

Shellcode Development

What is a program?

A program is just a set of instructions (assembly) with variables

- Where are the instructions stored?
- What if those instructions could be overwritten?
- What if the *pointer* to those instructions could be redirected?

Attacker Goal

To execute arbitrary code from a victim program

1. Open a new shell from a running program
 - Or a **Root shell** if the victim program is setuid root
 - The instructions to open a new shell is “shellcode”
2. Once you have a shell, you can execute arbitrary code
 - Add a new user with administrative privileges
 - Point “google.com” to an attacker server
 - ...

Attacker Steps

1. Target a **vulnerable** program
2. Construct **shellcode** to attack the victim program
3. **Inject** this code in the normal flow of the program
 - To do this, an attacker needs to find a vulnerable function
 - Details next week!

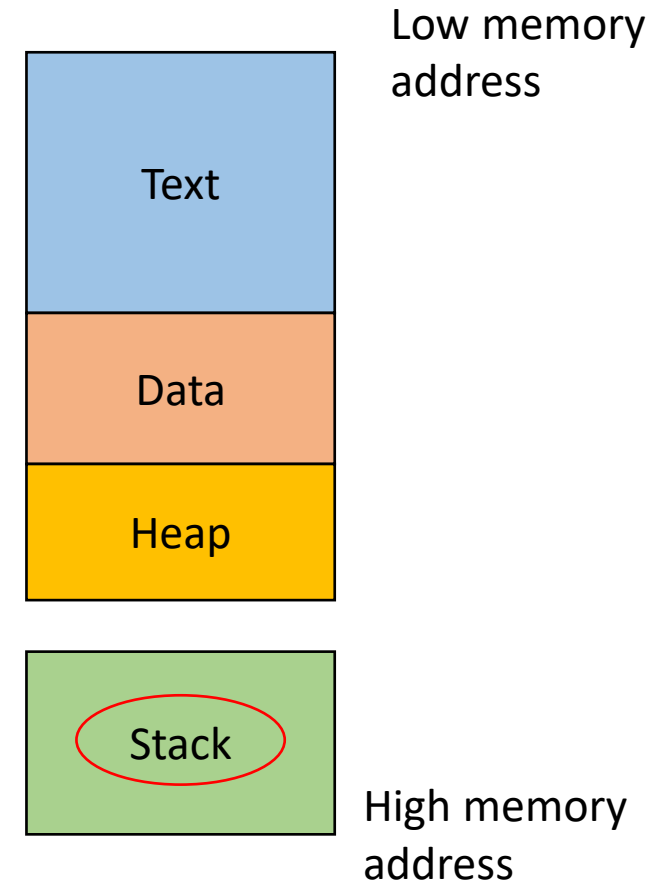
This week's goal

1. Assembly revision
2. Understanding program flow and memory stack
3. Constructing shellcode
 - No injection yet – we will just call it directly



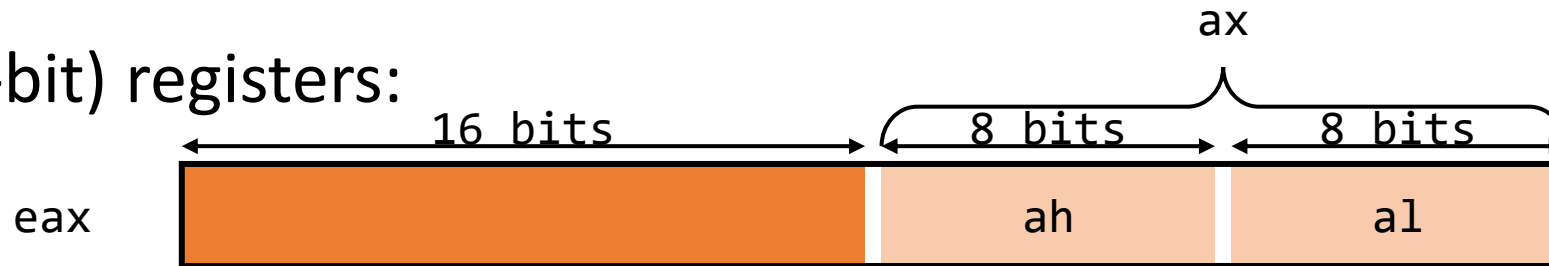
Process Memory Organization

- A process is divided into four regions.
- Text
 - Fixed region
 - Includes instructions and Read-only data
- Data
 - Initialized and uninitialized data
- Heap (dynamic vars)
- Stack (LIFO abstraction)
 - Maintains state of caller/callee of functions
 - Used for storing:
 - Local variables
 - Parameters
 - Return value

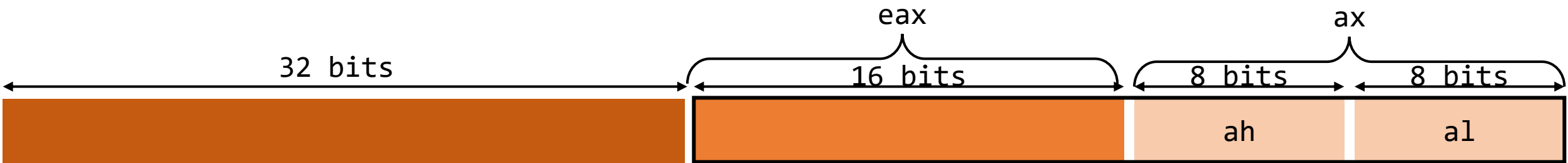


Registers – General-purpose

- x86 (32-bit) registers:

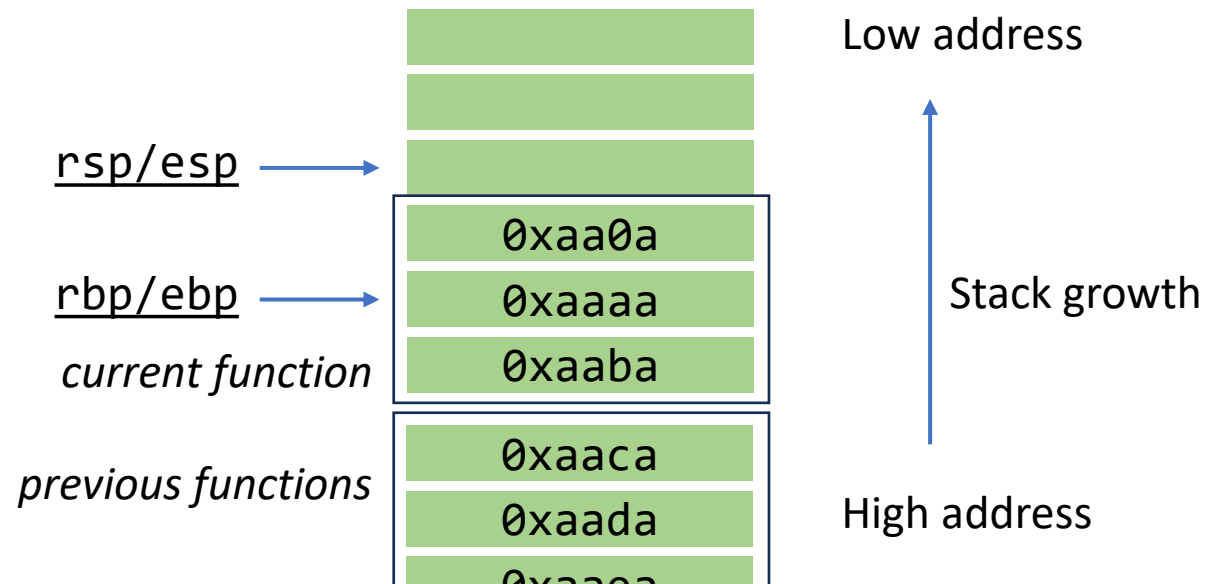


- Also: `ebx`, `ecx`, `edx`, `esi`, `edi`
 - `eax`..`edx` are used as arguments to syscalls
- x64 (64-bit) registers (also `rbx`, `rcx`, ...):



Registers – Special-purpose

- `rsp/esp`: Stack pointer. Points to the top of the stack (**lowest** possible memory address)
- `rbp/ebp`: Base pointer. Points to the “center” of the current function
- Most labs are in x64 but some are in x86 – we will use x86 terminology due to widely available material



Sample x86 Instructions – Arithmetic/Logic

`add dst, src`

`inc dst`

`xor dst, src`

Sample x86 Instructions – Data Movement

`mov dst, src`

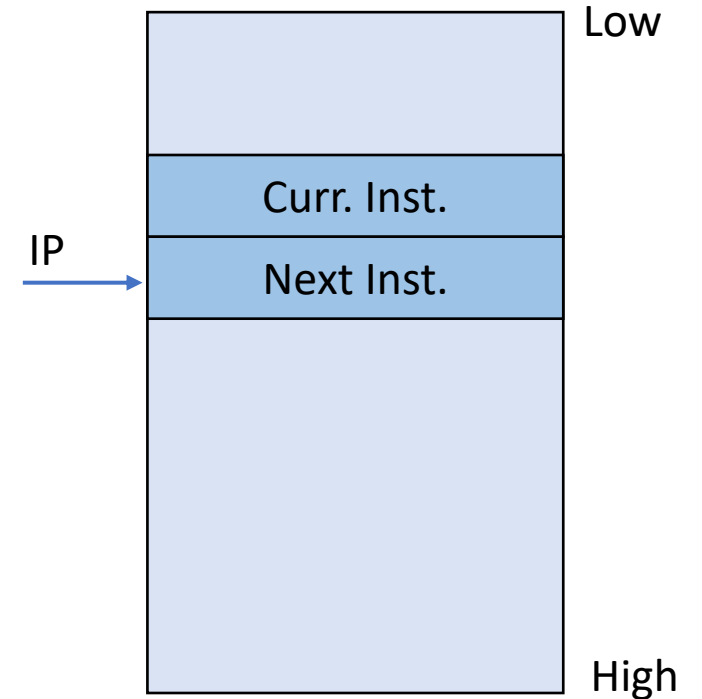
`push src`

`pop dst`

Control Flow

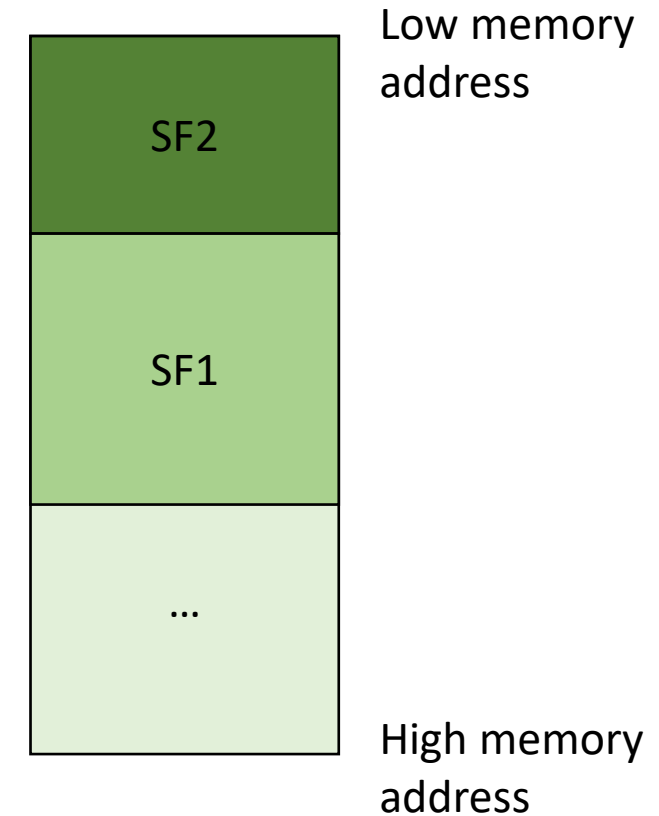
- A special register called IP:
 - Points to the next instruction to be executed
 - Cannot be **directly** altered
- CPU increments IP unless it executes an inst. that changes the flow of control, e.g.,:
 - jmp, jne, jeq, ...
 - call
 - ret
 - ...

(More on call and ret later)



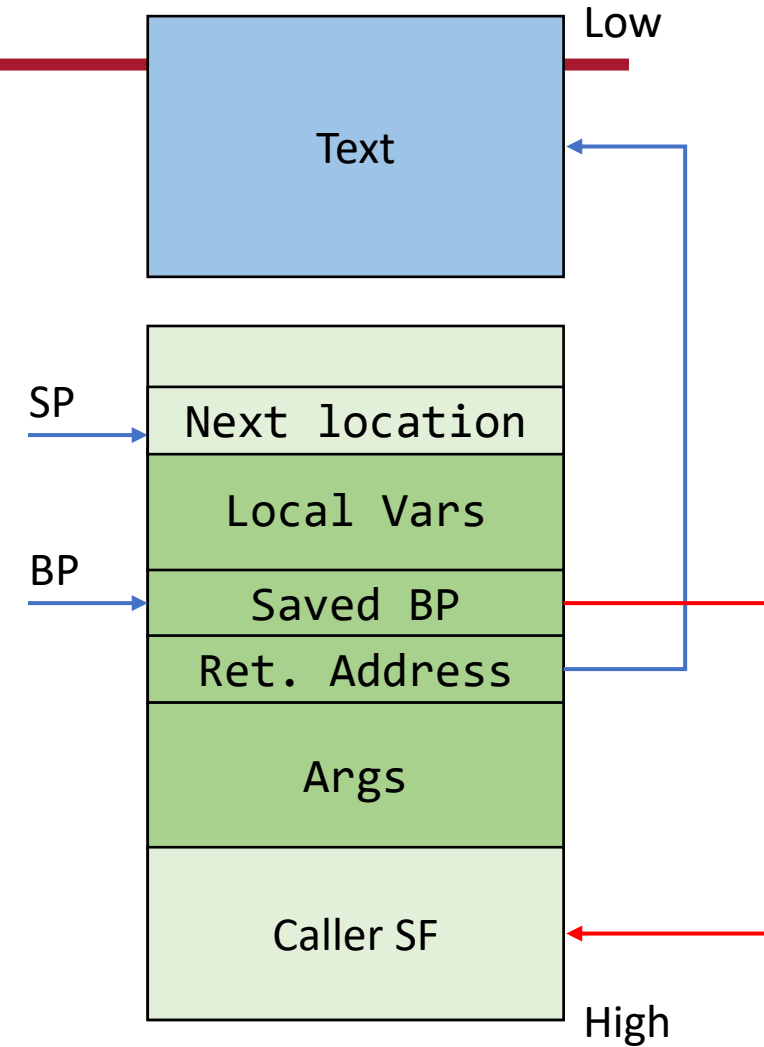
Stack Region

- Contiguous block of memory containing data
- 1 Stack Frame = 1 function call
- Every Stack Frame is:
 - Pushed when calling a function
 - Popped when returning
- Stack Frame (activation record) contains:
 - the parameters to a function,
 - its local variables, and
 - the data necessary to recover the previous stack frame



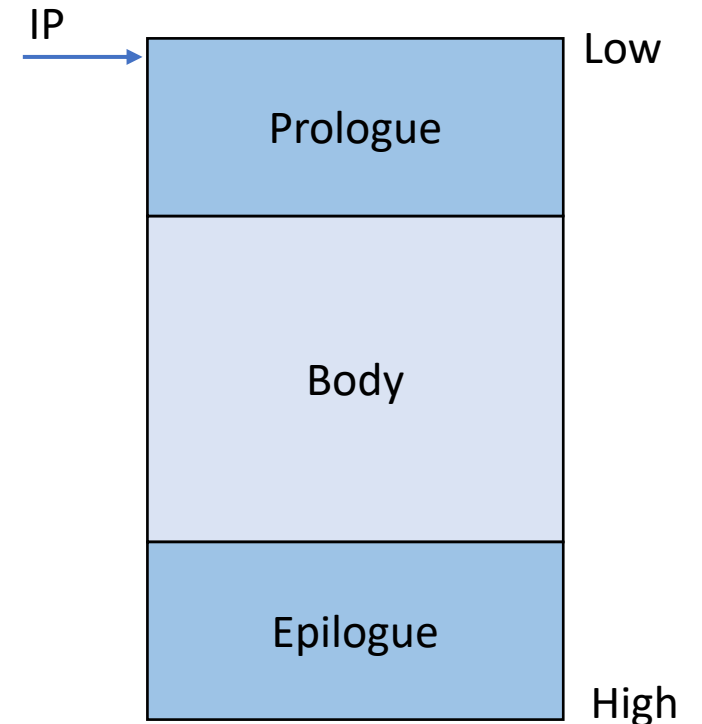
Stack Region

- Two pointers:
 - BP: points to a fixed location of a stack frame
 - SP: points to the top of the stack



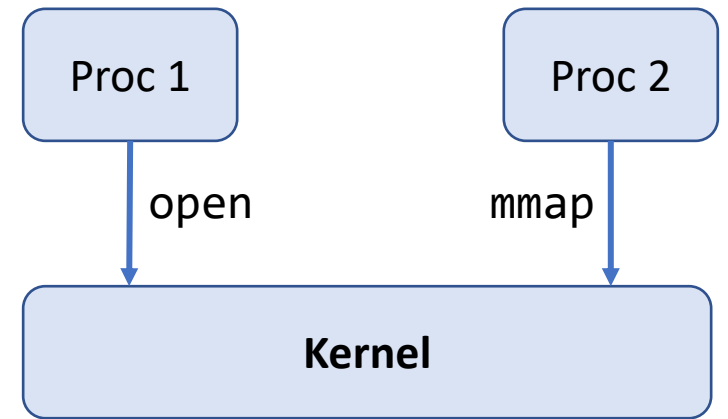
Functions: Calling Conventions

- A function **prologue**:
 - maintains a snapshot of ebp
 - push ebp to the stack
 - copy esp to ebp
 - allocates local variables by decrementing esp
 - saves register values on the stack
- A function **epilogue**:
 - recovers register values from the stack
 - deallocates the local variables by resetting esp
 - recovers the caller's ebp
 - calls ret



System Calls

- User-space programs often need services from the kernel:
 - IO (open, read, write, ...)
 - Modify address space (mmap, sbrk, ...)
- These programs trigger the kernel to perform these operations by using System Calls.
 - “Software Interrupts”



System Calls

- Move arguments to ebx, ecx, edx, esi, edi
 - Move syscall number to eax
 - `int 0x80`
-
- We need two pieces of info.
 - System call number
 - System call interfaces

Linux System Call Table

- System call numbers:
https://github.com/torvalds/linux/blob/master/arch/x86/entry/syscalls/syscall_32.tbl

NOTE: Different table for x64!

0	i386	restart_syscall	sys_restart_syscall
1	i386	exit	sys_exit
2	i386	fork	sys_fork
3	i386	read	sys_read
4	i386	write	sys_write
5	i386	open	sys_open
6	i386	close	sys_close
7	i386	waitpid	sys_waitpid
8	i386	creat	sys_creat
9	i386	link	sys_link
10	i386	unlink	sys_unlink
11	i386	execve	sys_execve
12	i386	chdir	sys_chdir
13	i386	time	sys_time32
14	i386	mknod	sys_mknod
15	i386	chmod	sys_chmod
16	i386	lchown	sys_lchown16

Linux System Call Table

- System call interfaces:

<https://github.com/torvalds/linux/blob/master/include/linux/syscalls.h>

```
asmlinkage long sys_exit(int error_code);  
asmlinkage long sys_read(unsigned int fd, char __user *buf, size_t count);  
asmlinkage long sys_write(unsigned int fd, const char __user *buf,  
                           size_t count);
```

ELF

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 00
20:
   34 00 20 00 01 00
40: 01 00 00 00 00 00 00 00 00 00 00 00 08 00 00 00 08
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	
EI_MAG	0x7F, "ELF"
EI_CLASS, EI_DATA	1ELFCLASS32, 1ELFDATA2LSB
EI_VERSION	1EV_CURRENT
e_type	2ET_EXEC
e_machine	3EM_386
e_version	1EV_CURRENT
e_entry	0x8000060
e_phoff	0x0000040
e_ehsize	0x0034
e_phentsize	0x0020
e_phnum	0001
p_type	1PT_LOAD
p_offset	0
p_vaddr	0x8000000
p_paddr	0x8000000
p_filesz	0x0000070
p_memsz	0x0000070
p_flags	5PF_R PF_X

PROGRAM HEADER TABLE

EXECUTION INFORMATION

CODE

X86 ASSEMBLY

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

EQUIVALENT C CODE

```
return 42;
```

Syntax

- AT&T syntax

```
mov $42, %ebx
```

```
mnemonic source, destination
```

- Intel syntax

```
mov ebx, 42
```

```
mnemonic destination, source
```

We will use the Intel syntax

Required tools

- gcc
 - gdb
 - ld
 - nasm (<https://www.nasm.us/>)
 - objdump
- *nasm and objdump can understand AT&T and Intel syntax*

Three Examples

- `mini`
 - `HelloWorld`
 - `Spawn a Shell`
-
- We will:
 - write the assembly code, get its machine code, and call it from a C program

Techniques

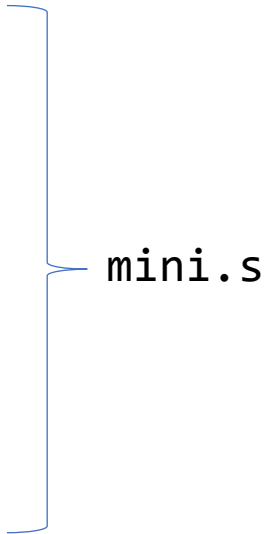
- Relative Addressing (or using code segment)
- Pushing data to stack
- Enable Privileges
- Shellcode Copying
- Reducing shellcode size (why?)

mini

- `exit(42)`

mini

```
[SECTION .text]
global _start
_start:
    mov ebx, 42
    mov eax, 0x1
    int 0x80
```



mini.s

```
$ nasm -f elf mini.s # creates an object file, mini.o
$ ld -o mini mini.o # runs the linker
$ ./mini # executes the program
$ echo $? # print status of mini
$ 42 # output
```

Disassemble mini

```
$ objdump -Mintel --disassemble mini
```

```
mini:      file format elf32-i386
```

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

```
08048060 <_start>:
```

8048060:	bb 2a 00 00 00	mov	ebx, 0x2a
8048065:	b8 01 00 00 00	mov	eax, 0x1
804806a:	cd 80	int	0x80

```
mini executable bytes are: bb 2a 00 00 00 b8 01 00 00 00 cd 80
```

helloworld

- How many syscalls?

helloworld

- Two syscalls: write and exit

```
[SECTION .data]
    msg db "Hello, world!", 0xA, 0xD
[SECTION .text]
global _start
_start:
    mov eax, 4      ; opcode for write system call
    mov ebx, 1      ; 1st arg, fd = 1
    mov ecx, msg    ; 2nd arg, msg
    mov edx, 15     ; 3rd arg, len
    int 0x80        ; system call interrupt

    mov eax, 1      ; opcode for exit system call
    mov ebx, 0      ; 1st arg, exit(0)
    int 0x80        ; system call interrupt
```

Shellcode

- The set of instructions injected and then executed by an exploited program
 - usually, a shell is started
 - for remote exploits - input/output redirection via socket – use system call (execve) to spawn shell
- Shellcode can do practically anything:
 - 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



Testing shellcode

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

char code[] = "bytecode will go here!";
int main(int argc, char **argv)
{
    int (*func)();
    func = (int (*)(())) code;
    (int)(*func)();
}
```

```
$ gcc code.c -o output -fno-stack-protector -z execstack -no-pie -m32
```



Shellocode: **mini**

```
char code[] =  
“\xbb\x2a\x00\x00\x00\xb8\x01\x00\x00\x00\xcd\x80”;
```

```
$ ./mini_shelltest  
$ echo $?  
$ 42
```

Shellocode: helloworld

```
char code[] = “\xb8\x04\x00\x00\x00\xbb\x01\x00\x00  
                \x00\xb9\xa4\x90\x04\x08\xba\x0f\x00  
                \x00\x00\xcd\x80\xb8\x01\x00\x00\x00  
                \xbb\x00\x00\x00\x00\xcd\x80”
```

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

[\

This isn't "Hello, world!\n", what happened?

Let's disassemble helloworld

```
$ objdump -d helloworld
```

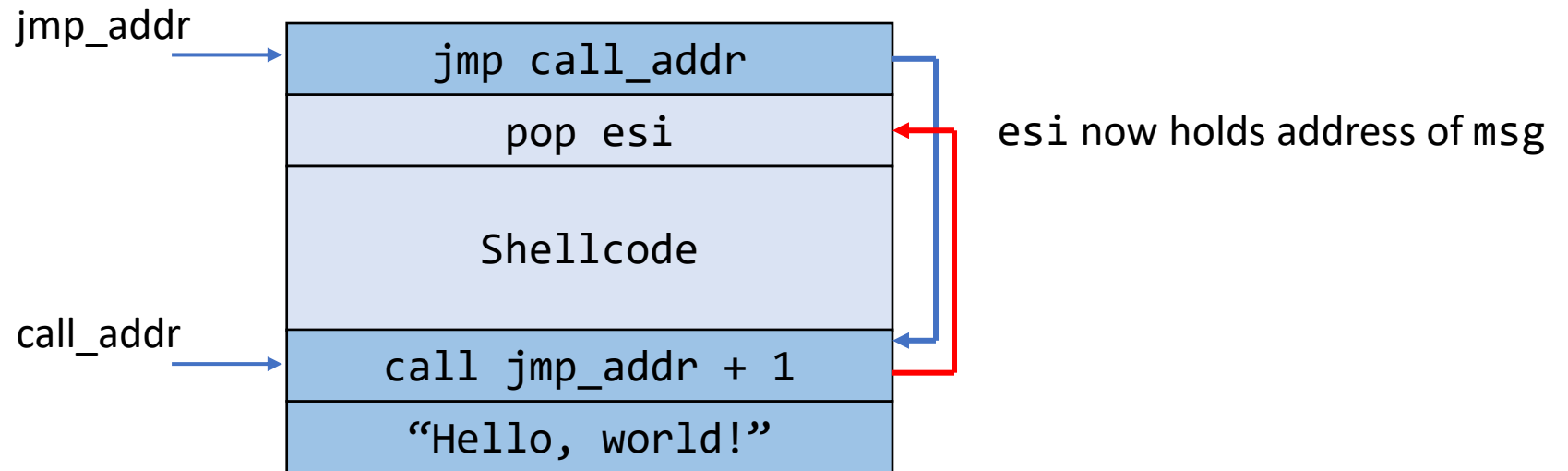
```
08048080 <_start>:
```

8048080:	b8 04 00 00 00	mov	eax, 4
8048085:	bb 01 00 00 00	mov	ebx, 1
804808a:	b9 a4 90 04 08	mov	ecx, 0x080490a4
804808f:	ba 0f 00 00 00	mov	edx, 15
8048094:	cd 80	int	0x80
8048096:	b8 01 00 00 00	mov	eax, 1
804809b:	bb 00 00 00 00	mov	ebx, 0
80480a0:	cd 80	int	0x80

Relative addressing (or Using Code Segment)

- Problem - position of code in memory is unknown
 - We cannot know the address of `msg`
- We can leverage instructions that modify the control flow
- `call` instruction saves IP on the stack and jumps
- Idea
 - `jmp` instruction at beginning of shellcode to `call` instruction
 - `call` instruction right before the string
 - `call` jumps back to first instruction after `jmp`
 - now address of "Hello, world!" is on the stack

Relative addressing (or Using Code Segment)



Shellocode: helloworld_v2

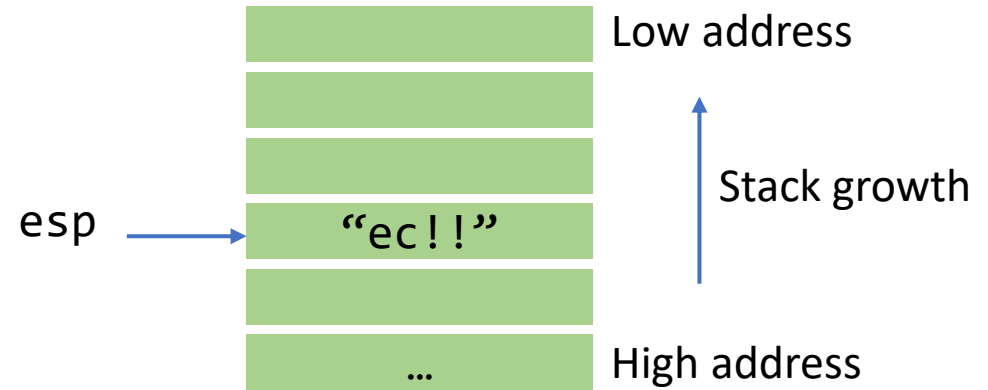
```
char code[] = "..."
```

```
$ gcc helloworld_v2.c -o shelltest_v2 -fno-stack-protector -z execstack -no-pie -m32  
$ ./shelltest_v2  
$ Hello, world!
```

Pushing Bytes to the Stack

To print “sfusec!!” (x86)

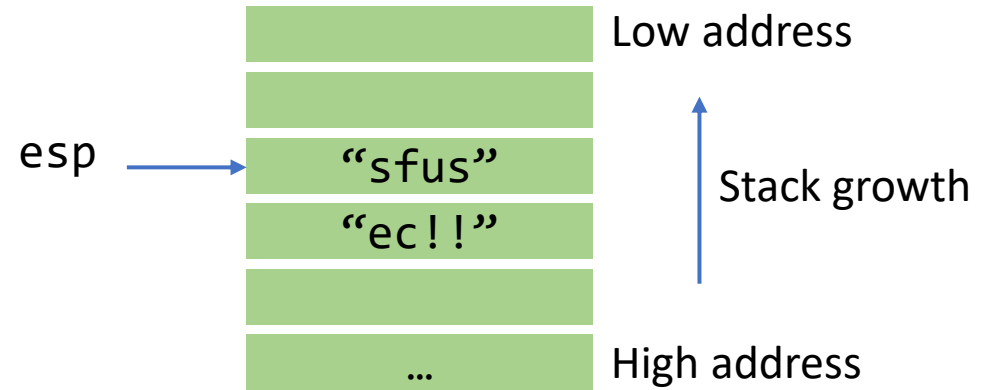
push “ec!!”



Pushing Bytes to the Stack

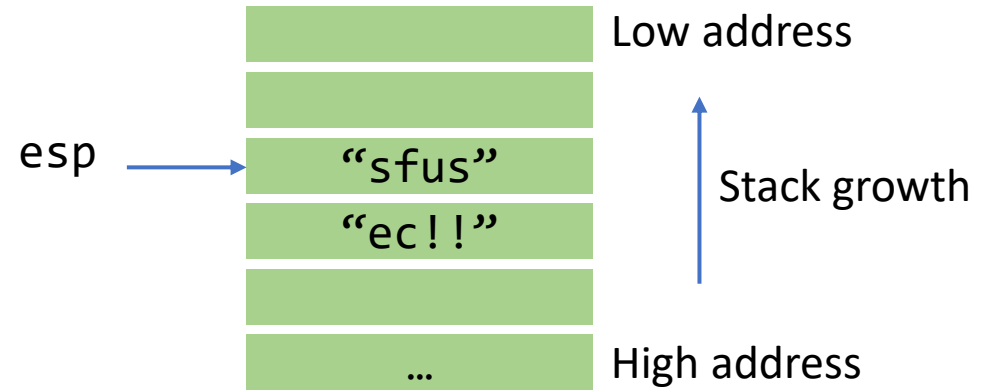
To print “sfusec!!”

```
push “ec!!”  
push “sfus”
```



Pushing Bytes to the Stack

To print “sfusec!!”



```
push “ec!!”
```

```
push “sfus”
```

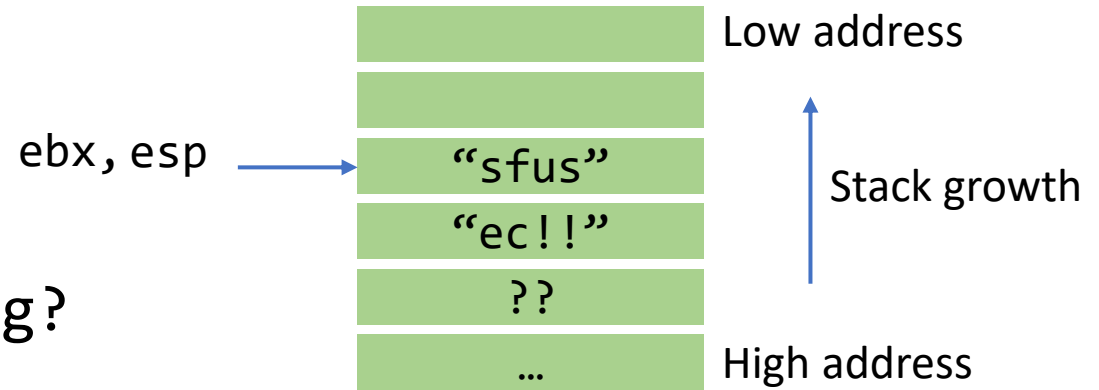
```
??
```

```
; how to get string address?
```

Pushing Bytes to the Stack

To print “sfusec!!”

```
??                ; what's missing?  
push “ec!!”  
push “sfus”  
mov ebx, esp      ; string address stored in ebx
```



Pushing Bytes to the Stack

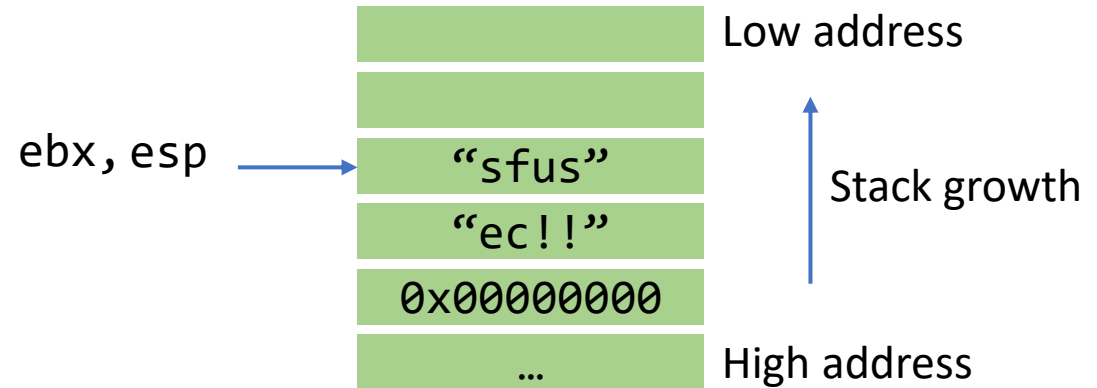
To print “sfusec!!”

```
push 0x00      ; NULL
```

```
push “ec!!”
```

```
push “sfus”
```

```
mov ebx, esp   ; string address stored in ebx
```



Spawn a Shell

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

1. file: name of program to be executed
2. argv: address of null-terminated argument array
3. env: address of null-terminated environment array

Spawn a Shell

```
#include <stdlib.h>
#include <unistd.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);
}
```

Spawn a Shell in Assembly

1. Check system call interface:

```
asm linkage long sys_execve(const char __user *filename,  
                             const char __user *const __user *argv,  
                             const char __user *const __user *envp);
```

ebx

ecx

edx

- move address of string “/bin/sh0” into ebx
- move address of the address of “/bin/sh0” into ecx (how?)
- move address of null word into edx

2. Check system call number:

- move system call number (11) into eax

3. Execute the interrupt 0x80 instruction

Spawn a Shell in Assembly

- file parameter (ebx)
 - we need the null terminated string /bin/sh somewhere in memory
- argv parameter (ecx)
 - we need the address of the string /bin/sh somewhere in memory
 - followed by a NULL word
- env parameter (edx)
 - we need a NULL word somewhere in memory

/bin/sh0	addr	0000
----------	------	------

Enable Privileges

- 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
- Technique:
 - Shellcode has to enable privileges again (using setuid)
 - How?

Further Reading

- Hacking: The Art of Exploitation, 2nd Edition
 - Chapter 5
 - Available online at SFU library (using your SFU email)

Todo list

- Project ideas