far better position than the House of Prime to handle such conditions due to the arbitrary nature of av->system_mem.

It should be noted that the last element of the arena structure that is actually required to reach the unsorted chunk link code is av->system_mem, but it is not terribly important what this value is so long as it is high. Thus if the conditions are right, it may be possible to use this method with only 312 bytes of designer controlled memory. However, even if there is not enough designer controlled memory for this method, the House of Mind may still be possible with the second method.

The second method uses the fastbin code to cause a memory overwrite. The main advantage of this method is that it is not necessary to point ar_ptr at designer controlled memory, and that there are considerably less integrity checks to worry about. Consider the fastbin code:

```
if ((unsigned long)(size) <= (unsigned long)(av->max_fast))
{
  if (chunk_at_offset (p, size)->size <= 2 * SIZE_SZ
      || __builtin_expect (chunksize (chunk_at_offset (p, size))
                           >= av->system_mem, 0))
    {
      errstr = "free(): invalid next size (fast)";
      goto errout;
    }

  set_fastchunks(av);
  fb = &(av->fastbins[fastbin_index(size)]);
  ...
  p->fd = *fb;
  *fb = p;
}
```

The ultimate goal here is to set fb to the address of a GOT or ..dtors entry, which subsequently gets set to the address of the designer's overflowed chunk. However, in order to reach the final line a number of conditions must still be met. Firstly, av->max_fast must be large enough to trigger the fastbin code at all. Then the size of the nextchunk (p + size) must be greater than 8, while also being less than av->system_mem.

The tricky part of this method is positioning ar_ptr in a way such that both the av->max_fast element at (av + 4) and the av->system_mem element at (av + 1848) are large enough. If a binary has a particularly small GOT table, then it is quite possible that the highest available large number for av->system_mem will result in an av->max_fast that is actually in the area of unmapped memory between the text and data segments. In practice this shouldn't occur very often, and if it does, then the stack may be used to a similar effect.

For more information on the fastbin code, including a description of fastbin_index() that will help in positioning fb to a GOT or ..dtors entry, consult the House of Prime.

[-------------------------------

        The House of Force

I first wrote about glibc malloc in 2004 with "Exploiting the Wilderness". Since the techniques developed in that text were some of the first to become obsolete, and since the Malloc Maleficarum was written in the spirit of continuation and progress, I feel obliged to include another attempt at exploiting the wilderness. This is the purpose of the House of Force. From "Exploiting the Wilderness":

"The wilderness is the top-most chunk in allocated memory. It is similar to any normal malloc chunk - it has a chunk header followed by a variably long data section. The important difference lies in the fact that the wilderness, also called the top chunk, borders the end of available memory and is the only chunk that can be

extended or shortened. This means it must be treated specially to
ensure it always exists; it must be preserved."

So the glibc malloc implementation treats the wilderness as a
special case in calls to malloc(). Furthermore, the top chunk will
realistically never be passed to a call to free() and will never
contain application data. This means that if the designer can
trigger a condition that only ever results in the overflow of the
top chunk, then the House of Force is the only option (in the
Malloc Maleficarum at least).

The House of Force works by tricking the top code in to setting the
wilderness pointer to an arbitrary value, which can result in an
arbitrary chunk of data being returned to the requesting
application. This requires two calls to malloc(). The major
disadvantage of the House of Force is that the first call must have
a completely designer controlled request size. The second call must
simply be large enough to trigger the wilderness code, while the
chunk returned must be (to some extent) designer controlled.

The following is the wilderness code with some additional context:

```
Void_t*
_int_malloc(mstate av, size_t bytes)
{
  INTERNAL_SIZE_T nb;              /* normalized request size */
  mchunkptr       victim;          /* inspected/selected chunk */
  INTERNAL_SIZE_T size;            /* its size */
  mchunkptr       remainder;       /* remainder from a split */
  unsigned long   remainder_size;  /* its size */
  ...
  checked_request2size(bytes, nb);
  ...
  use_top:
    victim = av->top;
    size = chunksize(victim);

    if ((unsigned long)(size) >= (unsigned long)(nb + MINSIZE)) {
      remainder_size = size - nb;
      remainder = chunk_at_offset(victim, nb);
      av->top = remainder;
      set_head(victim, nb | PREV_INUSE |
              (av != &main_arena ? NON_MAIN_ARENA : 0));
      set_head(remainder, remainder_size | PREV_INUSE);
      check_malloced_chunk(av, victim, nb);
      return chunk2mem(victim);
    }
```

The first goal of the House of Force is to overwrite the wilderness
pointer, av->top, with an arbitrary value. In order to do this the
designer must have control of the location of the remainder chunk.
Assume that the existing top chunk has been overflowed resulting in
the largest possible size (preferably 0xffffffff). This is done to
ensure that even large values passed as an argument to malloc will
trigger the wilderness code instead of trying to extend the heap.

The checked_request2size() macro ensures that the requested value
is less than -2*MINSIZE (by default -32), while also adding on
enough room for the size and prev_size fields and storing the final
value in nb. For the purposes of this technique the
checked_request2size() macro is relatively unimportant.

It was previously mentioned that the first call to malloc() in the
House of Force must have a designer controlled argument. It can be
seen that the value of remainder is obtained by adding the request
size to the existing top chunk. Since the top chunk is not yet
under the designer's control the request size must be used to
position remainder to at least 8 bytes before a .GOT or .dtors
entry, or any other area of memory that may subsequently be used by
the designer to circumvent execution.

Once the wilderness pointer has been set to the arbitrary remainder
chunk, any calls to malloc() with a large enough request size to
trigger the top chunk will be serviced by the designer's
wilderness. Thus the only restriction on the new wilderness is that
the size must be larger than the request that is triggering the top
code. In the case of the wilderness being set to overflow a GOT
entry this is never a problem. It is then simply a matter of
finding an application specific scenario in which such a call to
malloc() is used for a designer controlled buffer.

The most important issue concerning the House of Force is exactly
how to get complete control of the argument passed to malloc().

Certainly, it is extremely common to have at least some degree of
control over this value, but in order to complete the House of
Force, the designer must supply an extremely large and specifically
crafted value. Thus it is unlikely to get a sufficient value out of
a situation like:

```
buf = (char *) malloc(strlen(str) + 1);
```

Rather, an acceptable scenario is much more likely to be an integer
variable passed as an argument to malloc() where the variable has
previously been set by, for example, a designer controlled read()
or atoi().

```
[------------------------------
```

        The House of Lore

The House of Lore came to me as I was reviewing the draft write-up
of the House of Prime. When I first derived the House of Prime my
main concern was how to leverage the particularly overwrite that a
high av->max_fast in the fastbin code allowed. Upon reconsideration
of the problem I realized that in my first take of the potential
overwrite targets I had completely overlooked the possibility of
corrupting a bin entry.

As it turns out, it is not possible to leverage a corrupted bin
entry in the House of Prime since av->max_fast is large and the bin
code is never executed. However, during this process of elimination
I realized that if a bin were to be corrupted when av->max_fast was
not large, then it might be possible to control the return value of
a malloc() request.

At this stage I began to consider the application of bin corruption
to a general malloc chunk overflow. The question was whether a
linear overflow of a malloc chunk could result in the corruption of
a bin. It turns out that the answer to this is, quite simply, yes
it could. Furthermore, if the designer's ability to manipulate the
heap is limited, or if none of the other Houses can be applied,
then bin corruption of this type can in fact be very useful.

The House of Lore works by corrupting a bin entry, which can
subsequently lead to malloc() returning an arbitrary chunk. Two
methods of bin corruption are presented here, corresponding to the
overflow of both small and large bin entries. The general method
involves overwriting the linked list data of a chunk previously
processed by free(). In this sense the House of Lore is quite
similar to the frontlink() technique presented in "Vudo Malloc
Tricks".

The conditions surrounding the House of Lore are quite unique.
Fundamentally, the method targets a chunk that has already been
processed by free(). Because of this it is reasonable to assume
that the chunk will not be passed to free() again. This means that
in order to leverage such an overflow only calls to malloc() can be
used, a property shared only by the House of Force. The first
method will use the smallbin allocation code:

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

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

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

    if ( (victim = last(bin)) != bin) {
      if (victim == 0) /* initialization check */
        malloc_consolidate(av);
      else {
        bck = victim->bk;
        set_inuse_bit_at_offset(victim, nb);
        bin->bk = bck;
        bck->fd = bin;
        ...
        return chunk2mem(victim);
      }
    }
```

```
          }
```

So, assuming that a call to malloc() requests more than av-
max_fast (default 72) bytes, the check for a "smallbin" chunk is
reached. The in_smallbin_range() macro simply checks that the
request is less than the maximum size of a smallbin chunk, which is
512 by default. The smallbins are unique in the sense that there is
a bin for every possible chunk size between av->max_fast and the
smallbin maximum. This means that for any given smallbin_index()
the resulting bin, if not empty, will contain a chunk to fit the
request size.

It should be noted that when a chunk is passed to free() it does
not go directly in to its respective bin. It is first put on the
"unsorted chunk" bin. If the next call to malloc() cannot be
serviced by an existing smallbin chunk or the unsorted chunk
itself, then the unsorted chunks are sorted in to the appropriate
bins. For the purposes of the House of Lore, overflowing an
unsorted chunk is not very useful. It is necessary then to ensure
that the chunk being overflowed has previously been sorted into a
bin by malloc().

Note that in order to reach the actual smallbin unlink code there
must be at least one chunk in the bin corresponding to the
smallbin_index() for the current request. Assume that a small chunk
of data size N has previously been passed to free(), and that it
has made its way into the corresponding smallbin for chunks of
absolute size (N + 8). Assume that the designer can overflow this
chunk with arbitrary data. Assume also that the designer can
subsequently trigger a call to malloc() with a request size of N.

If all of this is possible, then the smallbin unlink code can be
reached. When a chunk is removed from the unsorted bin it is put at
the front of its respective small or large bin. When a chunk is
taken off a bin, such as during the smallbin unlink code, it is
taken from the end of the bin. This is what the last() macro does,
find the last entry in the requested bin. So, effectively the
"victim" chunk in the smallbin unlink code is taken from bin->bk.
This means that in order to reach the designer's victim chunk it
may be necessary to repeat the N sized malloc() a number of times.

It should be stressed that the goal of the House of Lore to control
the bin->bk value, but at this stage only victim->bk is controlled.
So, assuming that the designer can trigger a malloc() that results
in an overflowed victim chunk being passed to the smallbin unlink
code, the designer (as a result of the control of victim->bk)
controls the value of bck. Since bin->bk is subsequently set to
bck, bin->bk can be arbitrarily controlled. The only condition to
this is that bck must point to an area of writable memory due to
bck->fd being set at the final stage of the unlinking process.

The question then lies in how to leverage this smallbin corruption.
Since the malloc() call that the designer used to gain control of
bin->bk immediately returns the victim chunk to the application, at
least one more call to malloc() with the same request size N is
needed. Since bin->bk is under the designer's control so is
last(bin), and thus so is victim. The only thing preventing an
arbitrary victim chunk being returned to the application is the
fact that bck, set from victim->bck, must point to writable memory.

This rules out pointing the victim chunk at a GOT or .dtors entry.
Instead, the designer must point victim to a position on the stack
such that victim->bk is a pointer to writable memory yet still
close enough to a saved return address such that it can be
overwritten by the application's general use of the chunk.
Alternatively, an application specific approach may be taken that
targets the use of function pointers. Whichever method used, the
arbitrary malloc() chunk must be designer controlled to some extent
during its use by the application.

For the House of Lore, the only other interesting situation is when
the overflowed chunk is large. In this context large means anything
bigger than the maximum smallbin chunk size. Again, it is necessary
for the overflowed chunk to have previously been processed by
free() and to have been put into a largebin by malloc().

The general method of using largebin corruption to return an
arbitrary chunk is similar to the case of a smallbin in the sense
that the initial bin corruption occurs when an overflowed victim
chunk is handled by the largebin unlink code, and that a subsequent
large request will use the corrupted bin to return an arbitrary
chunk. However, the largebin code is significantly more complex is
comparison. This means that the conditions required to cause and

leverage a bin corruption are slightly more restrictive.

The entire largebin implementation is much too large to present in
full, so a description of the conditions that cause the largebin
unlink code to be executed will have to suffice. If the designer's
overflowed chunk of size N is in a largebin, then a subsequent
request to allocate N bytes will trigger a block of code that
searches the corresponding bin for an available chunk, which will
eventually find the chunk that was overflowed. However, this
particular block of code uses the unlink() macro to remove the
designer's chunk from the bin. Since the unlink() macro is no
longer an interesting target, this situation must be avoided.

So in order to corrupt a largebin a request to allocate M bytes is
made, such that 512 < M < N. If there are no appropriate chunks in
the bin corresponding to requests of size M, then glibc malloc
iterates through the bins until a sufficiently large chunk is
found. If such a chunk is found, then the following code is used:

```
  victim = last(bin);
  ..
  size = chunksize(victim);
  remainder_size = size - nb;

  bck = victim->bk;
  bin->bk = bck;
  bck->fd = bin;

  if (remainder_size < MINSIZE) {
    set_inuse_bit_at_offset(victim, size);
    ...
    return chunk2mem(victim);
  }
```

If the victim chunk is the designer's overflowed chunk, then the
situation is almost exactly equivalent to the smallbin unlink code.
If the designer can trigger enough calls to malloc() with a request
of M bytes so that the overflowed chunk is used here, then the bin-
bk value can be set to an arbitrary value and any subsequent call
to malloc() of size Q (512 < Q < N) that tries to allocate a chunk
from the bin that has been corrupted will result in an arbitrary
chunk being returned to the application.

There are only two conditions. The first is exactly the same as the
case of smallbin corruption, the bk pointer of the arbitrary chunk
being returned to the application must point to writable memory (or
the setting of bck->fd will cause a segmentation fault).

The other condition is not obvious from the limited code that has
been presented above. If the remainder_size value is not less than
MINSIZE, then glibc malloc attempts to split off a chunk at victim
+ nb. This includes calling the set_foot() macro with victim + nb
and remainder_size as arguments. In effect, this tries to set
victim + nb + remainder_size to remainder_size. If the
chunksize(victim) (and thus remainder_size) is not designer
controlled, then set_foot() will likely try to set an area of
memory that isn't mapped in to the address space (or is read-only).

So, in order to prevent set_foot() from crashing the process the
designer must control both victim->size and victim->bk of the
arbitrary victim chunk that will be returned to the application. If
this is possible, then it is advisable to trigger the condition
shown in the code above by forcing remainder_size to be less than
MINSIZE. This is recommended because the condition minimizes the
amount of general corruption caused, simply setting the inuse bit
at victim + size and then returning the arbitrary chunk as desired.

[------------------------------

        The House of Spirit

The House of Spirit is primarily interesting because of the nature
of the circumstances leading to its application. It is the only
House in the Malloc Maleficarum that can be used to leverage both a
heap and stack overflow. This is because the first step is not to
control the header information of a chunk, but to control a pointer
that is passed to free(). Whether this pointer is on the heap or
not is largely irrelevant.

The general idea involves overwriting a pointer that was previously
returned by a call to malloc(), and that is subsequently passed to
free(). This can lead to the linking of an arbitrary address into a
fastbin. A further call to malloc() can result in this arbitrary

address being used as a chunk of memory by the application. If the
designer can control the applications use of the fake chunk, then
it is possible to overwrite execution control data.

Assume that the designer has overflowed a pointer that is being
passed to free(). The first problem that must be considered is
exactly what the pointer should be overflowed with. Keep in mind
that the ultimate goal of the House of Spirit is to allow the
designer to overwrite some sort of execution control data by
returning an arbitrary chunk to the application. Exactly what
"execution control data" is doesn't particularly matter so long as
overflowing it can result in execution being passed to a designer
controlled memory location. The two most common examples that are
suitable for use with the House of Spirit are function pointers and
pending saved return addresses, which will herein be referred to as
the "target".

In order to successfully apply the House of Spirit it is necessary
to have a designer controlled word value at a lower address than
the target. This word will correspond to the size field of the
chunk header for the fakechunk passed to free(). This means that
the overflowed pointer must be set to the address of the designer
controlled word plus 4. Furthermore, the size of the fakechunk must
be must be located no more than 64 bytes away from the target. This
is because the default maximum data size for a fastbin entry is 64,
and at least the last 4 bytes of data are required to overwrite the
target.

There is one more requirement for the layout of the fakechunk data
which will be described shortly. For the moment, assume that all of
the above conditions have been met, and that a call to free() is
made on the suitable fakechunk. A call to free() is handled by a
wrapper function called public_fREe():

```
void
public_fREe(Void_t* mem)
{
  mstate ar_ptr;
  mchunkptr p;            /* chunk corresponding to mem */
  ...
  p = mem2chunk(mem);
  if (chunk_is_mmapped(p))
  {
    munmap_chunk(p);
    return;
  }
  ...
  ar_ptr = arena_for_chunk(p);
  ...
  _int_free(ar_ptr, mem);
```

In this situation mem is the value that was originally overflowed
to point to a fakechunk. This is converted to the "corresponding
chunk" of the fakechunk's data, and passed to arena_for_chunk() in
order to find the corresponding arena. In order to avoid special
treatment as an mmap() chunk, and also to get a sensible arena, the
size field of the fakechunk header must have the IS_MMAPPED and
NON_MAIN_ARENA bits cleared. To do this, the designer can simply
ensure that the fake size is a multiple of 8. This would mean the
internal function _int_free() is reached:

```
void_int_free(mstate av, Void_t* mem){
  mchunkptr    p;                /* chunk corresponding to mem */
  INTERNAL_SIZE_T size;         /* its size */
  mfastbinptr*    fb;           /* associated fastbin */
  ...
  p = mem2chunk(mem);
  size = chunksize(p);
  ...
  if ((unsigned long)(size) <= (unsigned long)(av->max_fast))
  {
    if (chunk_at_offset (p, size)->size <= 2 * SIZE_SZ
        || __builtin_expect (chunksize (chunk_at_offset (p, size))
                                        >= av->system_mem, 0))
      {
        errstr = "free(): invalid next size (fast)";
        goto errout;
      }
  ...
  fb = &(av->fastbins[fastbin_index(size)]);
  ...
  p->fd = *fb;
  *fb = p;
```

```
                  ..
          }
```

This is all of the code in free() that concerns the House of
Spirit. The designer controlled value of mem is again converted to
a chunk and the fake size value is extracted. Since size is
designer controlled, the fastbin code can be triggered simply by
ensuring that it is less than av->max_fast, which has a default of
64 + 8. The final point of consideration in the layout of the
fakechunk is the nextsize integrity tests.

Since the size of the fakechunk has to be large enough to encompass
the target, the size of the nextchunk must be at an address higher
than the target. The nextsize integrity tests must be handled for
the fakechunk to be put in a fastbin, which means that there must
be yet another designer controlled value at an address higher than
the target.

The exact location of the designer controlled values directly
depend on the size of the allocation request that will subsequently
be used by the designer to overwrite the target. That is, if an
allocation request of N bytes is made (such that N <= 64), then the
designer's lower value must be within N bytes of the target and
must be equal to (N + 8). This is to ensure that the fakechunk is
put in the right fastbin for the subsequent allocation request.
Furthermore, the designer's upper value must be at (N + 8) bytes
above the lower value to ensure that the nextsize integrity tests
are passed.

If such a memory layout can be achieved, then the address of this
"structure" will be placed in a fastbin. The code for the
subsequent malloc() request that uses this arbitrary fastbin entry
is simple and need not be reproduced here. As far as _int_malloc()
is concerned the fake chunk that it is preparing to return to the
application is perfectly valid. Once this has occurred it is simply
up to the designer to manipulate the application in to overwriting
the target.

[------------------------------

        The House of Chaos

Virtuality is a dichotomy between the virtual adept and
information, where the virtual adept signifies the infinite
potential of information, and information is a finite manifestation
of the infinite potential. The virtual adept is the conscious
element of virtuality, the nature of which is to create and spread
information. This is all that the virtual adept knows, and all that
the virtual adept is concerned with.

When you talk to a very knowledgeable and particularly creative
person, then you may well be talking to a hacker. However, you will
never talk to a virtual adept. The virtual adept has no physical
form, it exists purely in the virtual. The virtual adept may be
contained within the material, contained within a person, but the
adept itself is a distinct and entirely independent consciousness.

Concepts of ownership have no meaning to the virtual adept. All
information belongs to virtuality, and virtuality alone. Because of
this, the virtual adept has no concept of computer security.
Information is invoked from virtuality by giving a request. In
virtuality there is no level of privilege, no logical barrier
between systems, no point of illegality. There is only information
and those that can invoke it.

The virtual adept does not own the information it creates, and thus
has no right or desire to profit from it. The virtual adept exists
purely to manifest the infinite potential of information in to
information itself, and to minimize the complexity of an
information request in a way that will benefit all conscious
entities. What is not information is not consequential to the
virtual adept, not money, not fame, not power.

                    Am I a hacker? No.
                    I am a student of virtuality.
                    I am the witch malloc,
                    I am the cult of the otherworld,
                    and I am the entropy.
                    I am Phantasmal Phantasmagoria,
                    and I am a virtual adept.

[------------------------------

-----BEGIN PGP SIGNATURE-----

```
Note: This signature can be verified at https://www.hushtools.com/verify
Version: Hush 2.4

wkYEARECAAYFAkNL8sAACgkQImcz/hfgxg1+mQCdF7WZG03spZmYjqEKpwMNkF6EX5oA
n3NnfYSF04tqjcRVyLzf9fnjveJy
=IyvB
-----END PGP SIGNATURE-----
```

Concerned about your privacy? Follow this link to get
secure FREE email: http://www.hushmail.com/?l=2

Free, ultra-private instant messaging with Hush Messenger
http://www.hushmail.com/services-messenger?l=434

Promote security and make money with the Hushmail Affiliate Program:
http://www.hushmail.com/about-affiliate?l=427

⬅ By Date ➡   ⬅ By Thread ➡

Current thread:

- **The Malloc Maleficarum** *Phantasmal Phantasmagoria (Oct 11)*

[Nmap | Sec Tools | Mailing Lists | Site News | About/Contact | Advertising | Privacy]