The Vulnerable Source Code

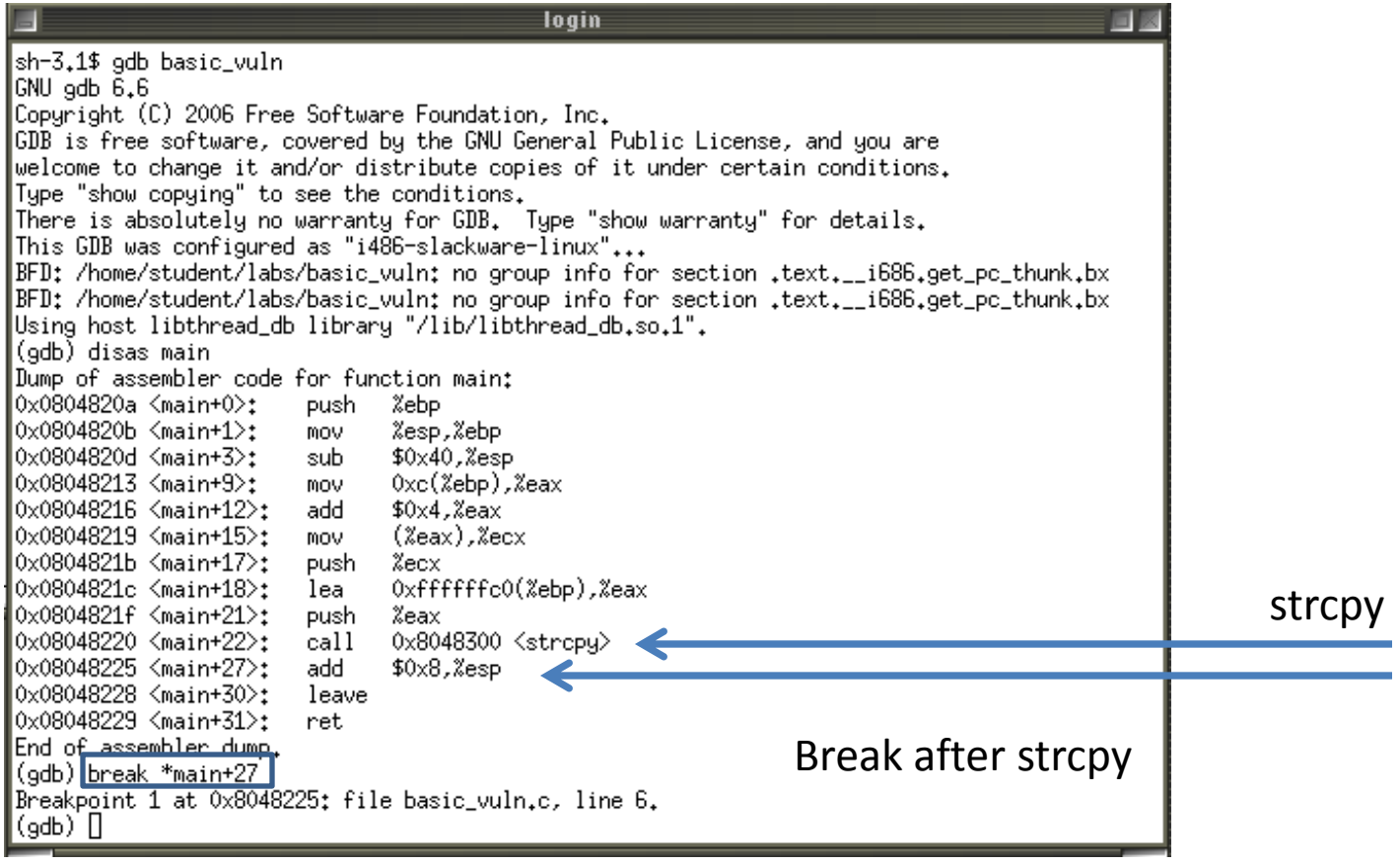
```
#include <stdio.h>

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 ‘AAAAA’



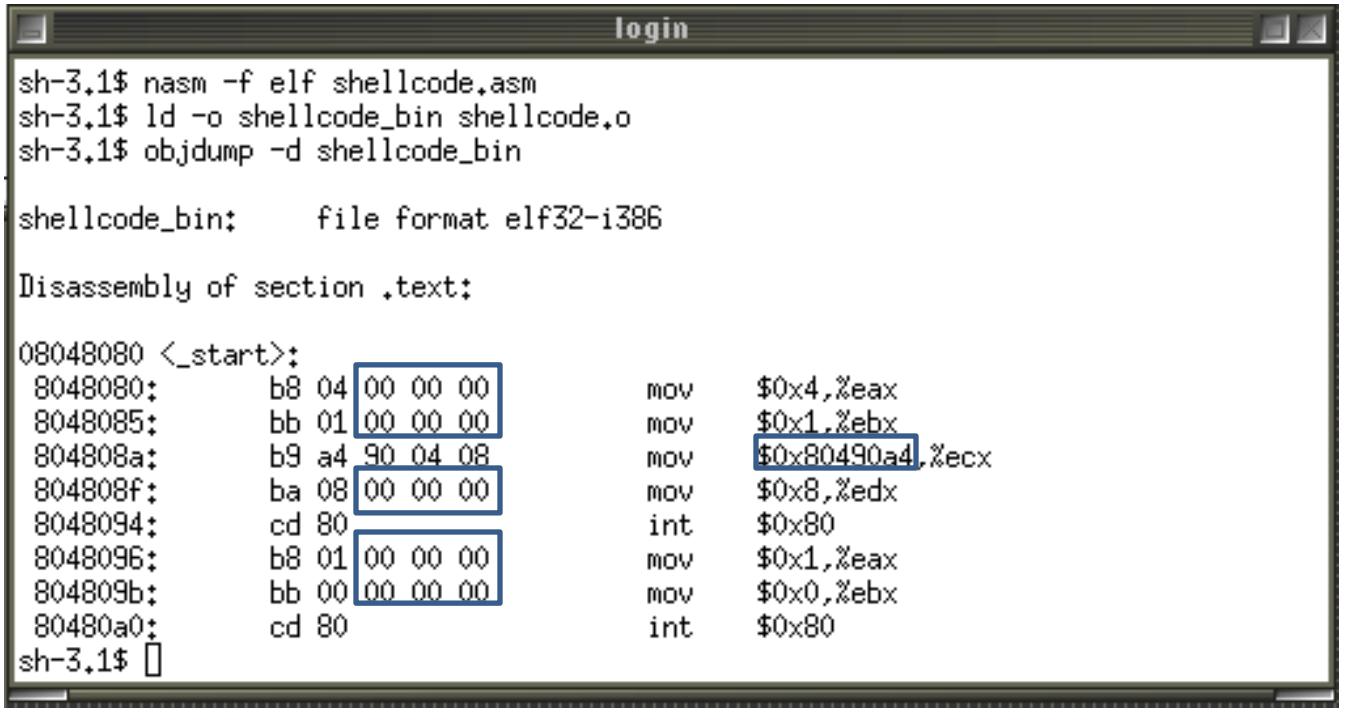
Write Some Shellcode (Hello World)

```
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



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

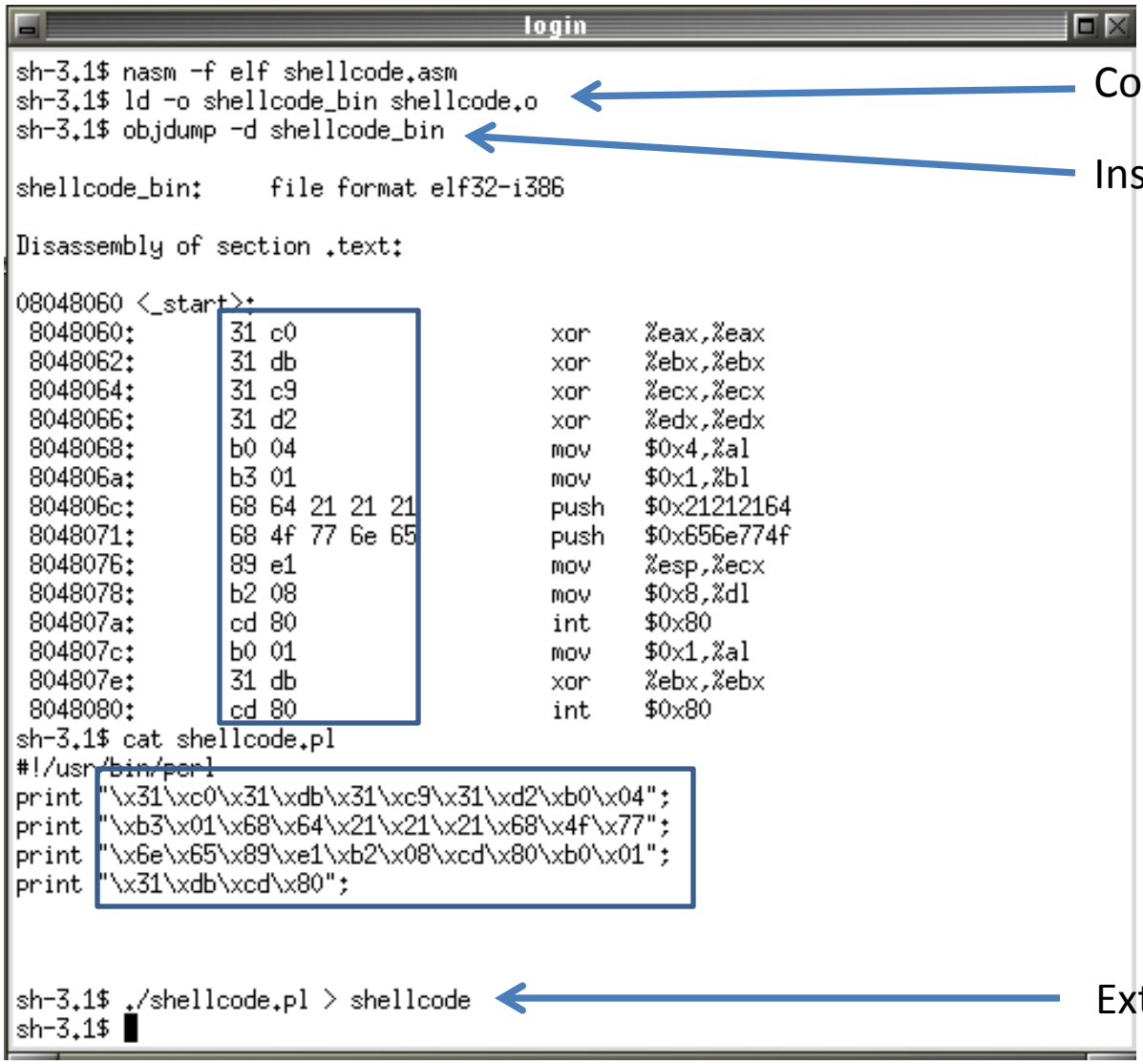
Shellcode Cleanup Tricks

- 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



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)

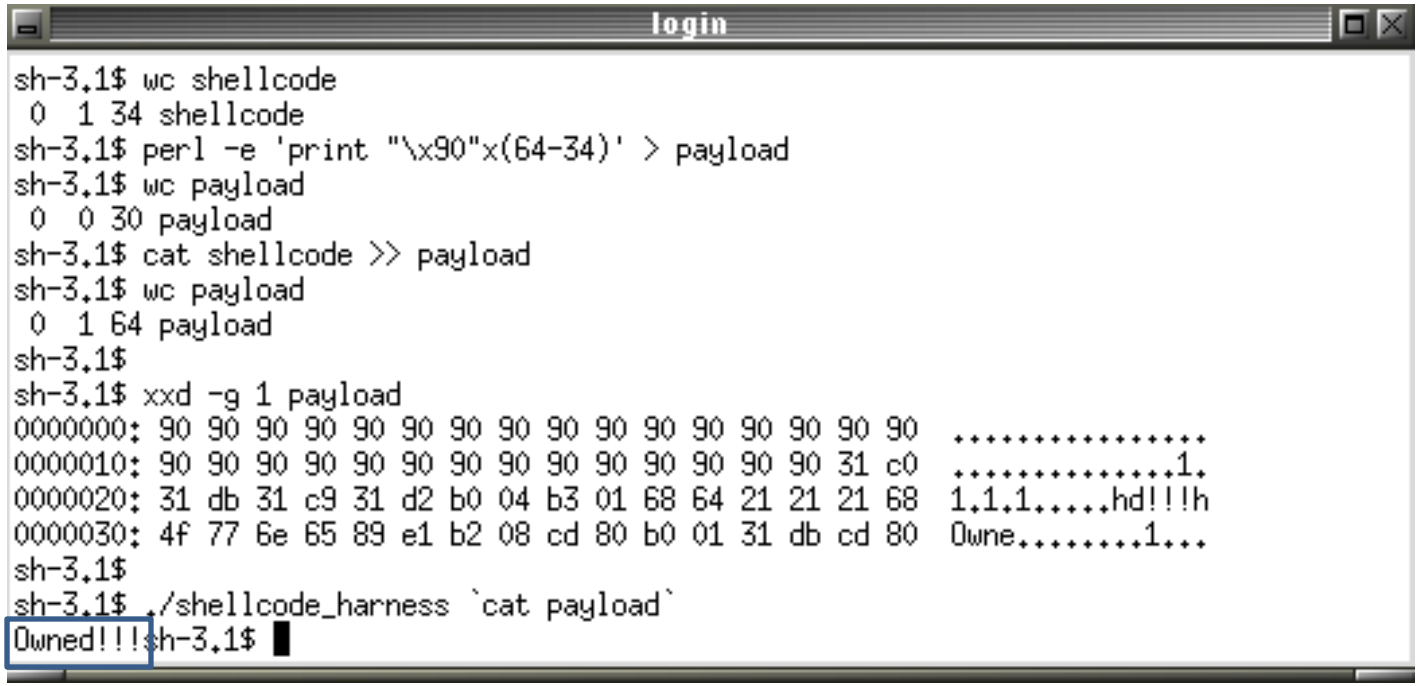
- Use "wc <filename>" command to get file size
- perl –e 'print "\x90"x(64-34)' > payload
- cat shellcode >> payload

Build and Test the Payload

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

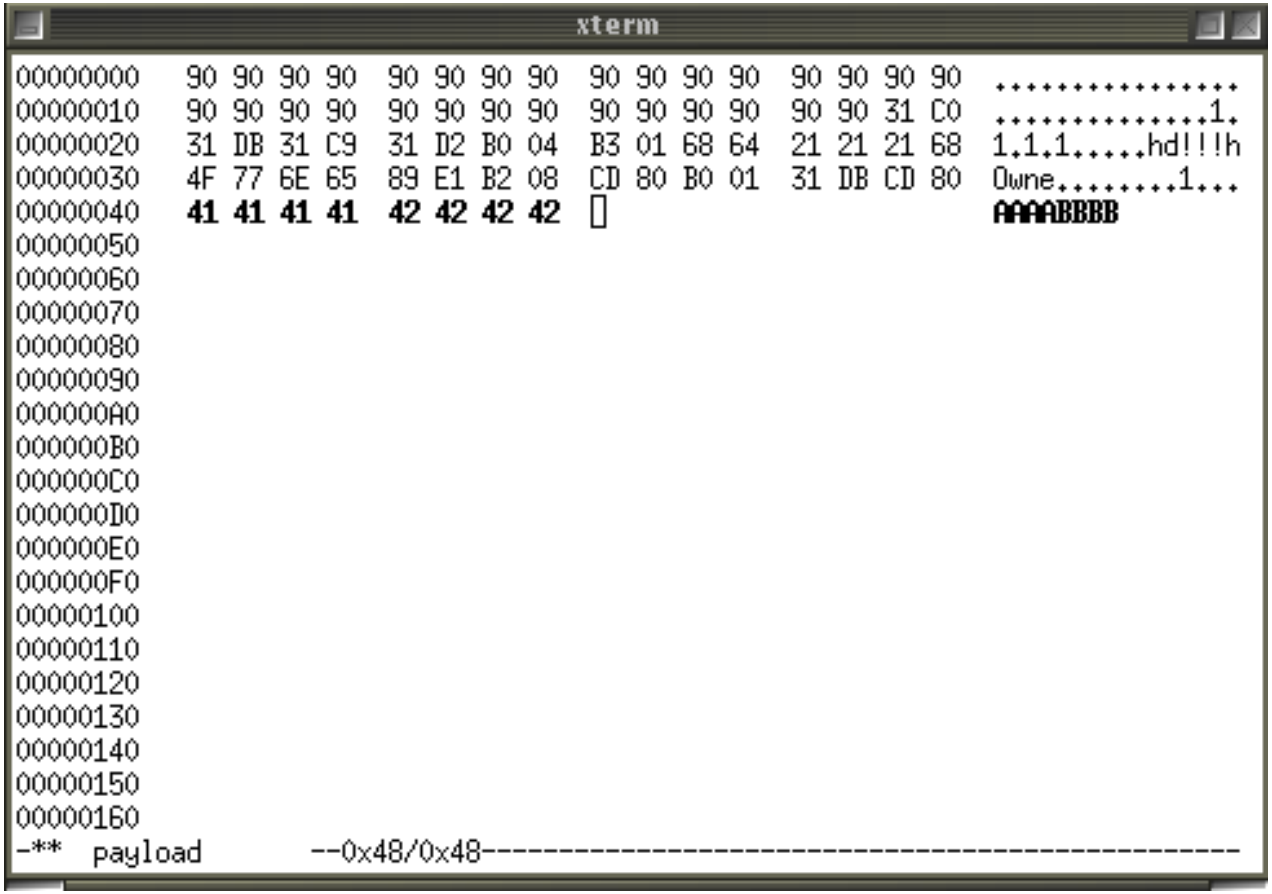
Returns main to the argv buffer, forcing the CPU to execute data passed in the program arguments…

Probably not a best practice…

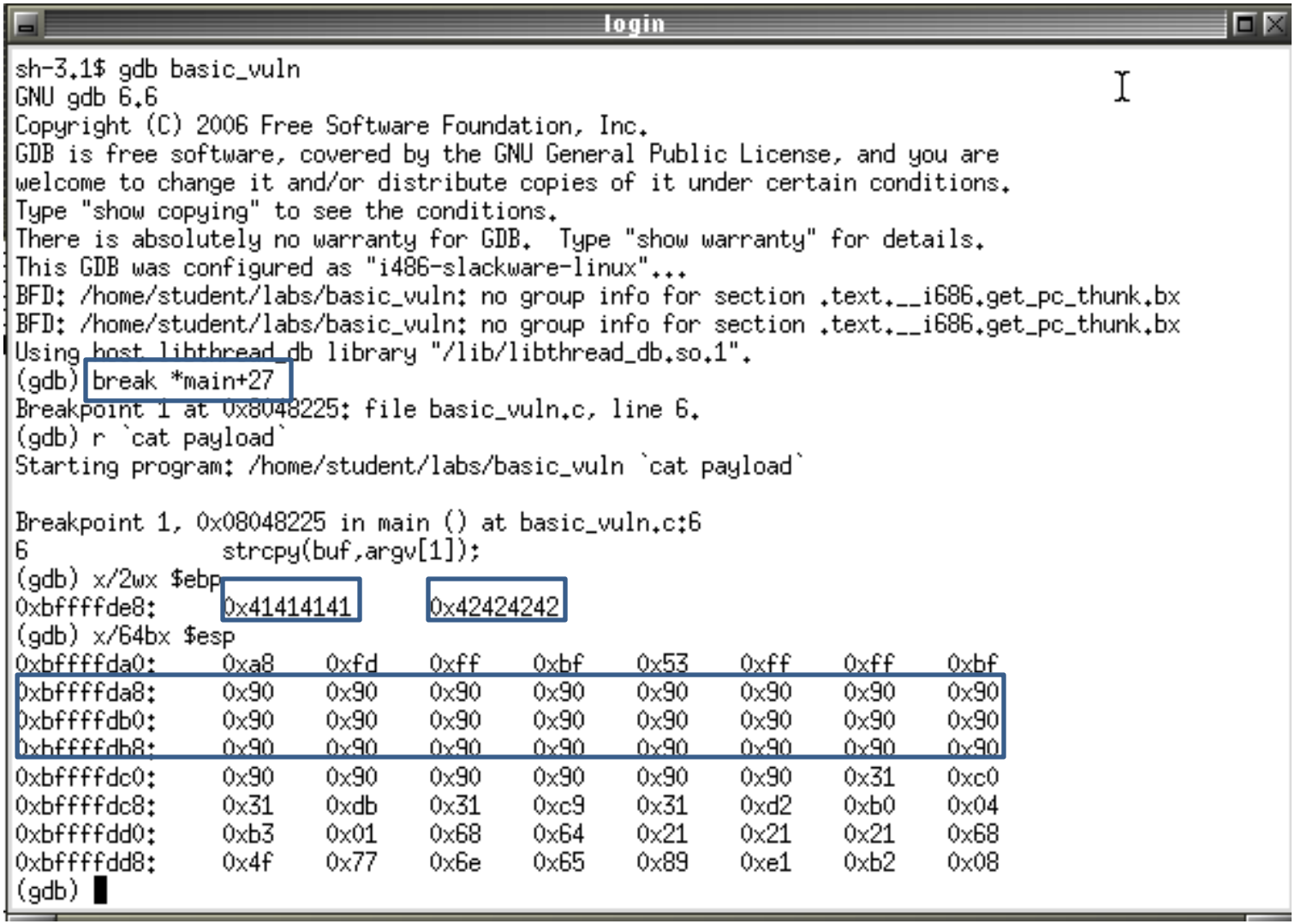


Find and Overwrite the Stack Pointer

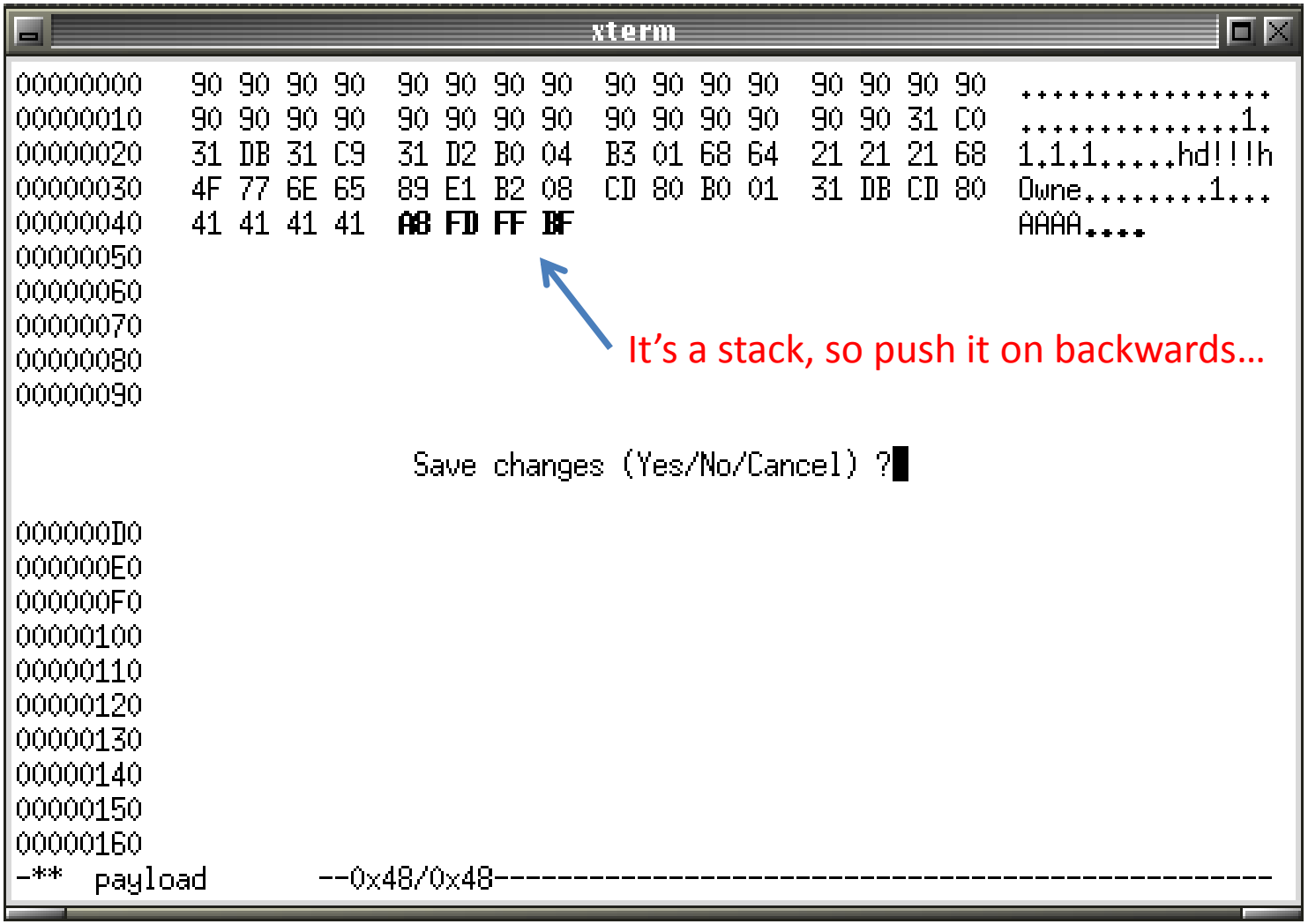
- "hexedit payload"
  - Add some visible bytes and inspect in gdb



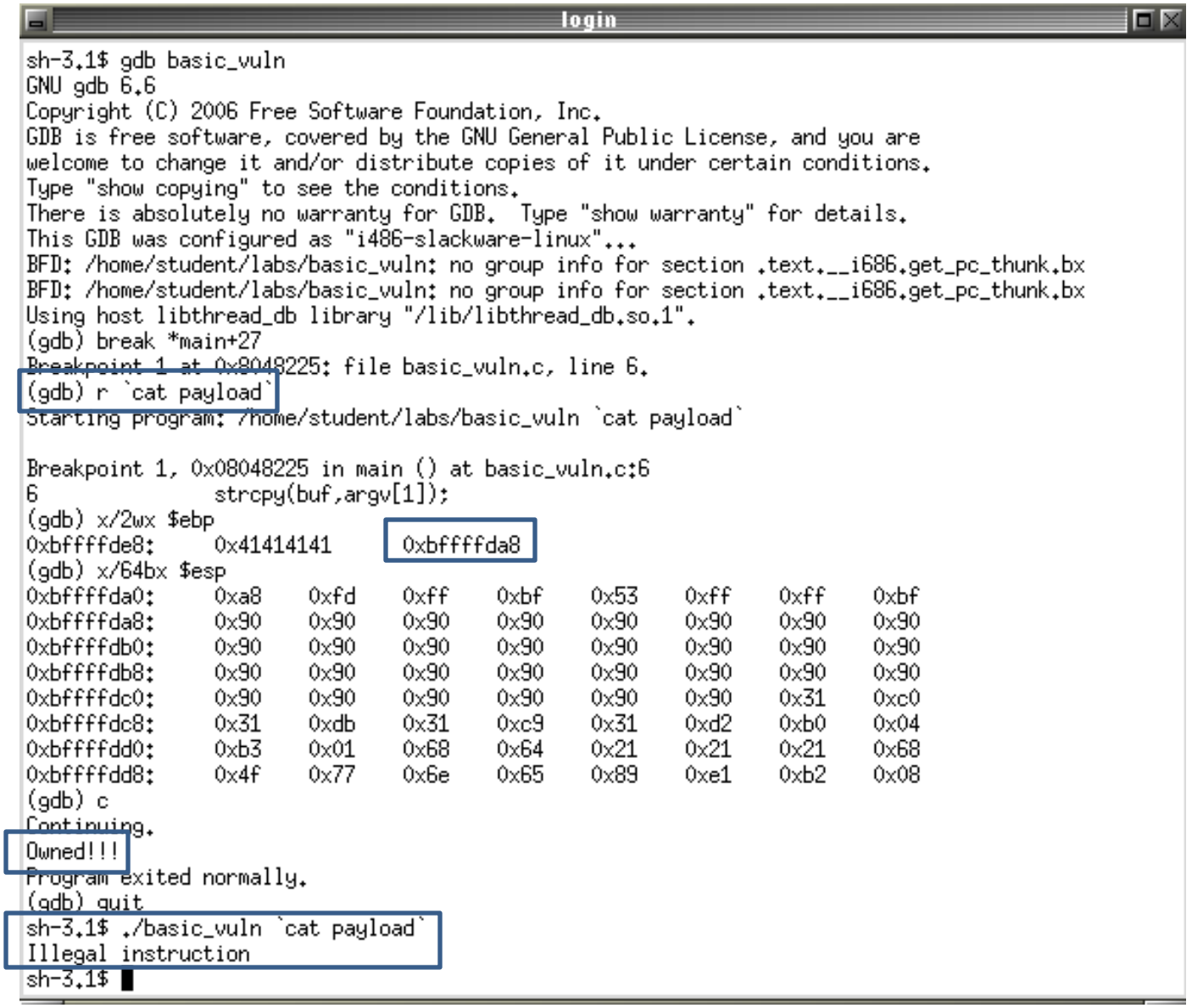
Find and Overwrite Stack Pointer (cont.)



Overwrite Stack Pointer



Exploited! (sorta…)



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

login

```
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.__i686.get_pc_thunk.bx
BFD: /home/student/labs/basic_notvuln: 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:
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    0xffffffc0(%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.
<AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Starting program: /home/student/labs/basic_notvuln AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Breakpoint 1, 0x0804824b in main () at basic_notvuln.c:6
6               strncpy(buf,argv[1],64-1);
(gdb) x/64bx $esp
0xbffffbec:     0xf8    0xfb    0xff    0xbf    0xa3    0xfd    0xff    0xbf
0xbffffbf4:     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      0xb7e9bdf8
(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]

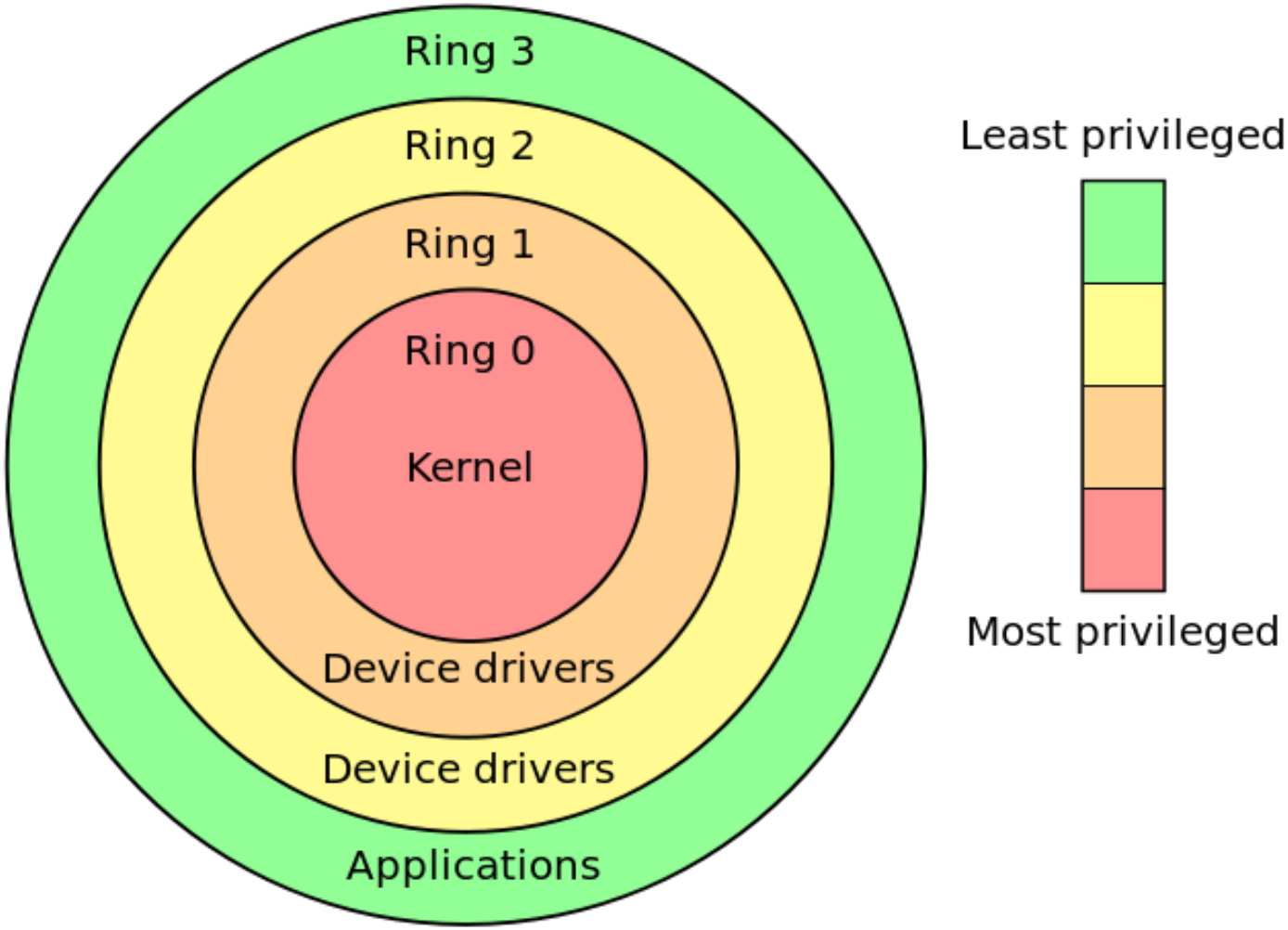
Payload = [NOP Sled + Shellcode]

- Payload is the length of the buffer
  - sizeof(NOP Sled) = sizeof(buffer) minus 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
vsprintf	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.

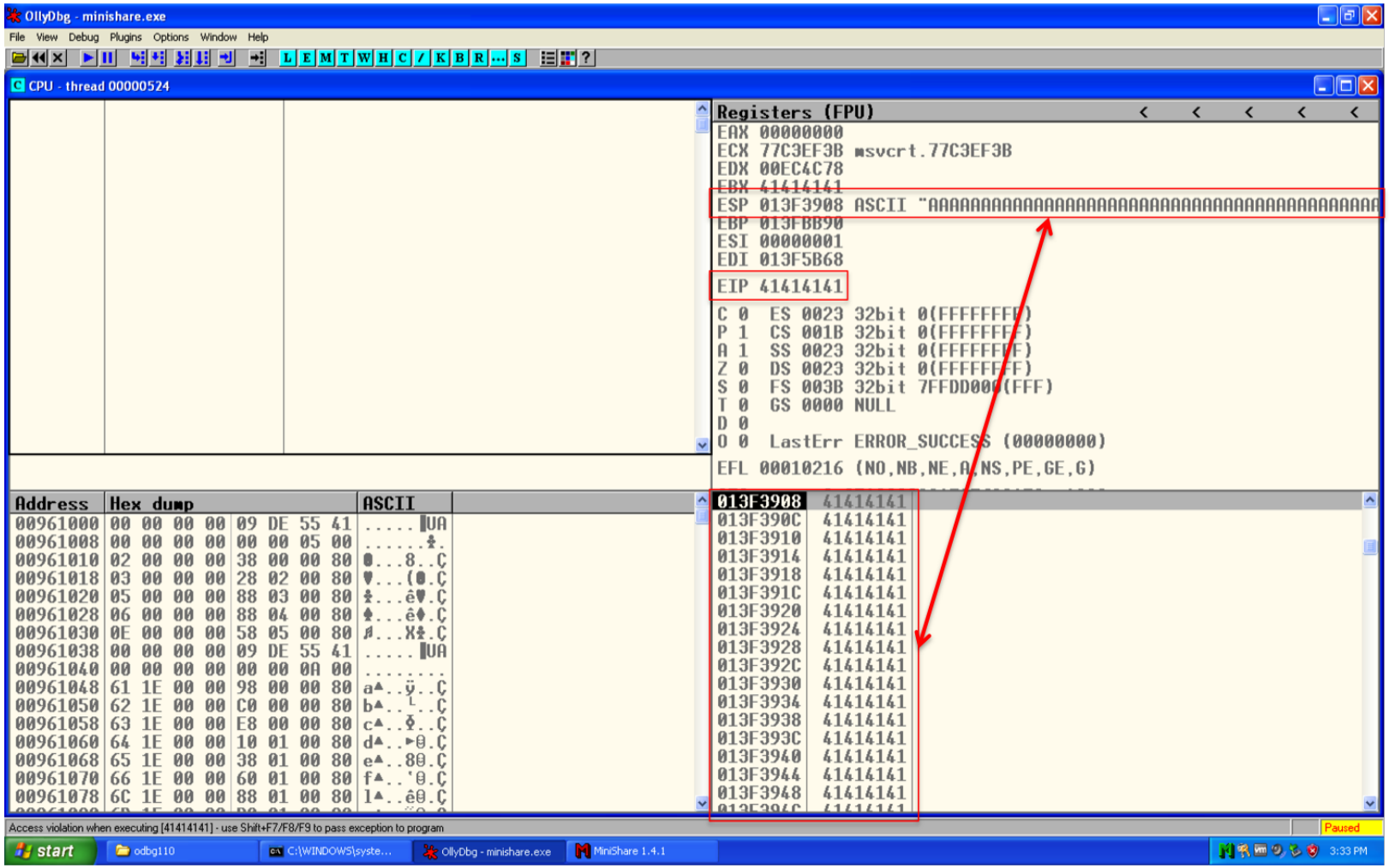
Reference:
https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2004-2271

Demo

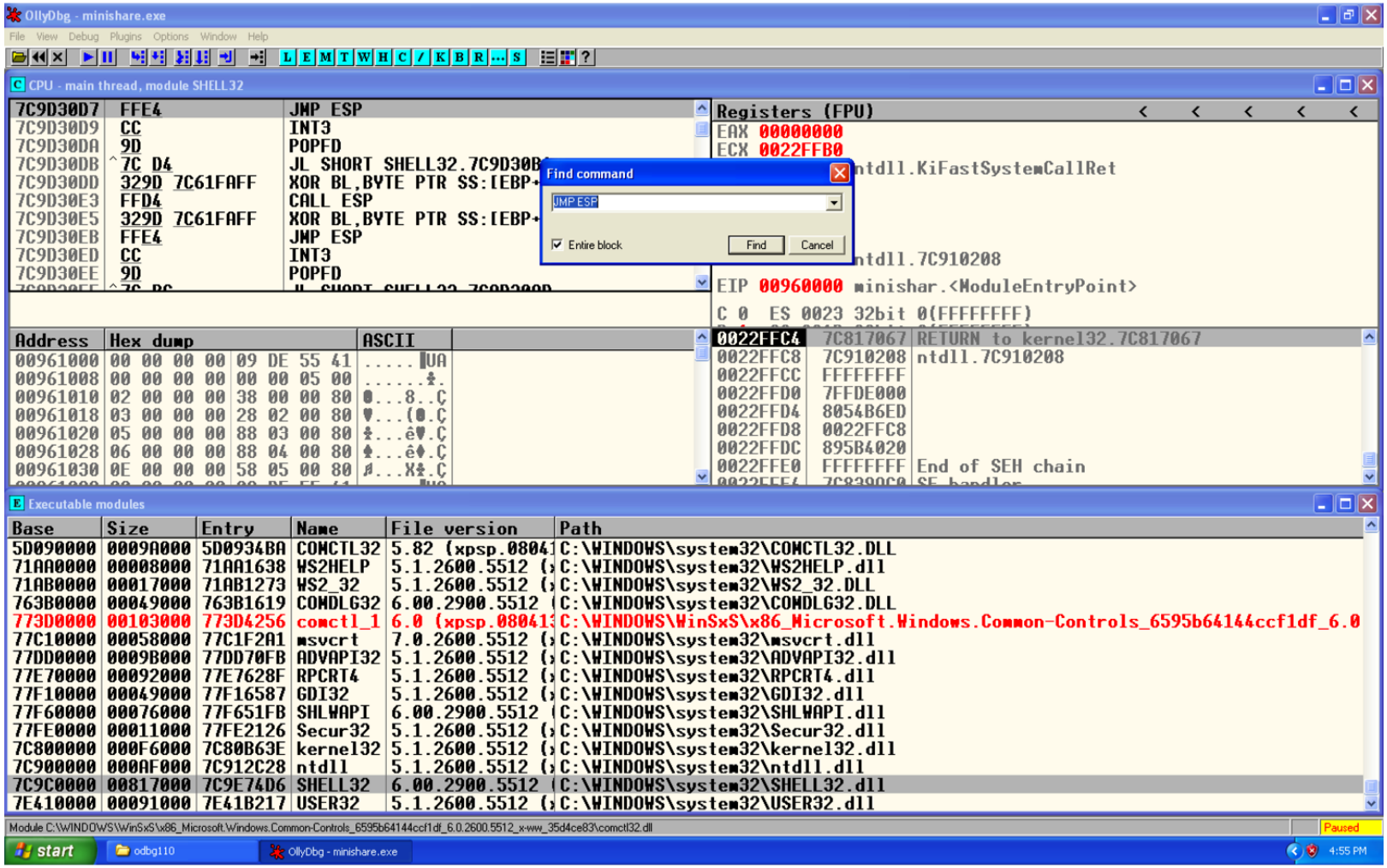
Demo building exploit for MiniShare 1.4.1

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

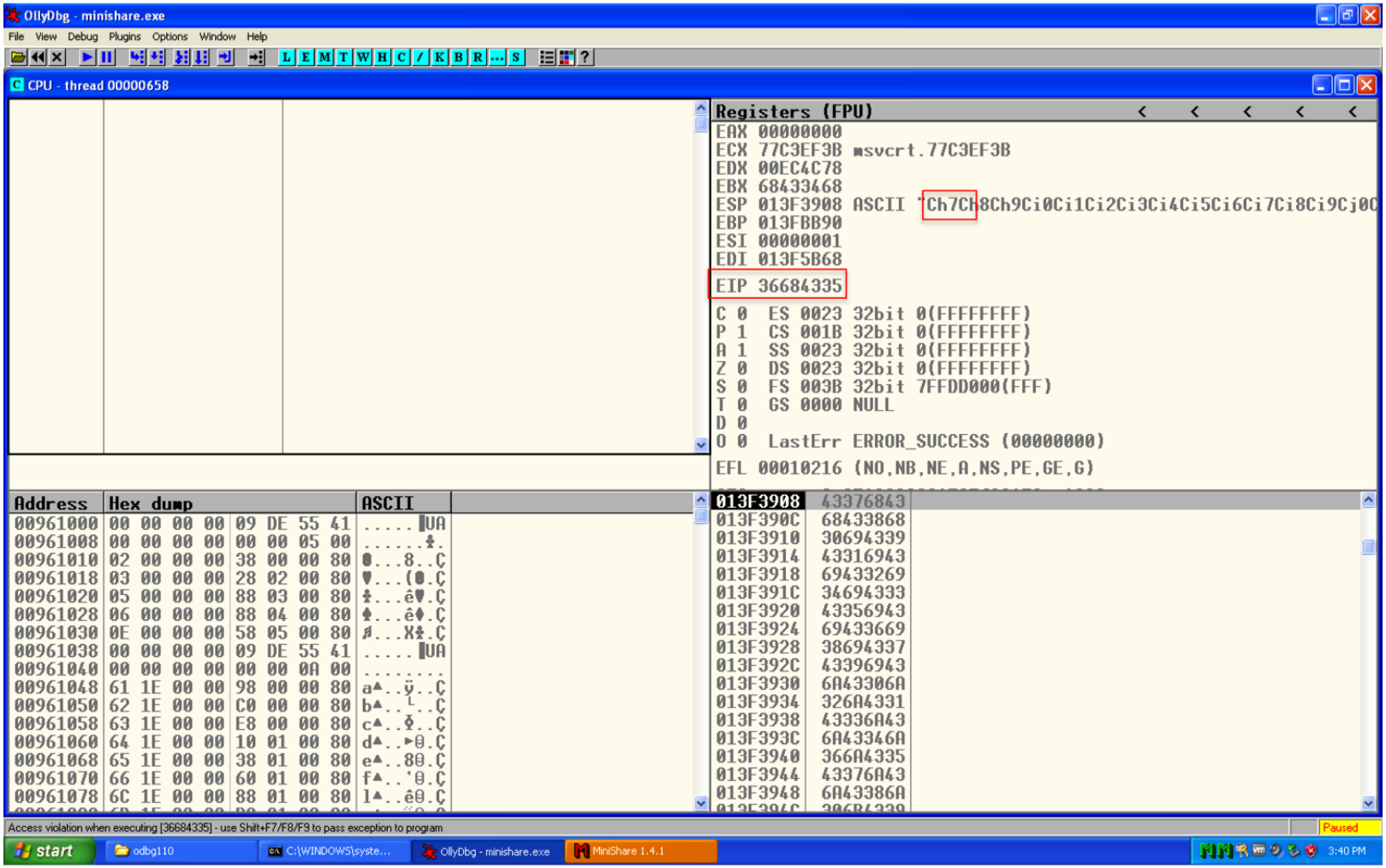
Reproduce Error



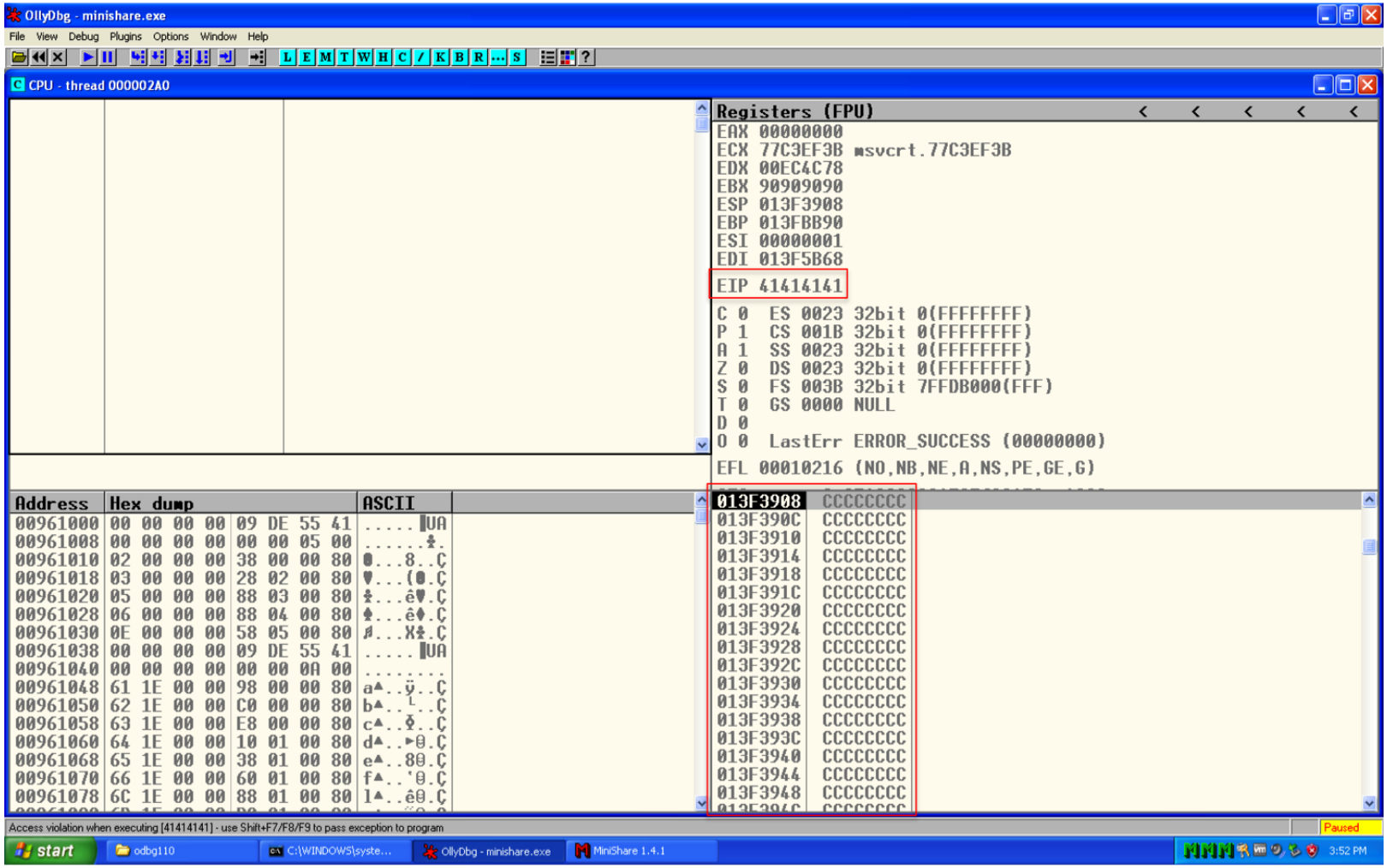
Find a JMP ESP Instruction



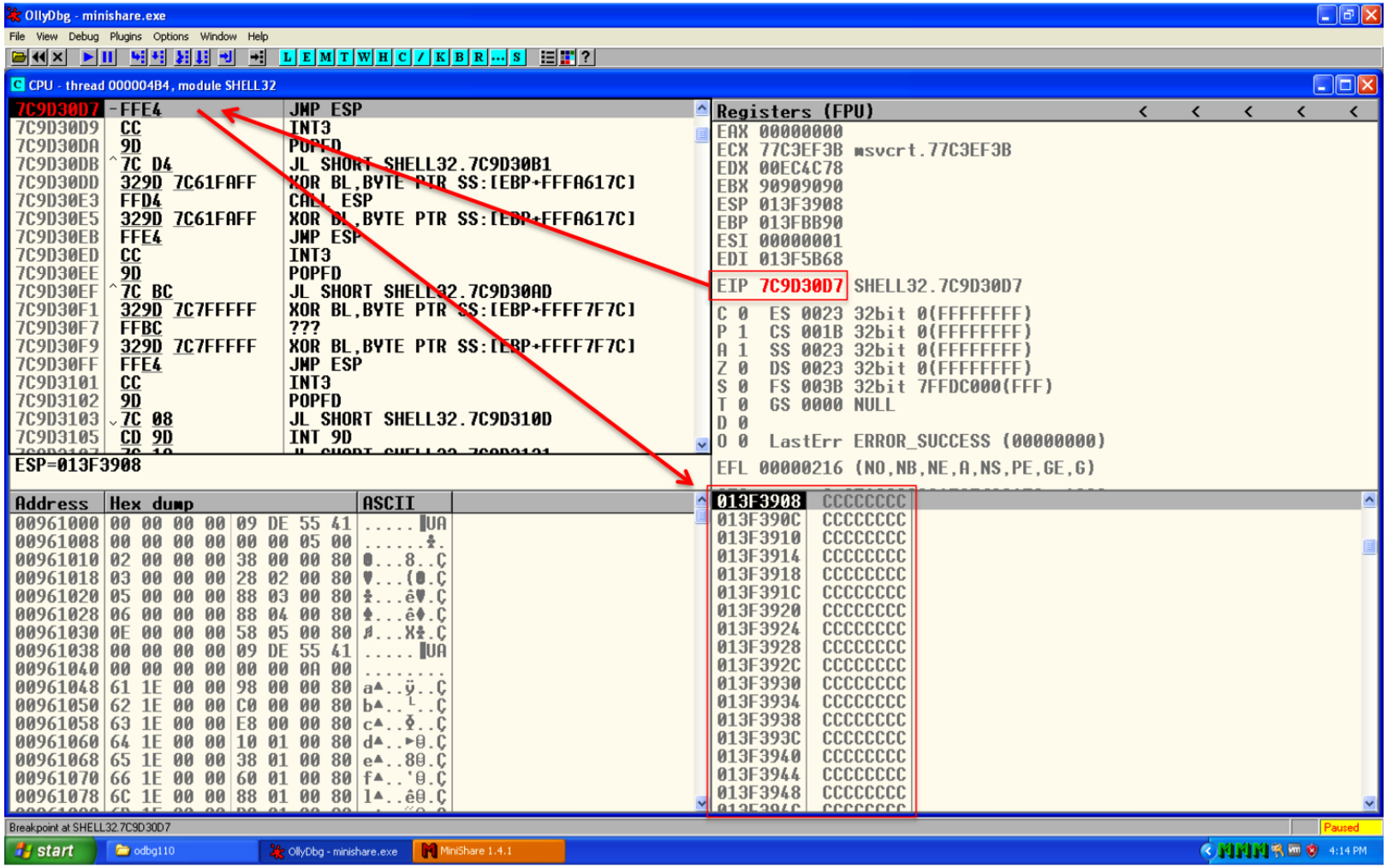
Calculate Overwrite Offsets



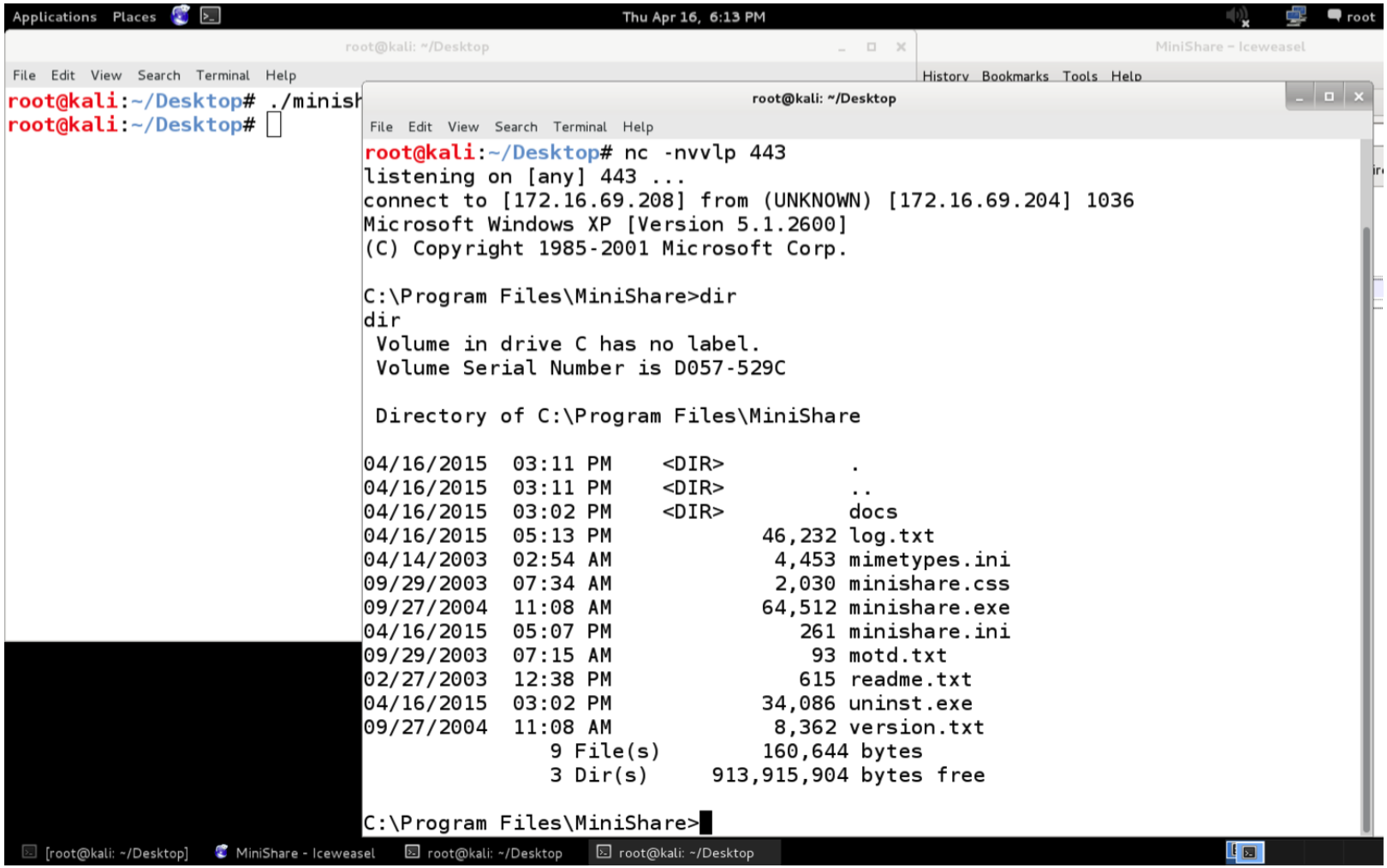
Test Offsets



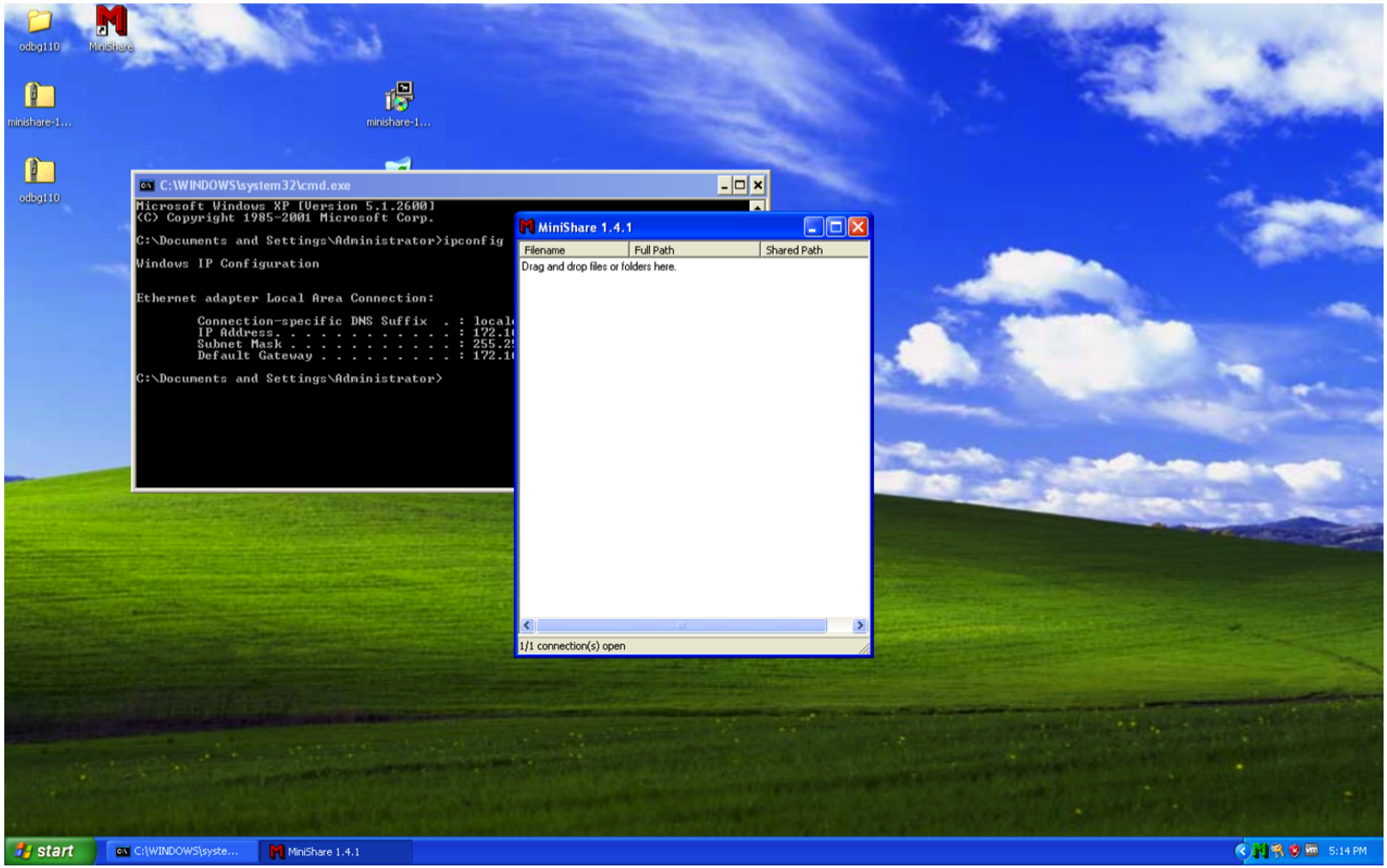
Jump to Shellcode



Execute Reverse TCP Shell



Silent Attack



Exploit

- len(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
- Futher 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 place 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 segementation 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 soley 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/danging pointers (pointers that don't point where they are suppose to be pointing)