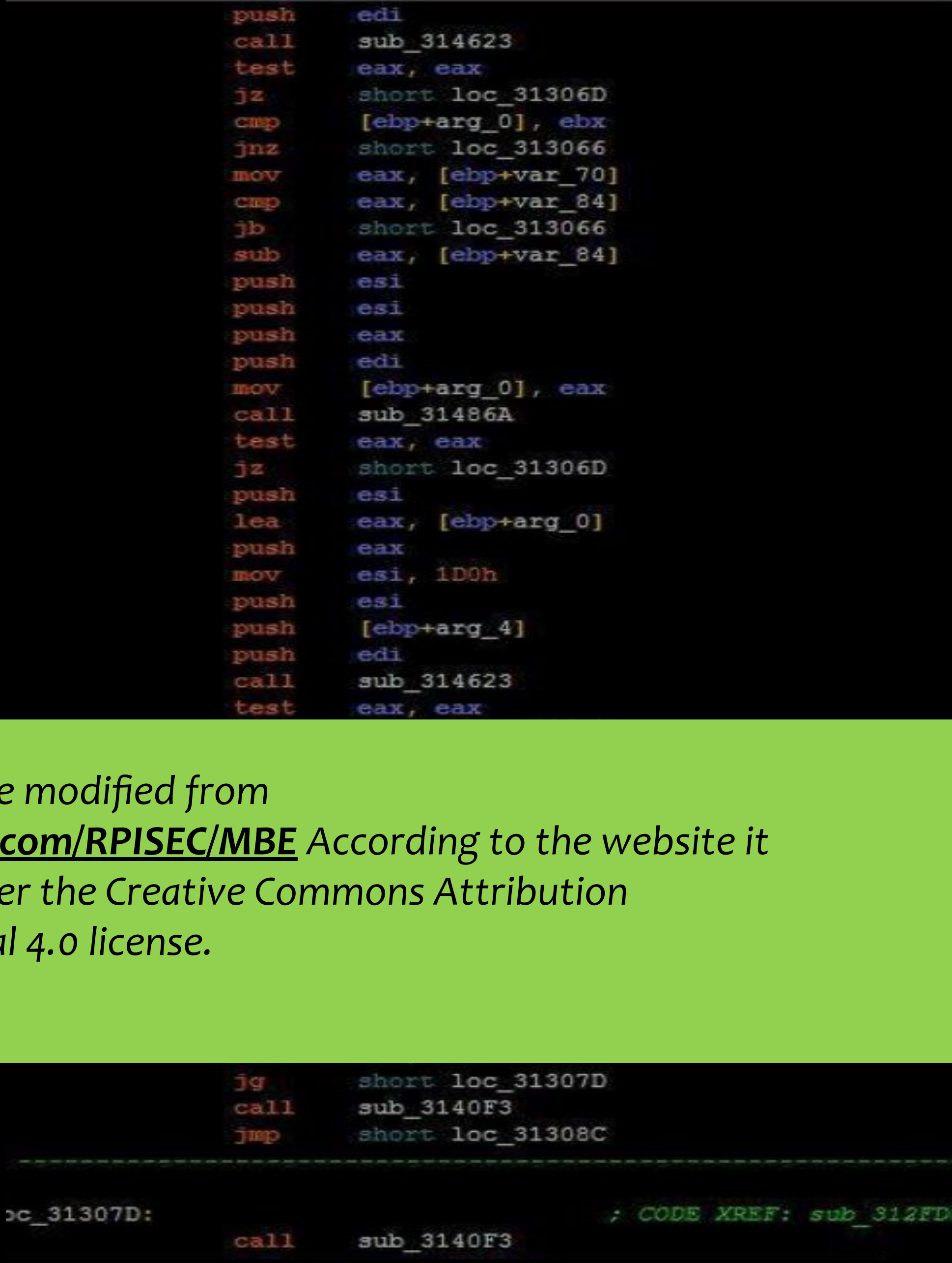
# Exploit Mitigation

Christopher Roberts (@caffix)

# Source Material

The majority of content is adapted from the Modern Binary Exploitation class taught at Rensselaer Polytechnic Institute (CSCI 4968 - Spring 2015) and released on github.




*These slides are modified from* ***[https://github.com/RPISEC/MBE](https://github.com/RPISEC/MBE)*** *According to the website it is licensed under the Creative Commons Attribution Noncommercial 4.0 license.*

# Stack Canaries

# Overview

1. How do we protect against overflows?
2. Different Types
3. Guarding the Stack
4. Ways to Leak Information

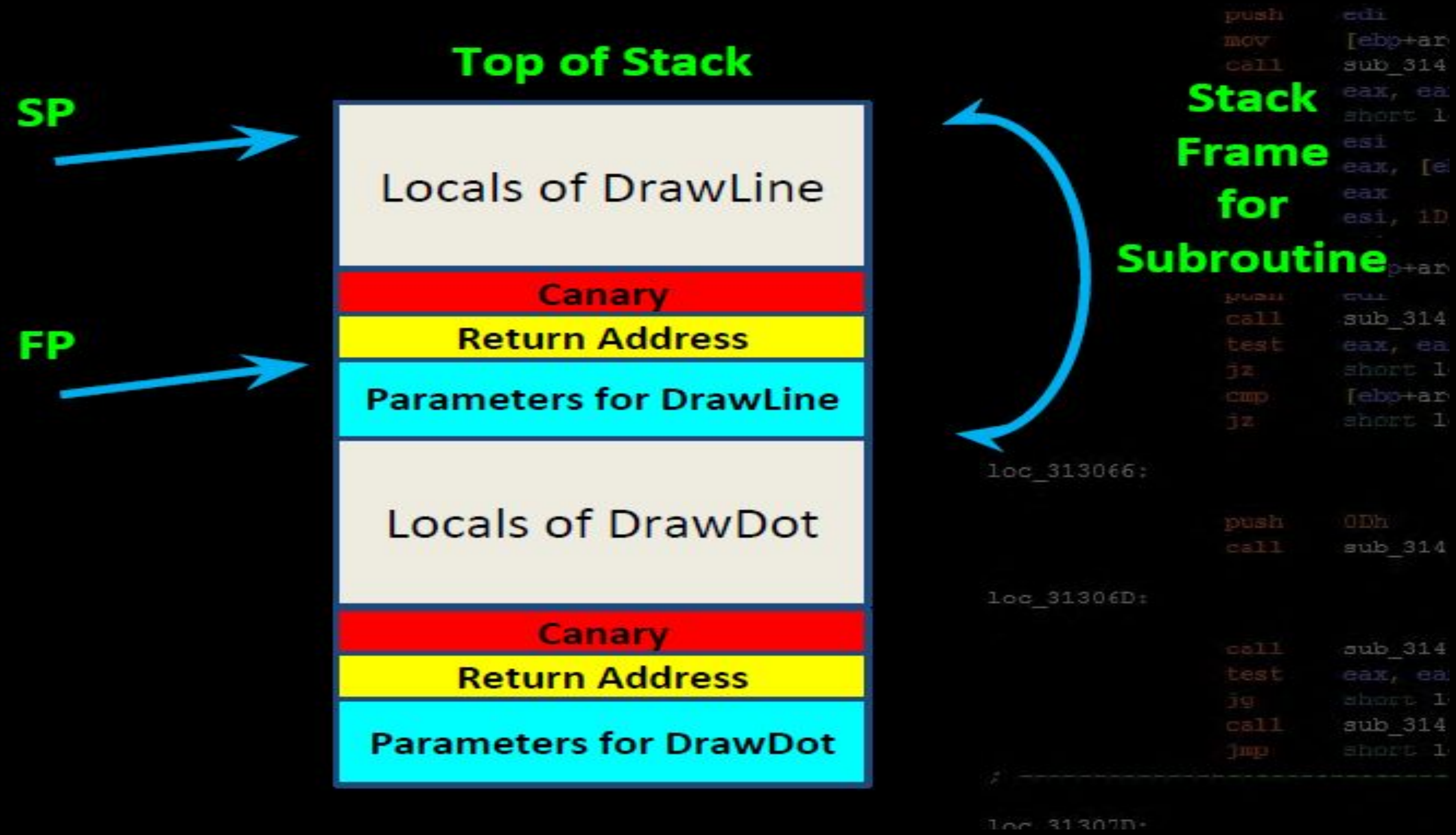
# Overflow Protections

- Before the Overflow (program and compile)
  - Program well – strcpy vs. strncpy
  - Validate input – ASCII?
  - Static / dynamic analysis tools – LLVM / SAT Solvers / valgrind
- After the Overflow (OS level)
  - Intercept function calls – Link Libsafe
  - Turn off execution – NX bit / DEP
  - Randomize the addresses – ASLR
  - Stack Canaries / Guards / Cookies – gcc / g++ / msvc

# Stack Canaries

- Detect a stack-based buffer overflow after it occurs
- Embed random "canaries" in stack frames
- Verify their integrity prior to returning from a function
- Generally terminates the program immediately if the canary is corrupted

# Stack Canaries



# Stack Canaries

- What is a canary?
  - It's a random integer
  - Pushed onto the stack after the return address and before local variables for some functions
    - Which functions would the compiler pick?
  - Popped off the stack and checked before function return
  - Compared to a saved global variable

# Stack Canary Drawbacks

- Adds overhead (instructions and cache)
- Only defends against stack overflows
- NULL canaries can potentially be abused
- Random canaries can potentially be learned
  - Format String Vulnerability
  - Other Information Leaks

# Types of Canaries

- Fixed Canaries
  - The Canary is a static random value compiled into the program
  - Targets functions that use stack buffers
  - Attackers can disassemble and overwrite with known value of the canary

# Types of Canaries

- Terminator Canaries
  - The Canary = 0 (null), newline, linefeed, EOF, -1
  - Targets string functions
  - They will stop copying at the terminator
  - Attackers cannot use string functions as the attack vector
  - Ignores rest of program security

# Types of Canaries

- Random Canaries
  - Random number chosen at program startup
    - Attacker must be dynamic
  - Inserts into (most) stack frames
  - Trigger: Return Addresses
  - Commonly used by compilers
  - Defeat with brute force, information leaks, or function pointer overwrites

# Types of Canaries

- GCC Random Canaries
  - `-fno-stack-protector`
  - `-fstack-protector` (default)
    - char array of 8 bytes or more declared on the stack
    - `--param=ssp-buffer-size=N`
  - `-fstack-protector-all`
    - same as stack-protector, but protects all functions
  - `-fstack-protector-strong`
    - declaration of type or length of local arrays
    - local var addresses or local register variables

# Ways to Defeat Canaries

- Brute Force
  - Cool example attack: http://vagmour.eu/persistence-1/
  - Requires same canary for each thread so can’t call execve()
  - overwrite canary byte by byte

# Ways to Defeat Canaries

- Learnable Random Numbers
    - GS calculate the canary 2007
    - http://uninformed.org/?v=7&a=2&t=sumry
    - Android PRNG example 2014 (IBM):
    - https://www.usenix.org/system/files/conference/woot14/woot14-kaplan.pdf
    - bad crypto for random generator
    - if /dev/random is not found, sometimes
    - pseudo-random generators are used

# Ways to Defeat Canaries

- Reading Off of the Stack
  - buffer overflow
    - overwrite null terminator
    - read past the end of array
  - format string vulnerabilities
    - http://www.exploit-monday.com/2011/06/leveragingformat-string.html
  - information leaks /memory leaks (out of scope)
    - more complicated attack
    - useful against the stack reordering done by StackGuard/ ProPolice
    - pointers dangling / writing or reading after free
    - http://phrack.org/issues/56/5.html

# DEP and ROP

# Lecture Overview

1. Introducing DEP
2. The History of DEP
3. Bypassing DEP with ROP
4. Stack Pivoting

# Class up until Now

- Reverse Engineering
- Basic memory corruption
- Shellcoding
- Format strings
- Classical exploitation, few protection, pretty eZ
- Time to add some ‘modern’ to the binary exploitation madness

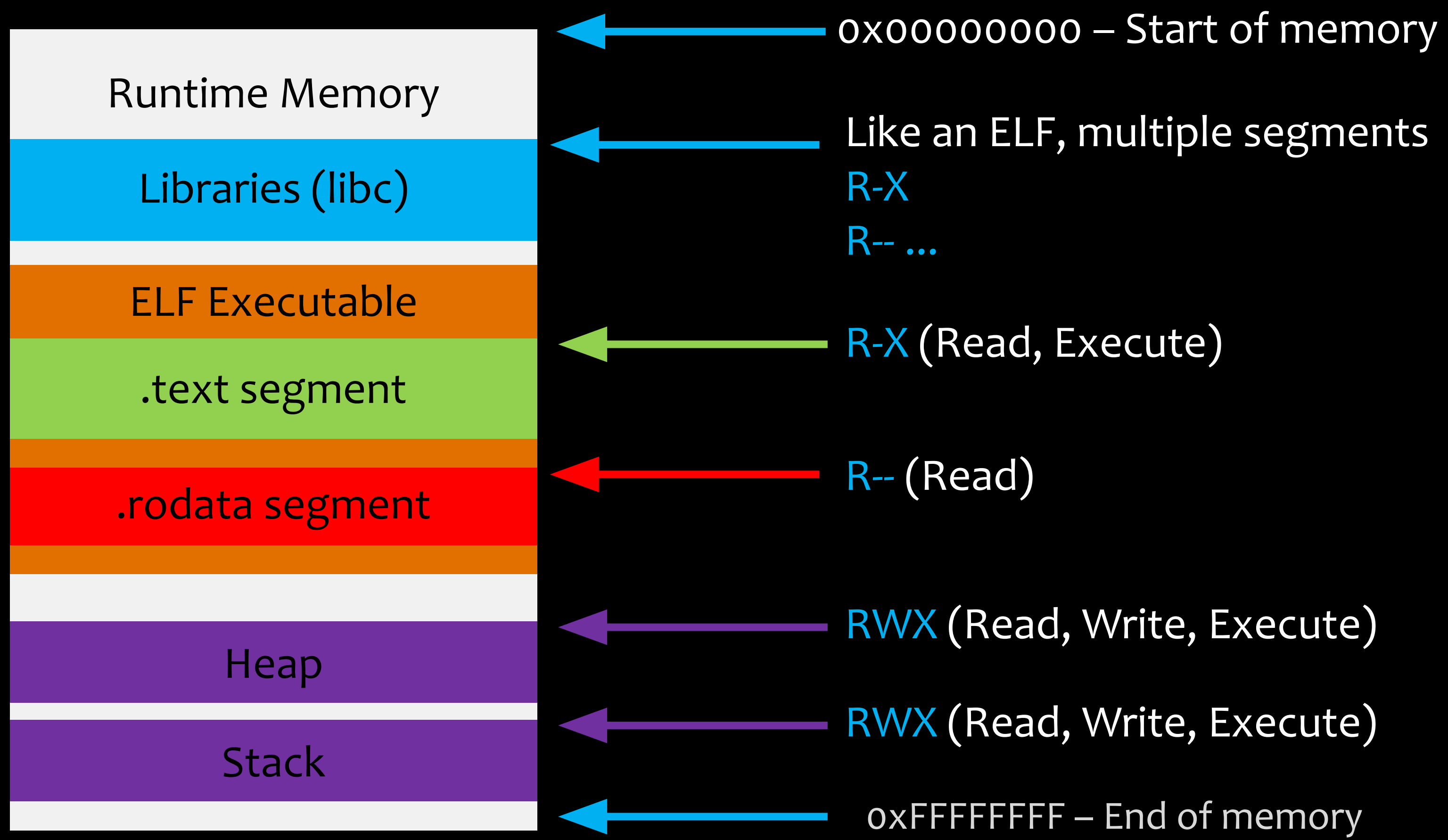
# Modern Exploit Mitigations

- Today we turn DEP back on
  - No more silly –z execstack in our gcc commands

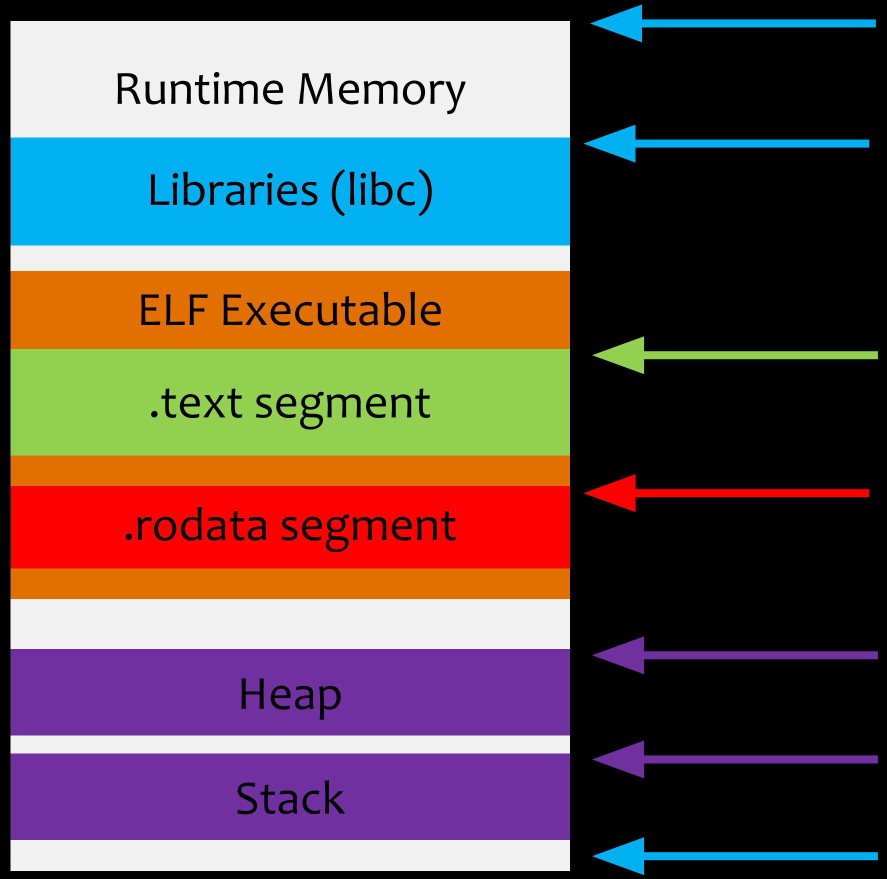
# Course Terminology

- Data Execution Prevention
  - An exploit mitigation technique used to ensure that only code segments are ever marked as executable
  - Meant to mitigate code injection / shellcode payloads
  - Also known as DEP, NX, XN, XD, W^X

# Runtime Process Without DEP



# Runtime Process Without DEP




0x00000000 – Start of memory

Like an ELF, multiple segments
R-X
R-- ...

R-X (Read, Execute)

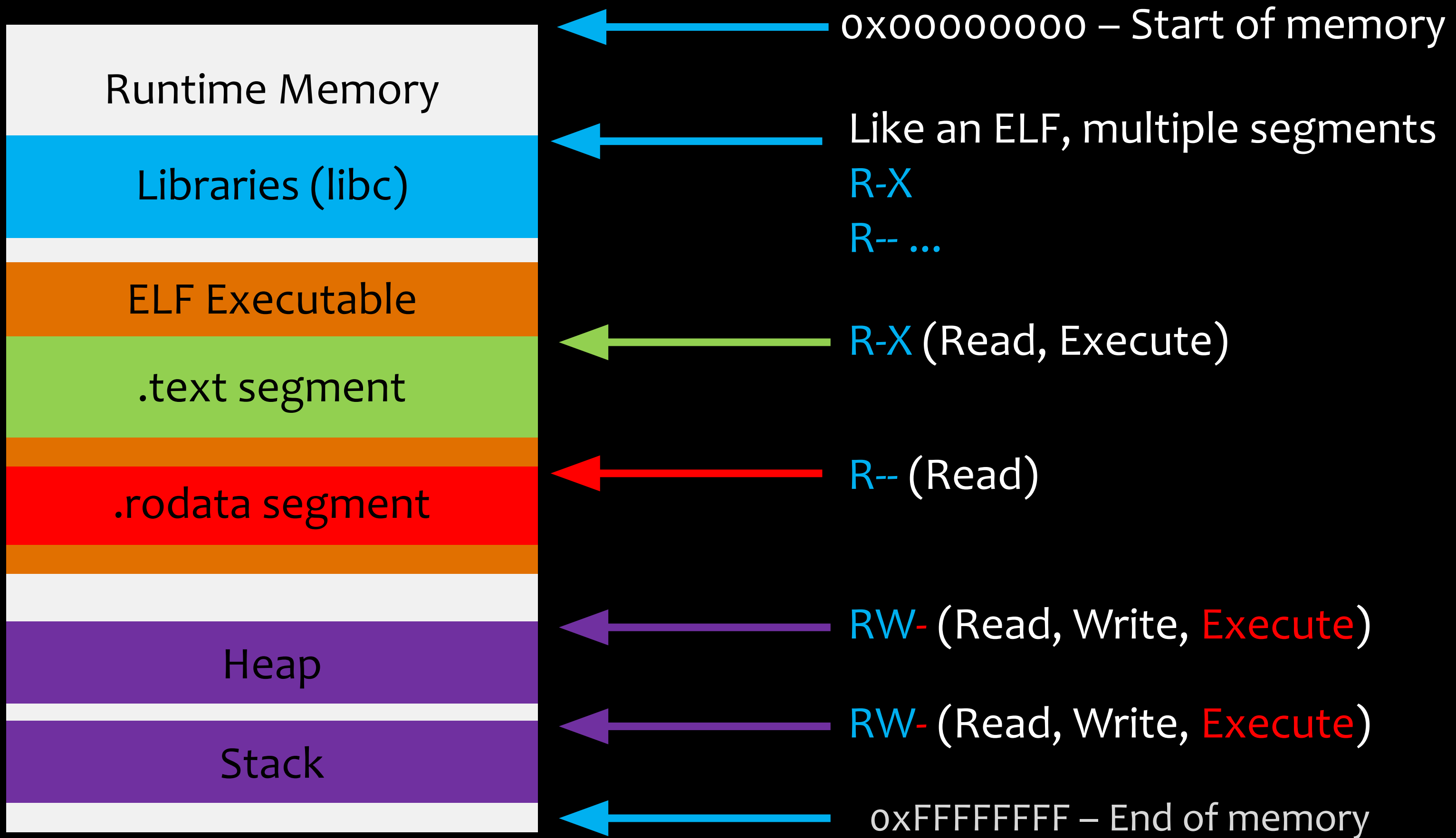
R-- (Read)

RWX (Read, Write, Execute)

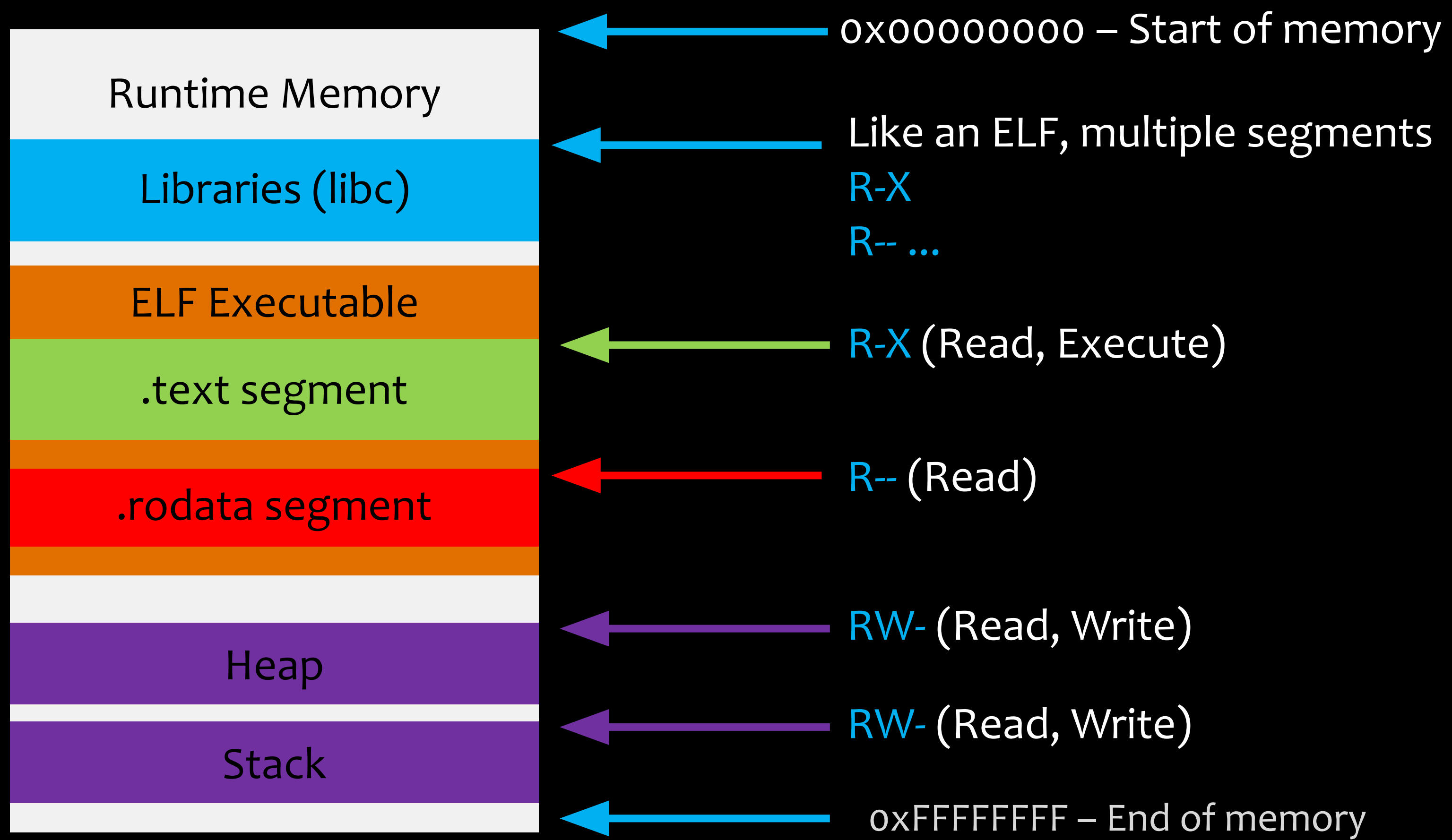
RWX (Read, Write, Execute)

0xFFFFFFFF – End of memory

# Runtime Process Without DEP



# Runtime Process With DEP

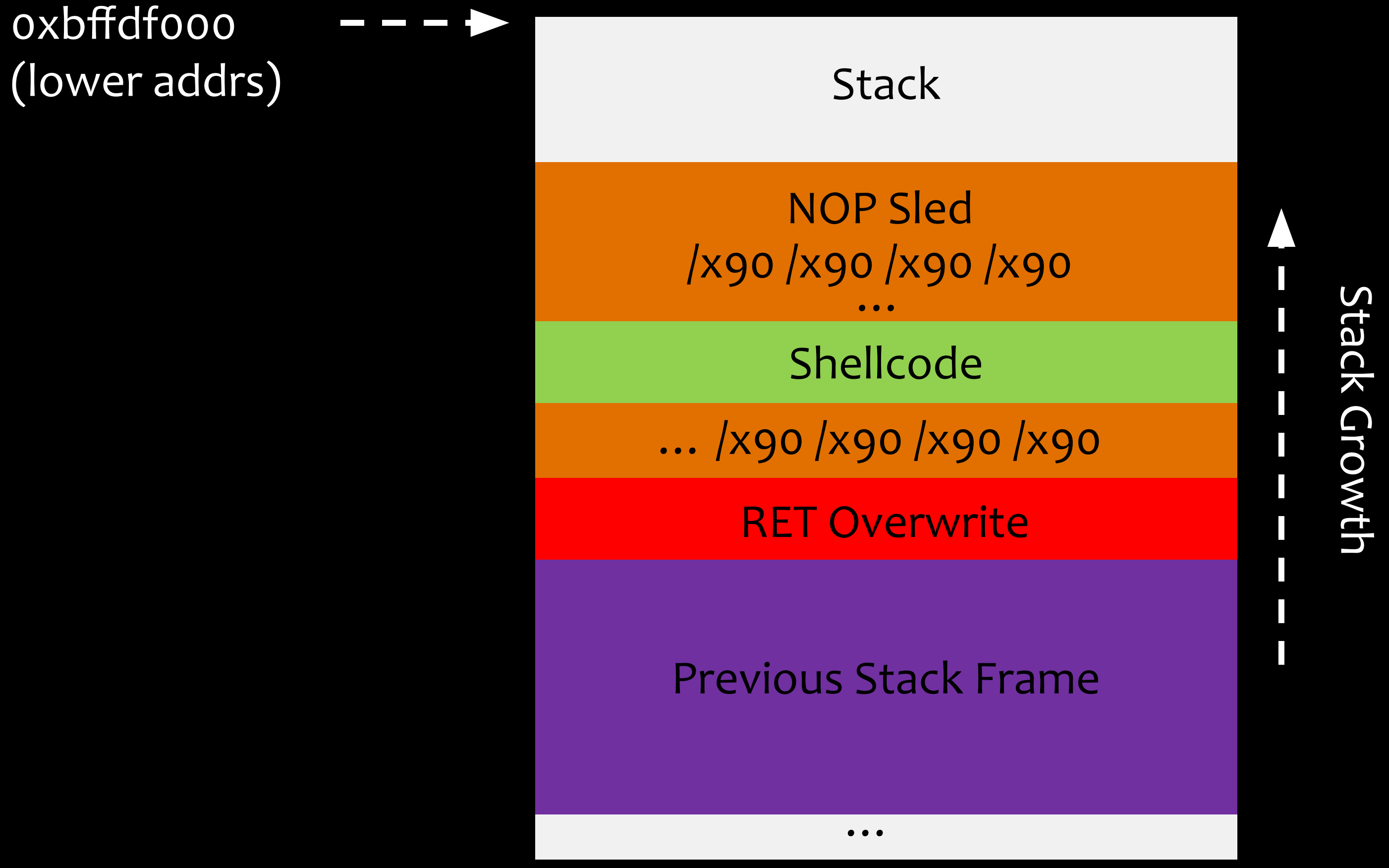


# DEP Basics

- No segment of memory should ever be Writable and Executable at the same time, ‘W^X’
- Common data segments
  - Stack, Heap
  - .bss
  - .ro
  - .data
- Common code segments
  - .text
  - .plt

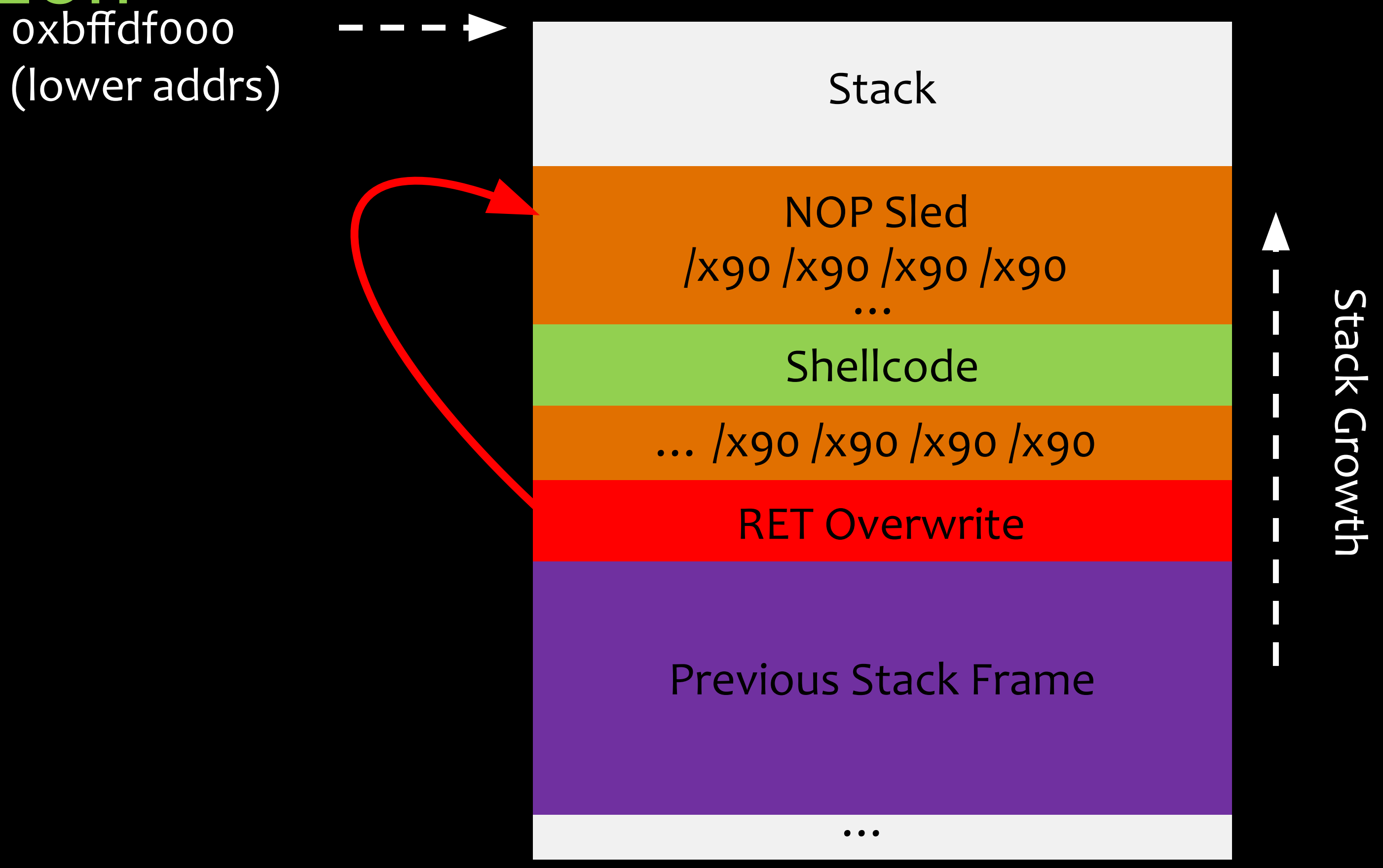
# DEP in Action

- Data should never be executable, only code
- What happens if we stack smash, inject shellcode, and try to jump onto the stack?



# DEP in Action

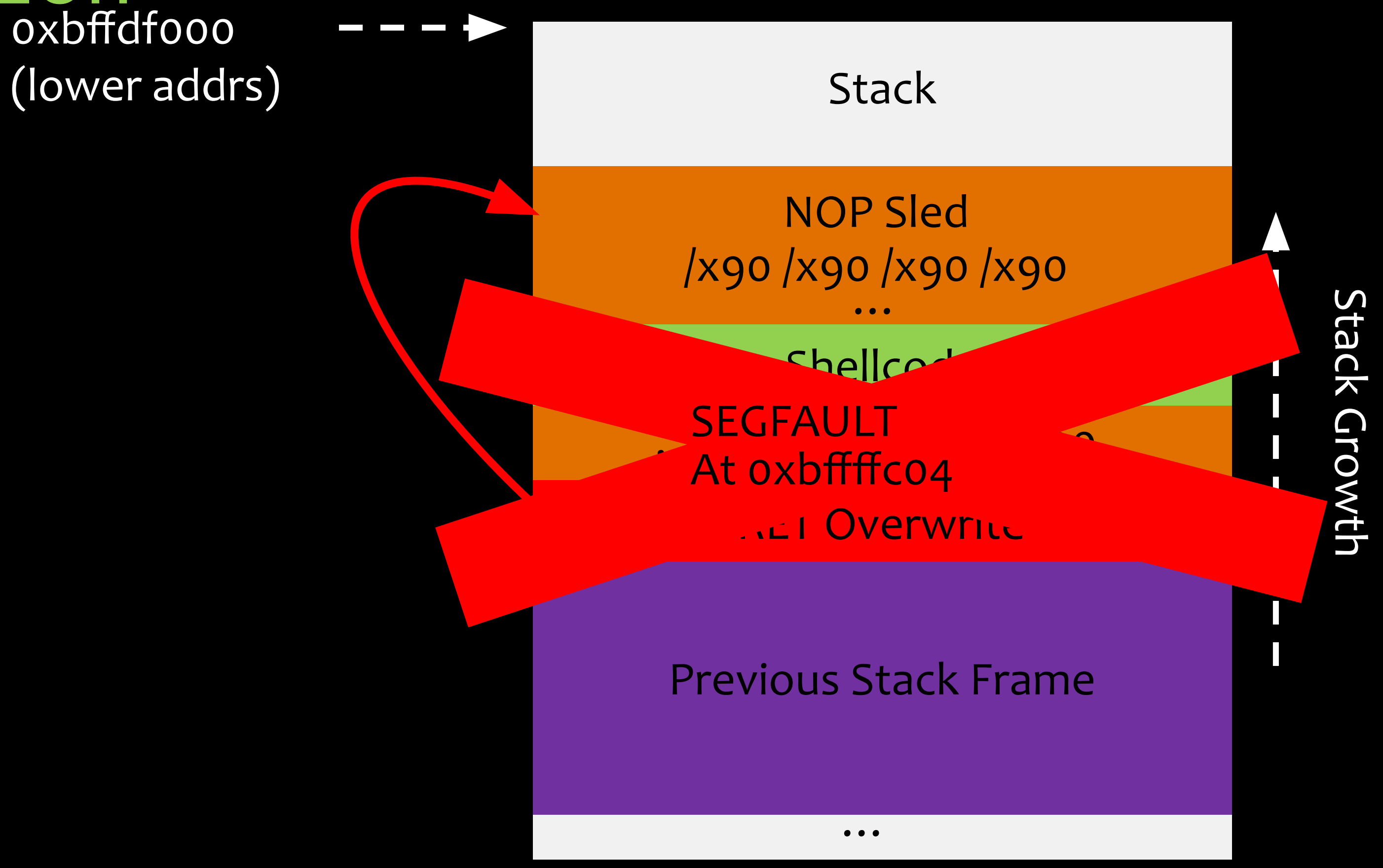
- Data should never be executable, only code
- What happens if we stack smash, inject shellcode, and try to jump onto the stack?



# DEP in Action

- Data should never be executable, only code
- What happens if we stack smash, inject shellcode, and try to jump onto the stack?

yay mitigation technologies!



# Lecture Overview

1. Introducing DEP
2. The History of DEP
3. Bypassing DEP with ROP
4. Stack Pivoting

# History of DEP

- When was DEP implemented?

# History of DEP

- When was DEP implemented?
  - August 14th, 2004 – Linux Kernel 2.6.8

# History of DEP

- When was DEP implemented?
  - August $14^{th}$, 2004 – Linux Kernel 2.6.8
  - August $25^{th}$, 2004 – Windows XP SP2

# History of DEP

- When was DEP implemented?
  - August 14th, 2004 – Linux Kernel 2.6.8
  - August 25th, 2004 – Windows XP SP2
  - June 26th, 2006 – Mac OSX 10.5

# History of DEP

- When was DEP implemented?
  - August 14th, 2004 – Linux Kernel 2.6.8
  - August 25th, 2004 – Windows XP SP2
  - June 26th, 2006 – Mac OSX 10.5

about 10 years ago

# 2004 in Perspective

- Facebook is created
- G-Mail launches as beta
- Ken Jennings begins his 74 win streak on Jeopardy
- Halo 2 is released, as is Half Life 2
- LOST airs its first episode


# Security is Young

- Technologies in modern exploit mitigations are incredibly young, and the field of computer security is rapidly evolving
- DEP is one of the main mitigation technologies you must bypass in modern exploitation

# Lecture Overview

1. Introducing DEP
2. The History of DEP
3. Bypassing DEP with ROP
4. Stack Pivoting

# Bypassing DEP

- DEP stops an attacker from easily executing injected shellcode assuming they gain control of EIP
  - shellcode almost always ends up in a RW – region

- If you can't inject (shell) code to do your bidding, you must re-use the existing code!
  - This technique is usually some form of ROP

# Course Terminology

- Return Oriented Programming
  - A technique in exploitation to reuse existing code gadgets in a target binary as a method to bypass DEP
  - Also known as ROP
- Gadget
  - A sequence of meaningful instructions typically followed by a return instruction
  - Usually multiple gadgets are chained together to compute malicious actions like shellcode does
  - These chains are called ROP Chains

# Relevant Quotes

“Preventing the introduction of malicious code is not enough to prevent the execution of malicious computations”

- Dino Dai Zovi

# Gadgets

- ROP Chains are made up of gadgets
- Example gadgets –

```
xor     eax, eax
ret

pop     ebx
pop     eax
ret

add eax, ebx
ret
```

# $ ropgadget - - binary / bin / bash

```
0x080d2262 : xor ebx, ebx ; mov esi, edi ; jmp 0x80d227d
0x080ac337 : xor ecx, dword ptr [ecx + 0x448b2404] ; and al, 0xc ; call eax
0x080d02b8 : xor ecx, ecx ; cmp dword ptr [edx], 0x2e ; je 0x80d02f1 ; mov eax, ecx ; ret
0x080cc175 : xor ecx, ecx ; mov eax, edx ; pop ebx ; mov edx, ecx ; pop esi ; pop edi ; ret
0x0808b728 : xor ecx, ecx ; xor edx, edx ; mov eax, esi ; call 0x8087958
0x080bc610 : xor edi, edi ; pop ebx ; mov eax, edi ; pop esi ; pop edi ; pop ebp ; ret
0x0812b059 : xor edi, edx ; jmp dword ptr [ebx]
0x0811a06d : xor edx, edi ; jmp dword ptr [eax]
0x080fcc4d : xor edx, edx ; add esp, 0x14 ; pop esi ; pop edi ; pop ebp ; ret
0x080fcb6c : xor edx, edx ; add esp, 0xc ; pop esi ; pop edi ; pop ebp ; ret
0x080a395b : xor edx, edx ; call 0x80a2879
0x080d6e71 : xor edx, edx ; cmp eax, 0x16 ; setne dl ; jmp 0x80d6e53
0x08072090 : xor edx, edx ; mov dword ptr [eax + 8], edx ; add esp, 0x18 ; pop ebx ; ret
0x0808b72a : xor edx, edx ; mov eax, esi ; call 0x8087956
0x080861bd : xor edx, edx ; pop ebx ; pop esi ; ret
0x08070246 : xor edx, edx ; pop esi ; pop edi ; pop ebp ; ret
0x08075a58 : xor edx, edx ; pop esi ; pop edi ; ret
0x080f8877 : xor esi, 0x89c085ff ; ret
0x080f3a88 : xrelease ; mov dword ptr [esp], esi ; call 0x80efd46

Unique gadgets found: 15840
lecture@warzone:/levels$
```

# Understanding ROP

- It is almost always possible to create a logically equivalent ROP chain for a given piece of shellcode

exit(0) - shellcode

```
xor    eax, eax
xor    ebx, ebx
inc    eax
int    0x80
```

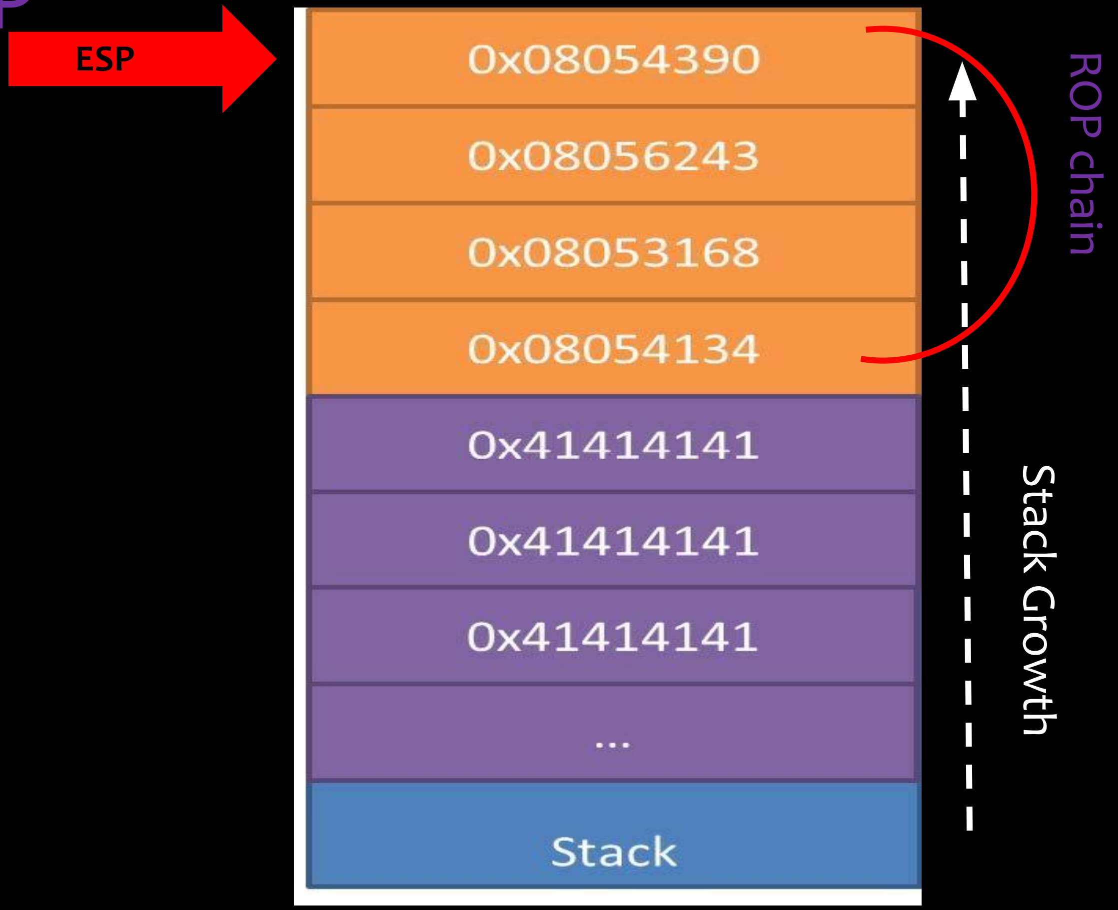
exit(0) - ROP chain

```
xor   eax, eax
ret
------------------------
xor   ebx, ebx
ret
------------------------
inc   eax
ret
------------------------
int   0x80
```

# Understanding ROP

exit(0) - ROP chain

```
xor   eax, eax
ret
-------------------
xor   ebx, ebx
ret
-------------------
inc   eax
ret
-------------------
int   0x80
```

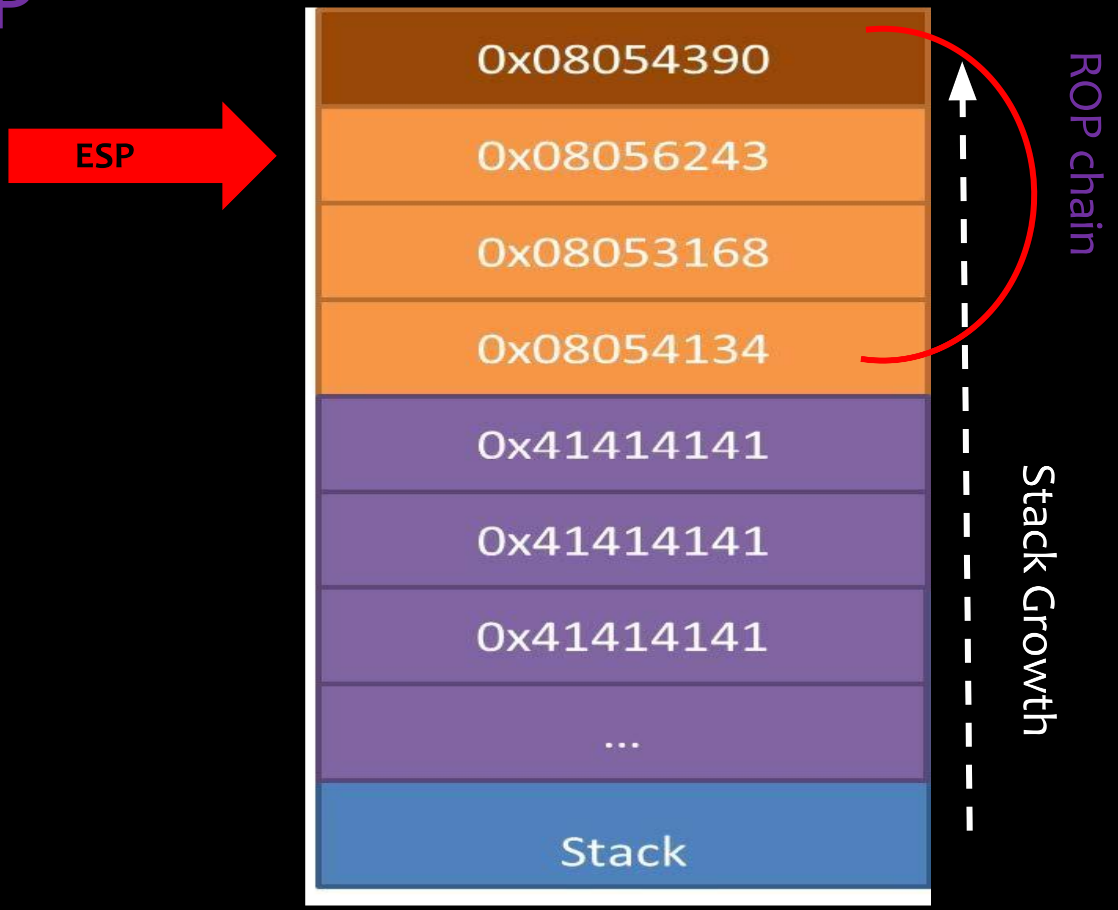


# Understanding ROP

exit(0) - ROP chain

EIP

```
xor   eax, eax
ret
-----------------------------
xor   ebx, ebx
ret
-----------------------------
inc   eax
ret
-----------------------------
int   0x80
```



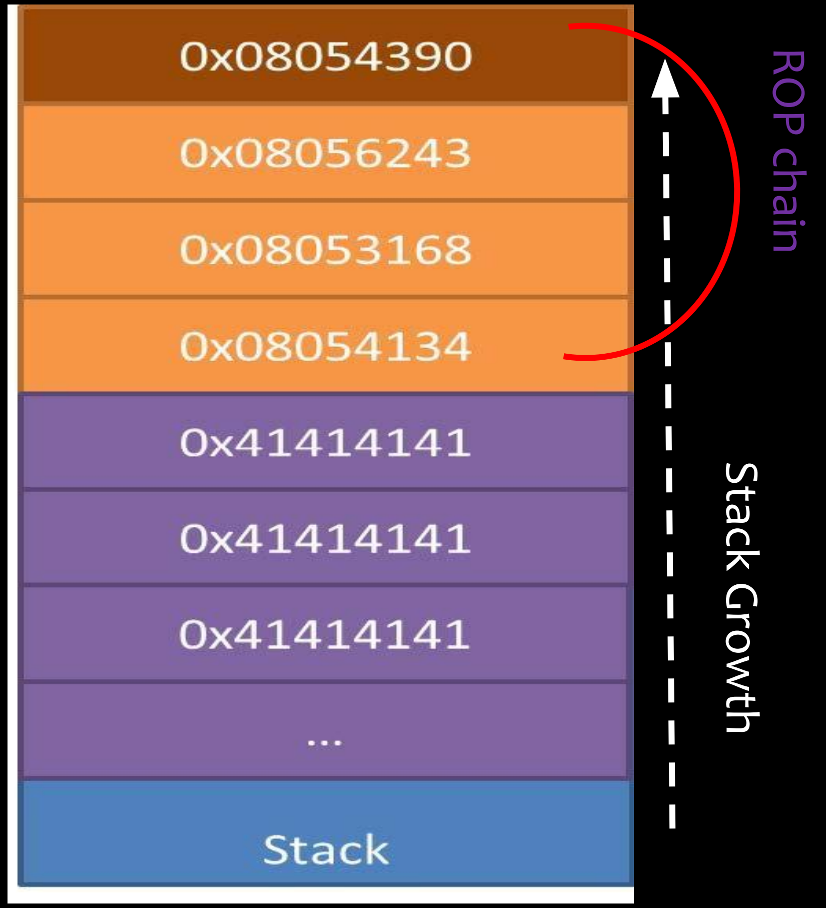
# Understanding ROP

exit(0) - ROP chain

```
xor   eax, eax
ret
-------------------
xor   ebx, ebx
ret
-------------------
inc   eax
ret
-------------------
int   0x80
```

EIP

ESP

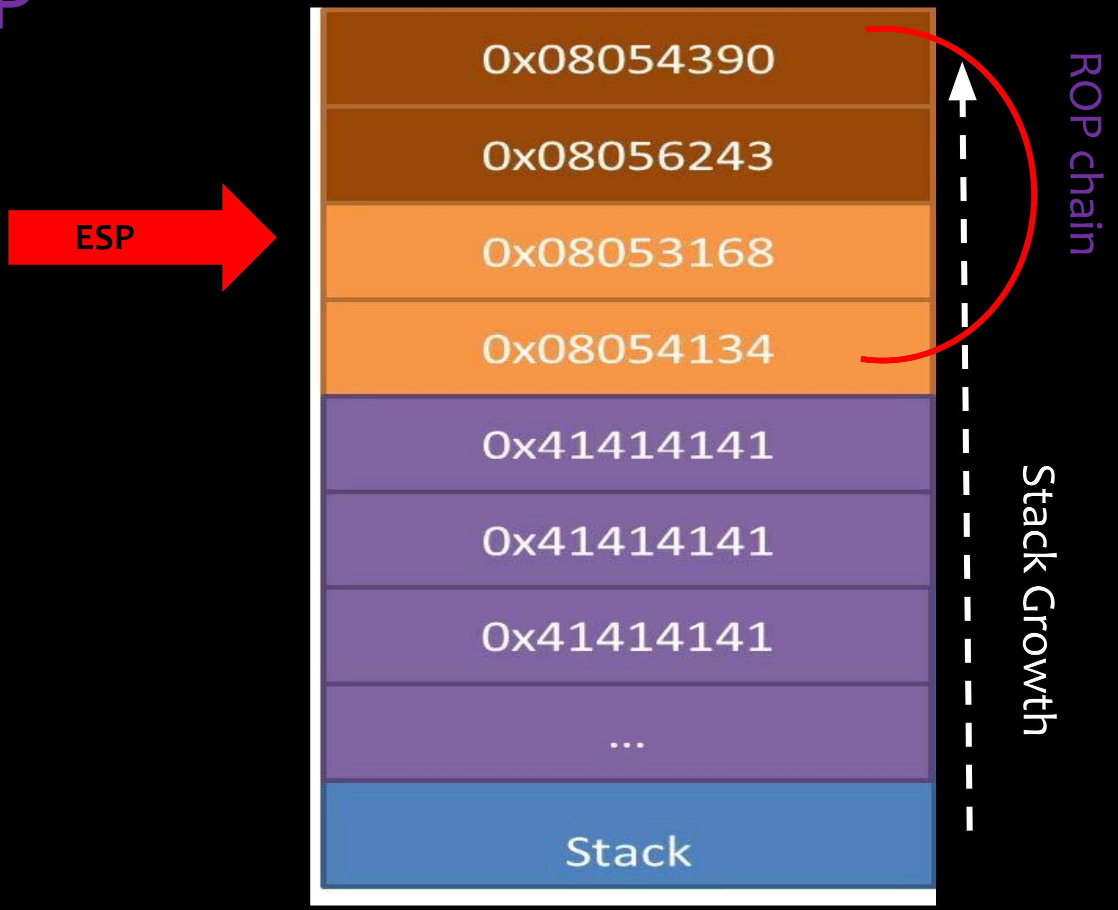


# Understanding ROP

exit(0) - ROP chain

```
xor   eax, eax
ret
-----------------------
xor   ebx, ebx
ret
-----------------------
inc   eax
ret
-----------------------
int   0x80
```

EIP



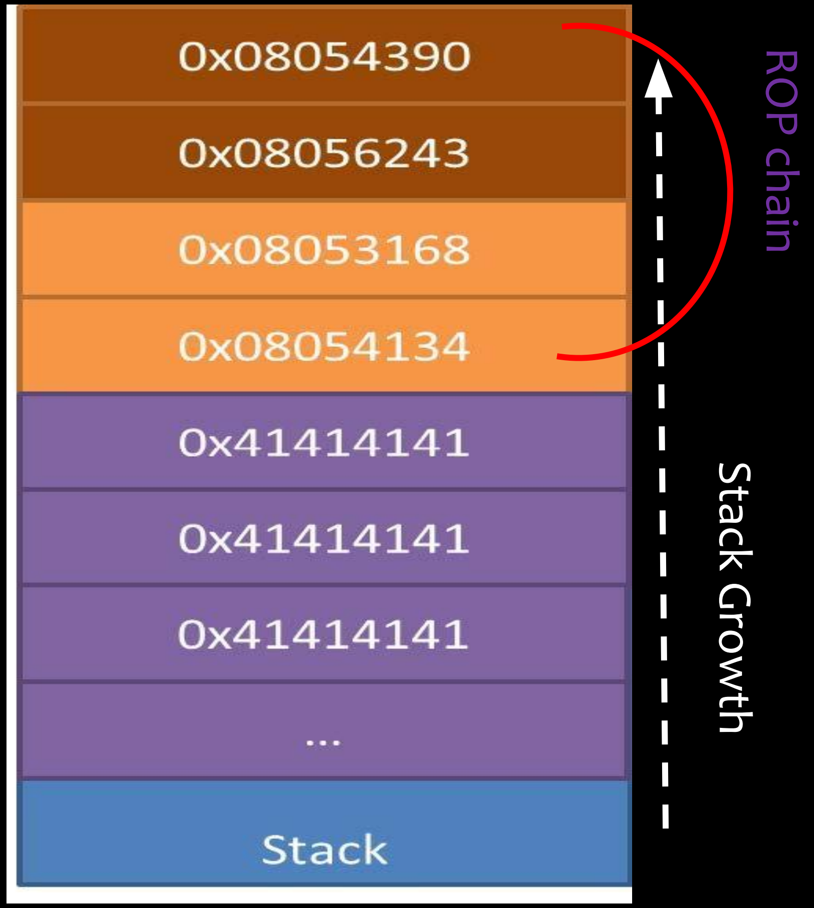
# Understanding ROP

exit(0) - ROP chain

```
xor   eax, eax
ret
-----------------------
xor   ebx, ebx
ret
-----------------------
inc   eax
ret
-----------------------
int   0x80
```

EIP

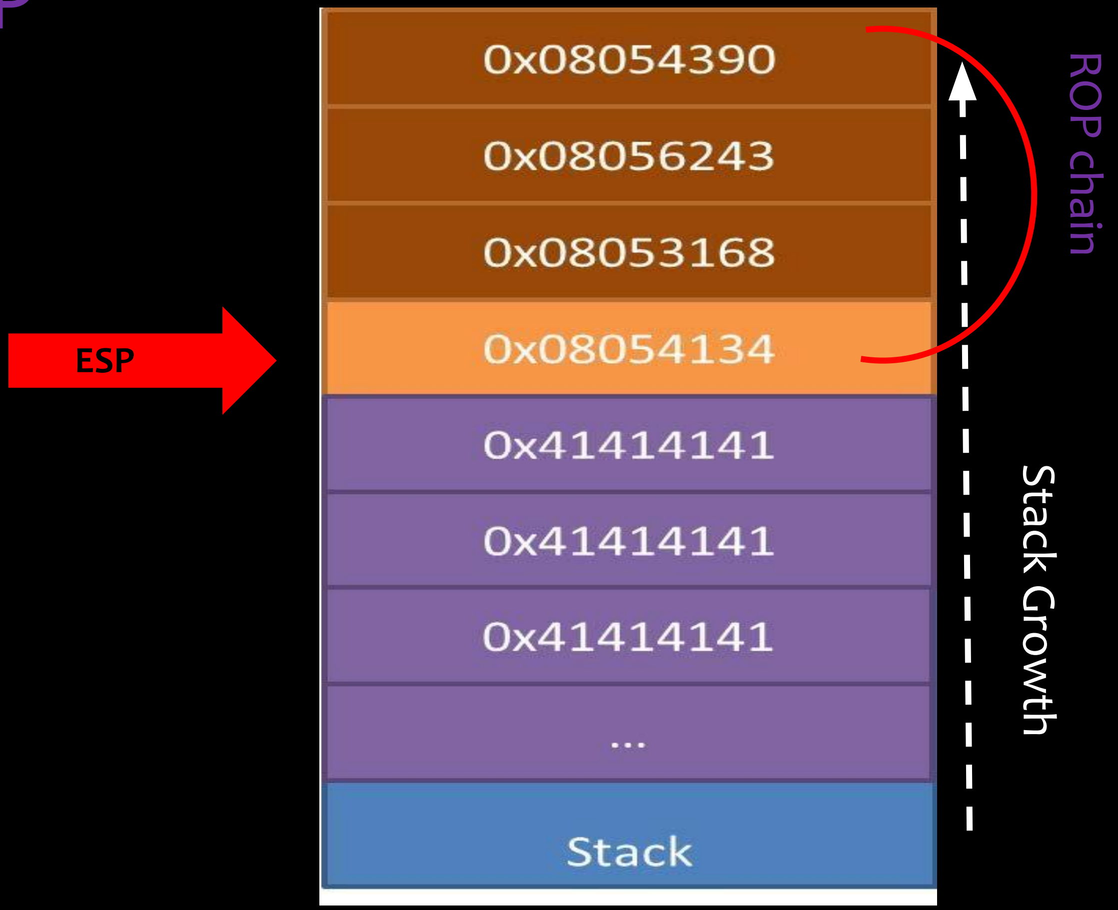
ESP



# Understanding ROP

exit(0) - ROP chain

```
xor   eax, eax
ret
------------------------------
xor   ebx, ebx
ret
------------------------------
inc   eax
ret
------------------------------
int   0x80
```

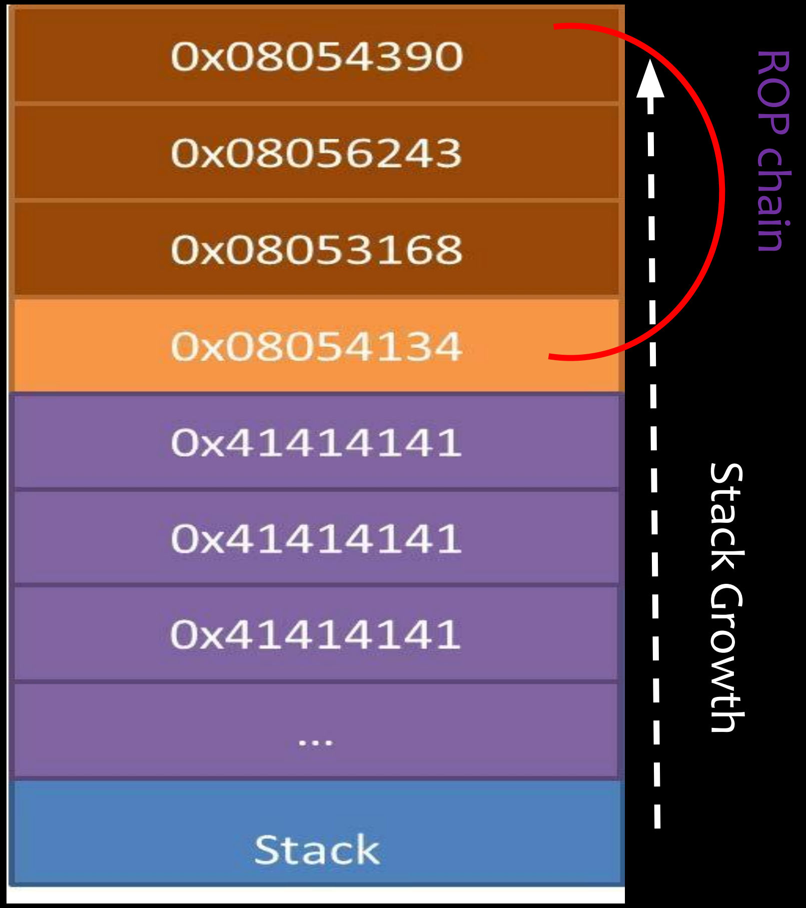
# Understanding ROP

exit(0) - ROP chain

```
xor   eax, eax
ret
-------------------
xor   ebx, ebx
ret
-------------------
inc   eax
ret
-------------------
int   0x80
```

EIP

ESP



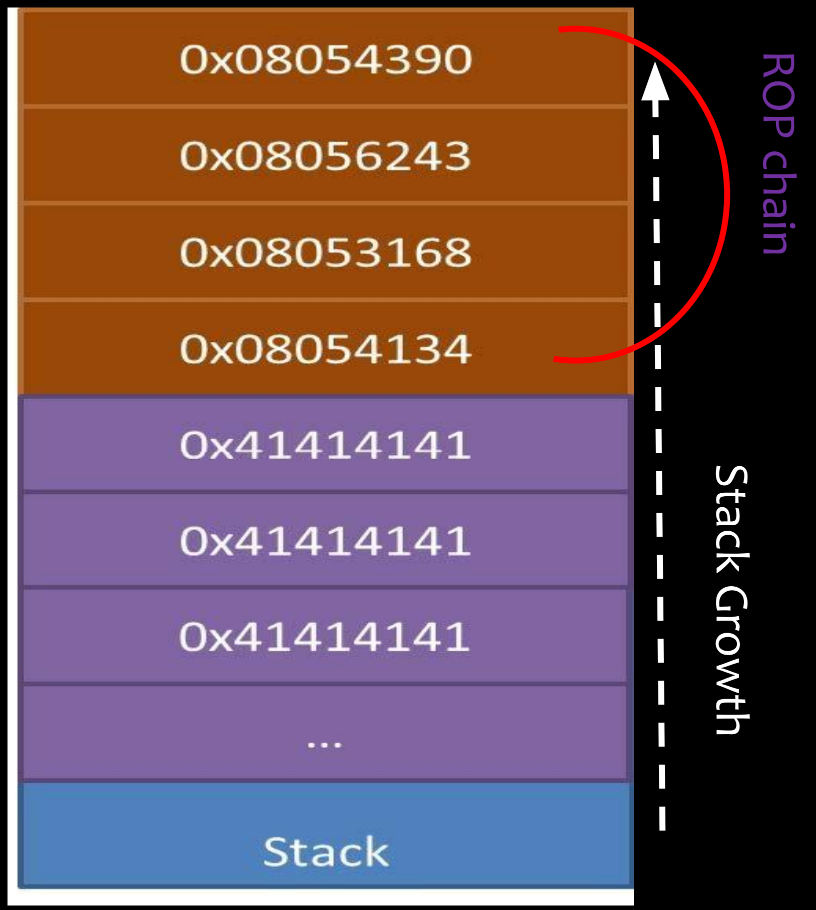
# Understanding ROP

exit(0) - ROP chain

```
xor   eax, eax
ret
--------------------
xor   ebx, ebx
ret
--------------------
inc   eax
ret
--------------------
int   0x80
```

EIP

ESP



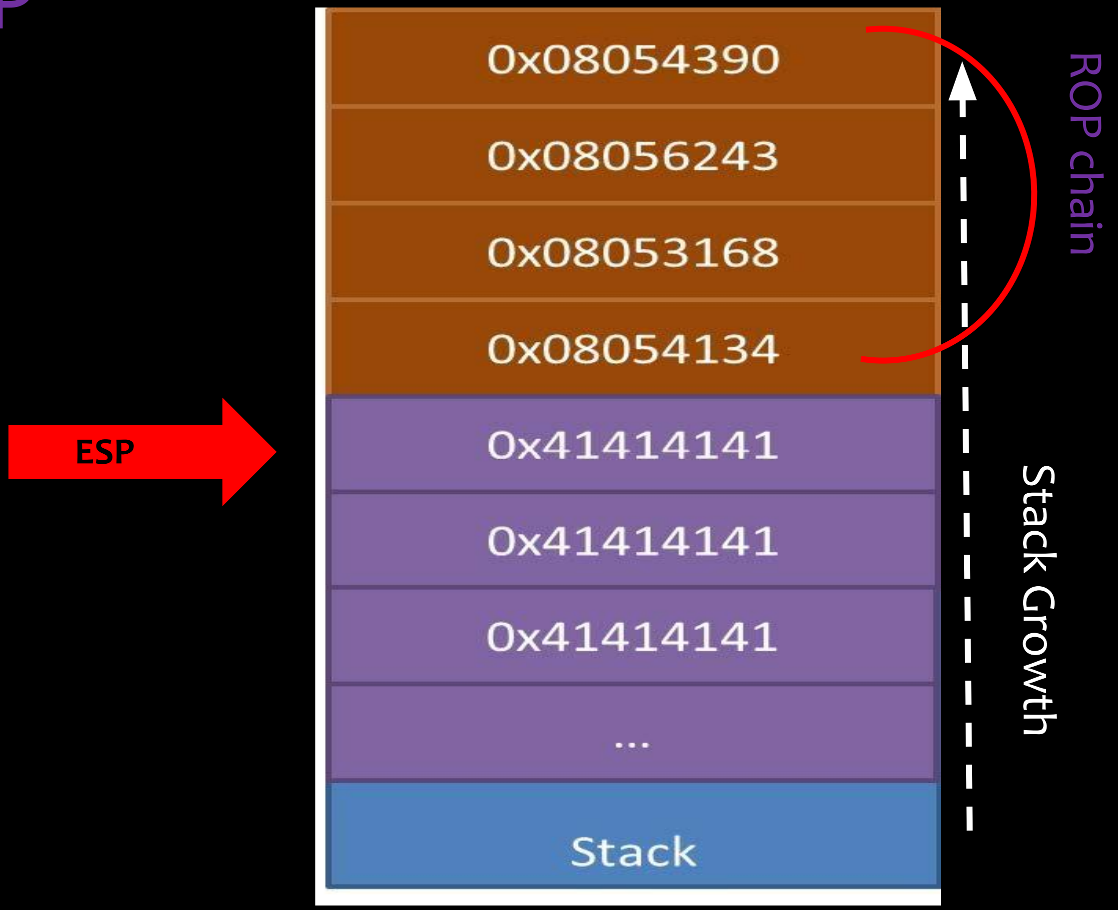
# Understanding ROP

exit(0) - ROP chain

```
xor   eax, eax
ret
-------------------------
xor   ebx, ebx
ret
-------------------------
inc   eax
ret
-------------------------
int   0x80
```

EIP

exits ...



# Bypassing DEP with ROP

- We called exit (0) without using any sort of shellcode!
- With that said, writing ROP can be difficult and you will usually have to get creative with what gadgets you find

# Relevant Tips/Tools/Commands

- $ ropgadget --binary ./rop_exit > /tmp/gadgetzXYZ.txt
  - $ cat /tmp/gadgetzXYZ.txt | grep "pop eax" | grep …
- $ asm
  - easy way to get the bytes for gadgets you're looking for
- $ gdbpeda
  - searchmem, find raw bytes in an executing program
  - ropsearch, a crappy rop gadget finder
- python

```
def q(addr):
    return struct.pack("I", addr)
```

# Lecture Overview

1. Introducing DEP
2. The History of DEP
3. Bypassing DEP with ROP
4. Stack Pivoting

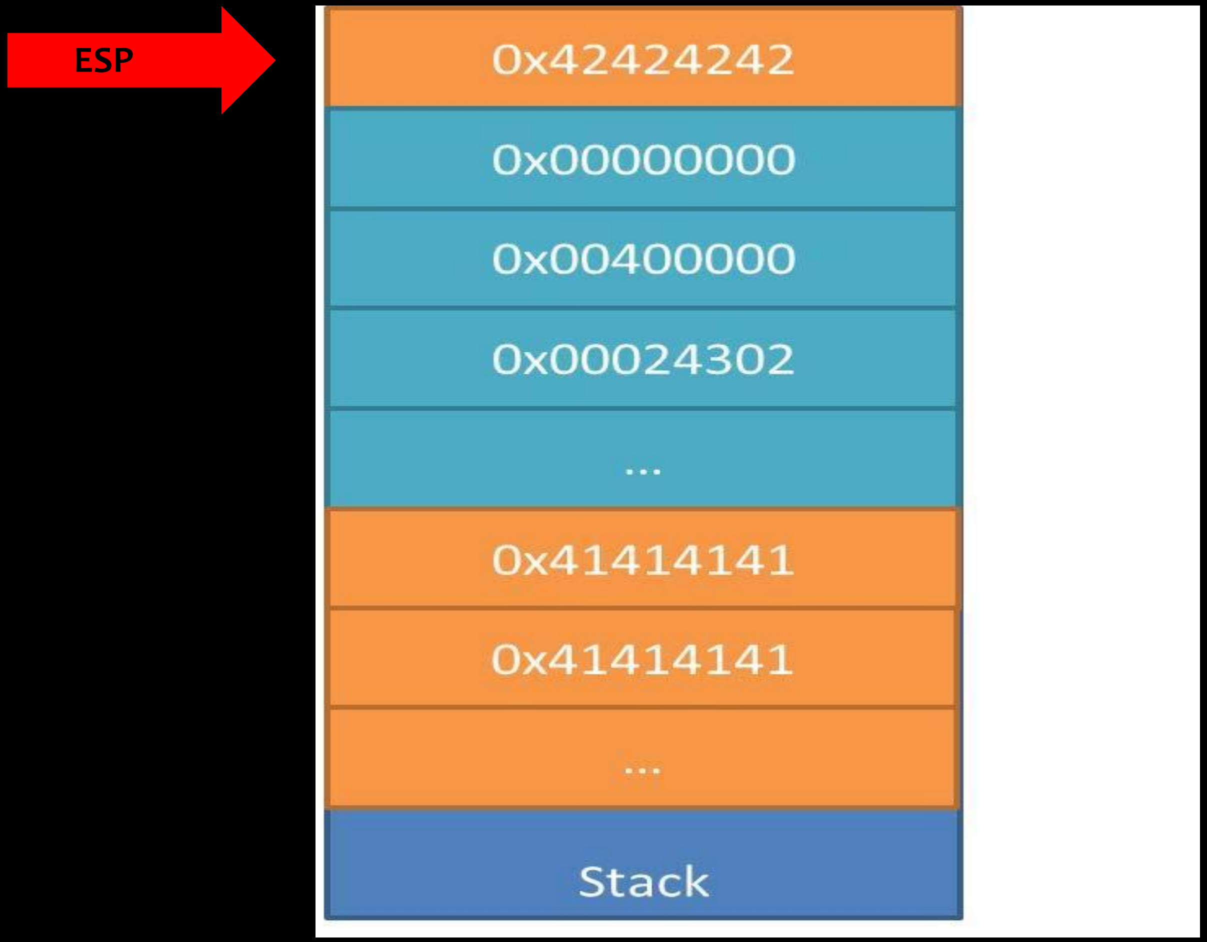
# Typical Constraints in ROP

- Typically in modern exploitation you might only get one targeted overwrite rather than a straight stack smash
- What can you do when you only have one gadget worth of execution?
  - Answer:  Stack Pivoting

# Stack Pivoting

You control the orange

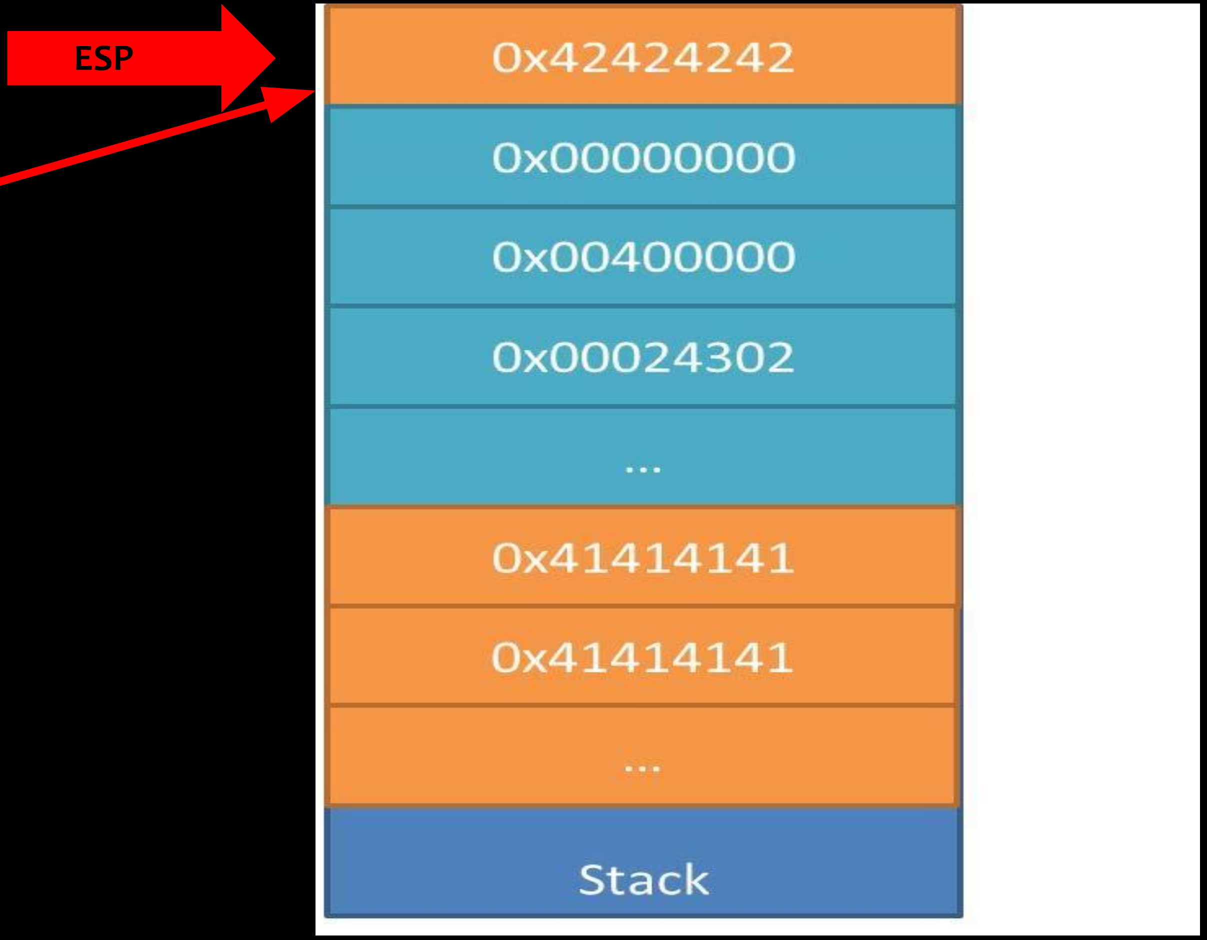
You have one gadget
before you drop into
arbitrary data on the stack



# Stack Pivoting

You control the orange

You have one gadget before you drop into arbitrary data on the stack



# Stack Pivoting

You control the orange

You have one gadget before you drop into arbitrary data on the stack

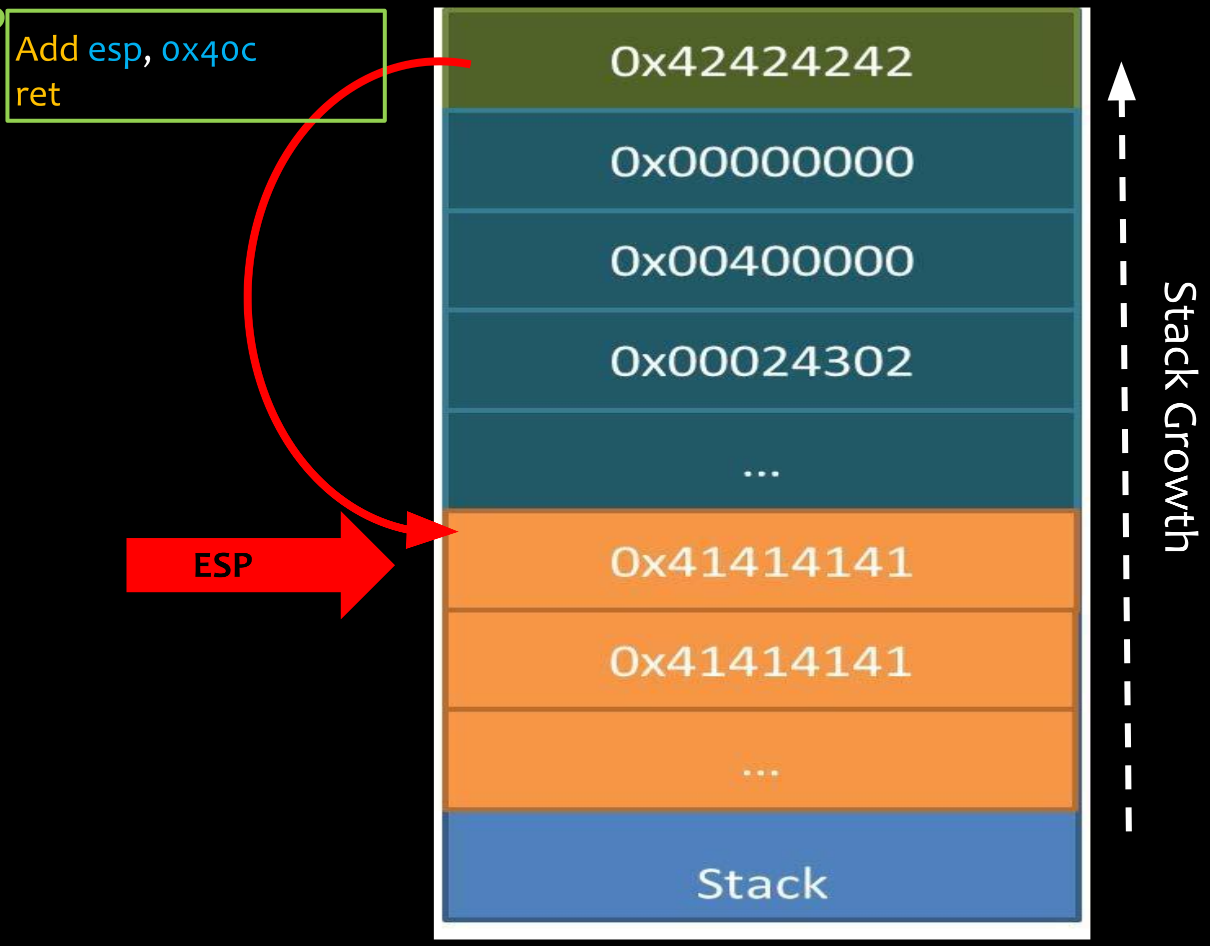
Use your one gadget to move ESP Into a more favorable location (Stack Pivot)

# Stack Pivoting

You control the orange

You have one gadget before you drop into arbitrary data on the stack

Use your one gadget to move ESP Into a more favorable location (Stack Pivot)

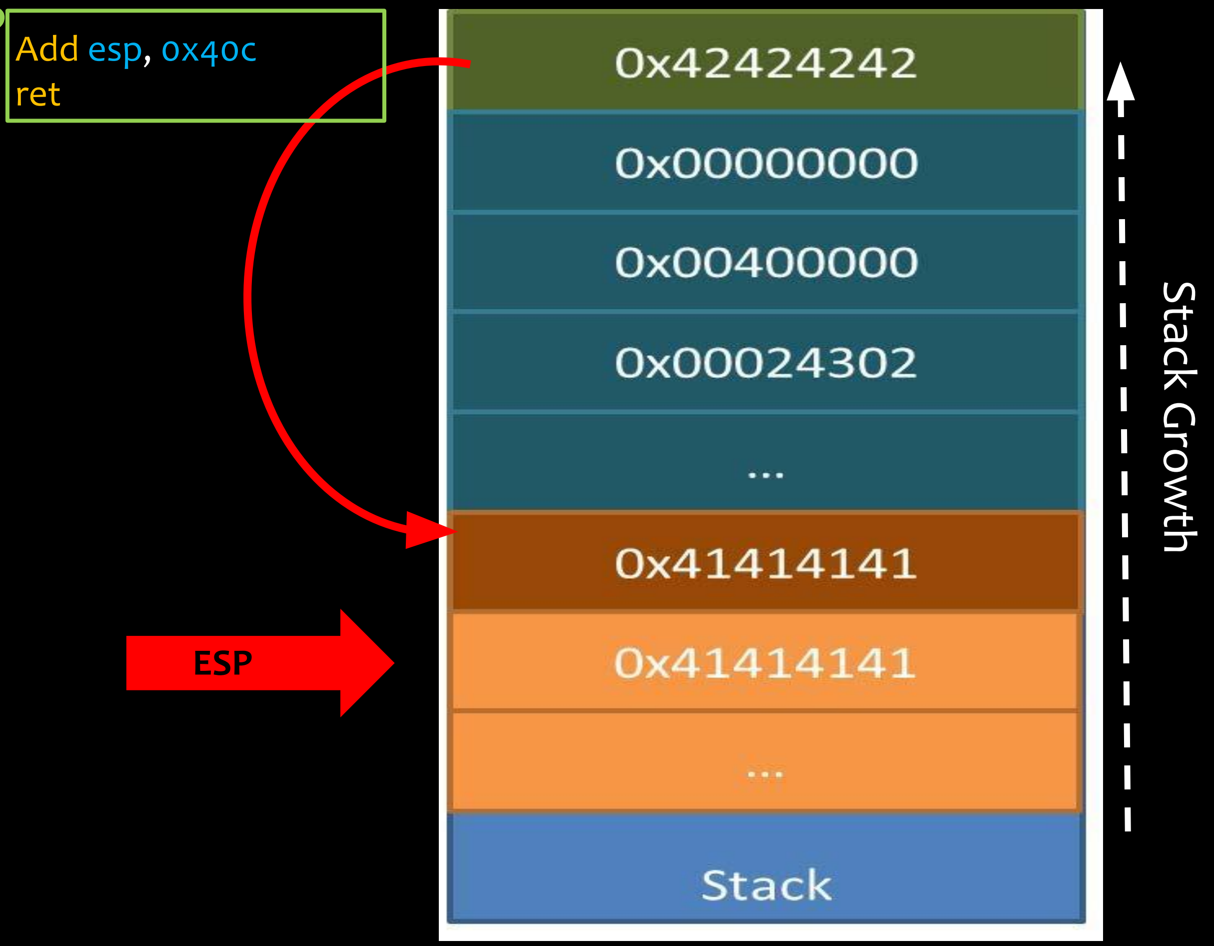


# Stack Pivoting

You control the orange

You have one gadget before you drop into arbitrary data on the stack

Use your one gadget to move ESP Into a more favorable location (Stack Pivot)



# Stack Pivoting Tips

```
add   esp, 0xXXXX
ret
```

---

```
sub   esp, 0xXXXX
ret
```

---

```
ret 0xXXXX
```

---

```
Leave ; (mov esp, ebp)
ret
```

---

```
xchg eXX, esp
ret
```

any gadgets that touch esp
will probably be of interest
for a pivot scenario

# Stack Pivoting Tips

- You may not find an exact pivot, or you may need to pivot multiple times!

- You can always pad your ROP Chains with ROP NOPs which are simply gadgets that point to ret's

# ret2libc

- ‘ret2libc’ is a technique of ROP where you return to functions in standard libraries (libc), rather than
  using gadgets

- If you know the addresses of the functions you want to ROP through in libc (assuming libc exists), ret2libc is easier than making a ROP chain with gadgets

# Common ret2libc Targets

- system()
  - Executes something on the command line
  - system (“cat flag.text”);

- (f) open() / read() / write()
  - Open/Read/Write a file contents

# ret2libc example

0x08045430: ret

EIP

- - - - - - - - - - - - - - - - - - - - - -

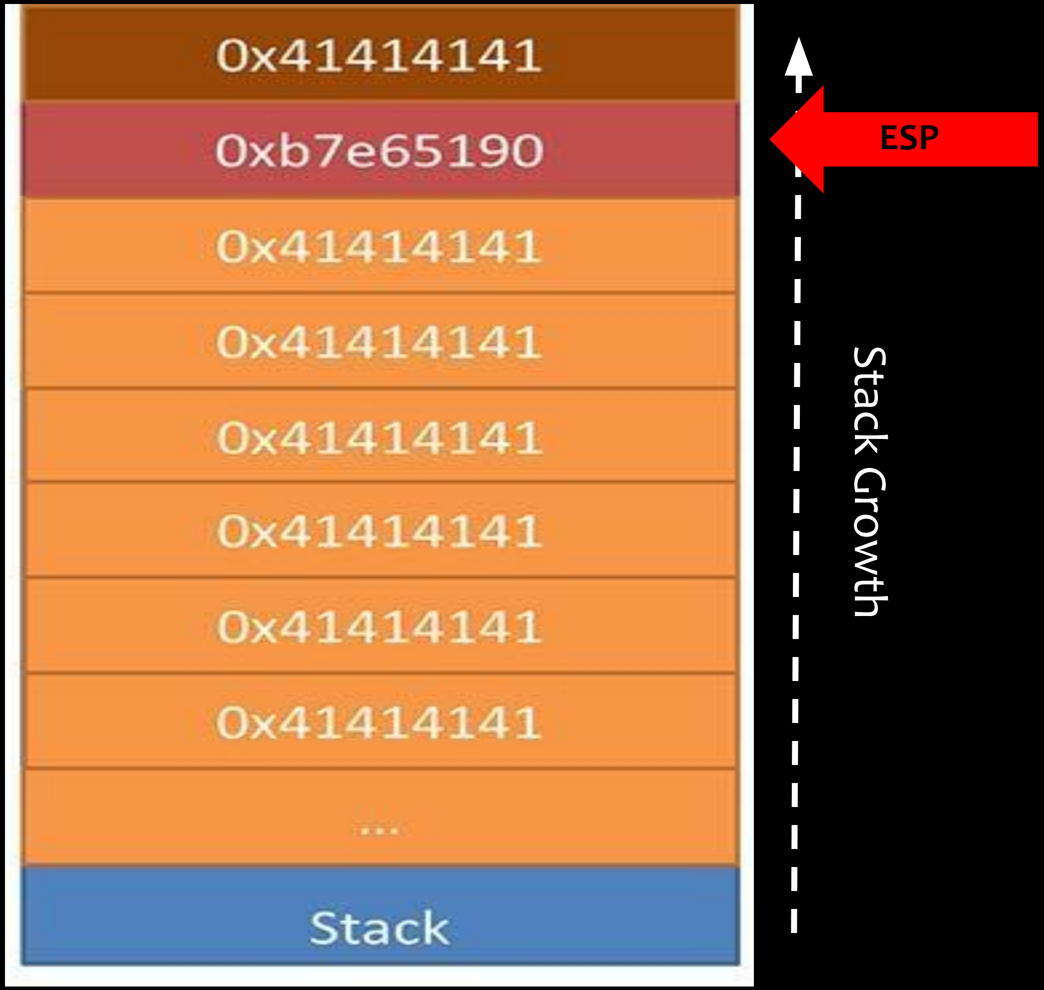
system()

0xb7e65190: push ebx

0xb7e65191: sub esp, 8

0xb7e65194: mov eax, DWORD PTR [esp+0x10]

...



# Returning to System

- We want to call system (“cat flag.text”);
- Because we are ROPing into system rather than calling it, you have to think about setting up the stack (to pass arguments) a little bit differently

# ret2libc example

system ()

0x08045430: ret

EIP

system()

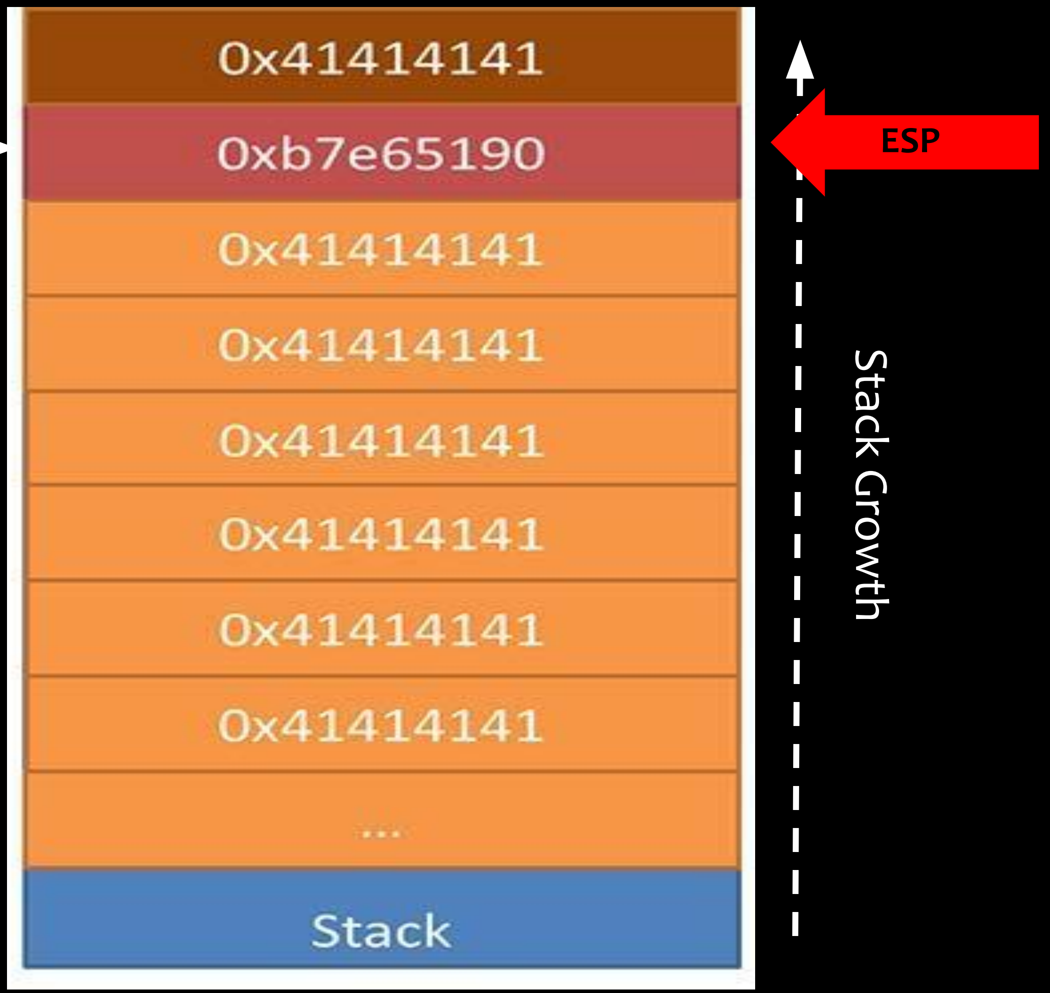
0xb7e65190: push ebx

0xb7e65191: sub esp, 8

0xb7e65194: mov eax, DWORD PTR

[esp+0x10]

...



# ret2libc example

0x08045430: ret

system()

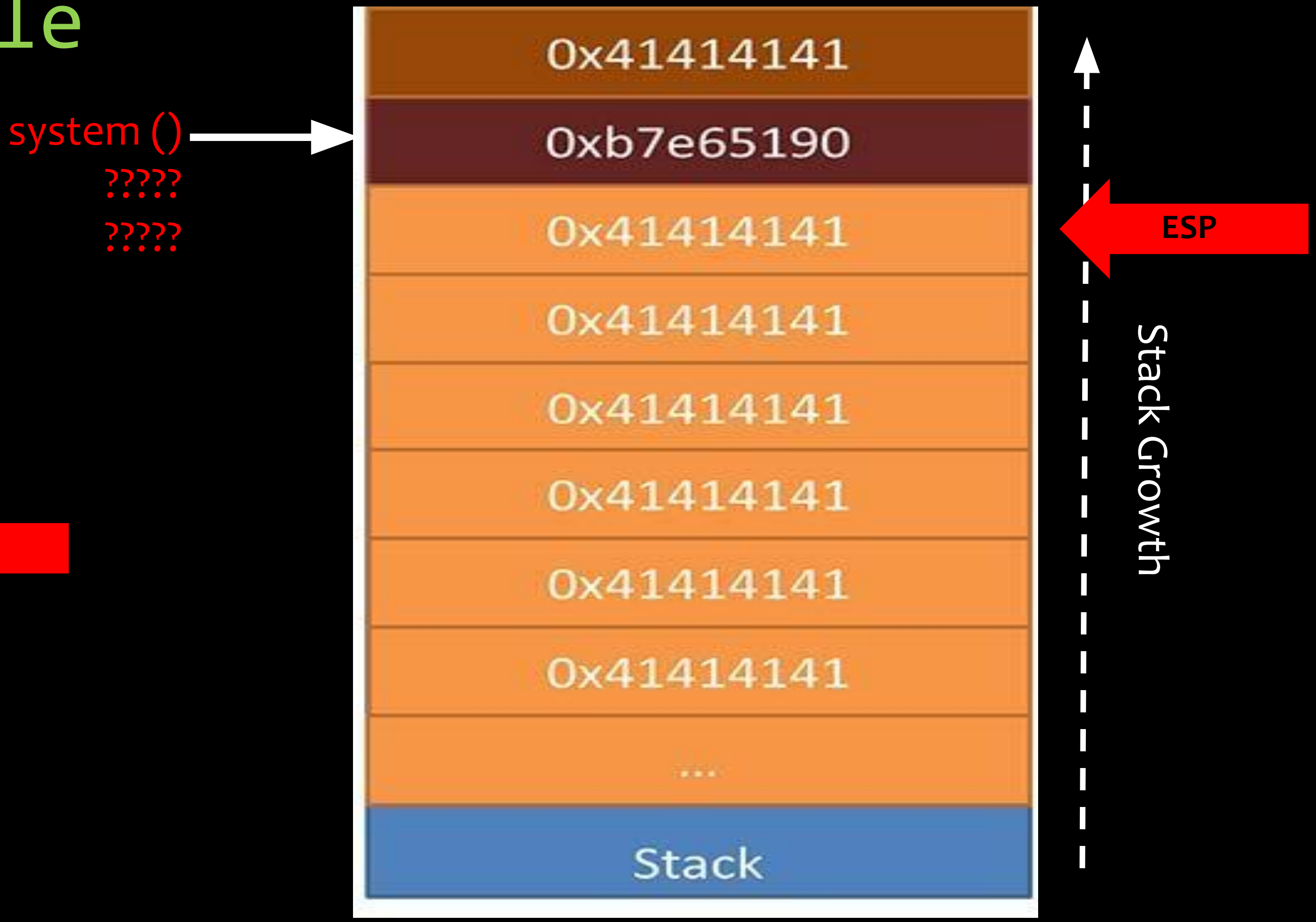
0xb7e65190: push ebx

EIP

0xb7e65191: sub esp, 8

0xb7e65194: mov eax, DWORD PTR [esp+0x10]

...



# ret2libc example

0x08045430: ret

---

system()

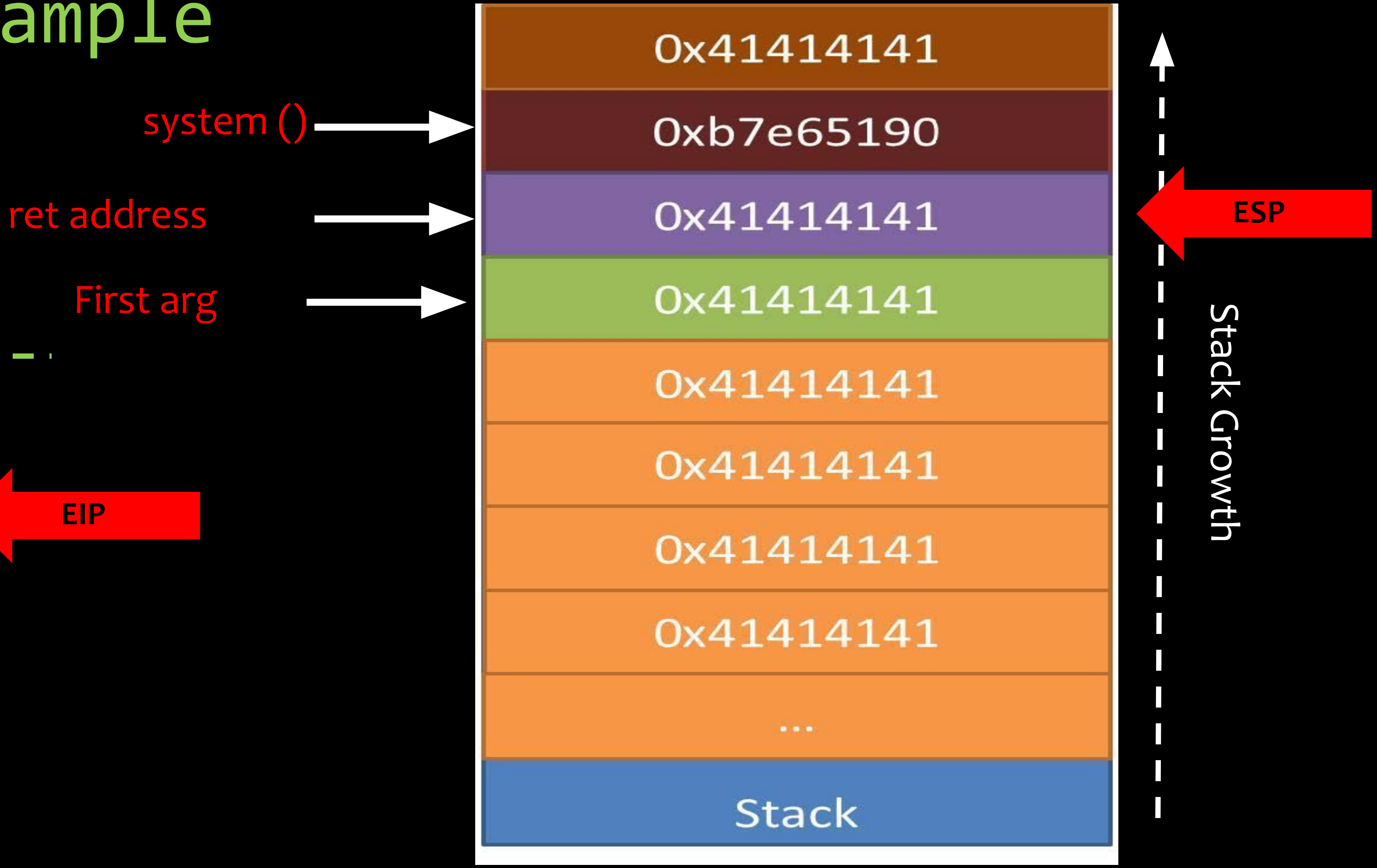
0xb7e65190: push ebx

EIP

0xb7e65191: sub esp, 8

0xb7e65194: mov eax, DWORD PTR [esp+0x10]

...



# ret2libc example

system()

```
0xb7e65190: push  ebx
0xb7e65191: sub   esp, 8
0xb7e65194: mov   eax, DWORD PTR [esp+0x10]
...
```

# ret2libc example

0x08045430: ret

- - - - - - - - - - - - - - - - - - - - - - -

system()

0xb7e65190: push ebx

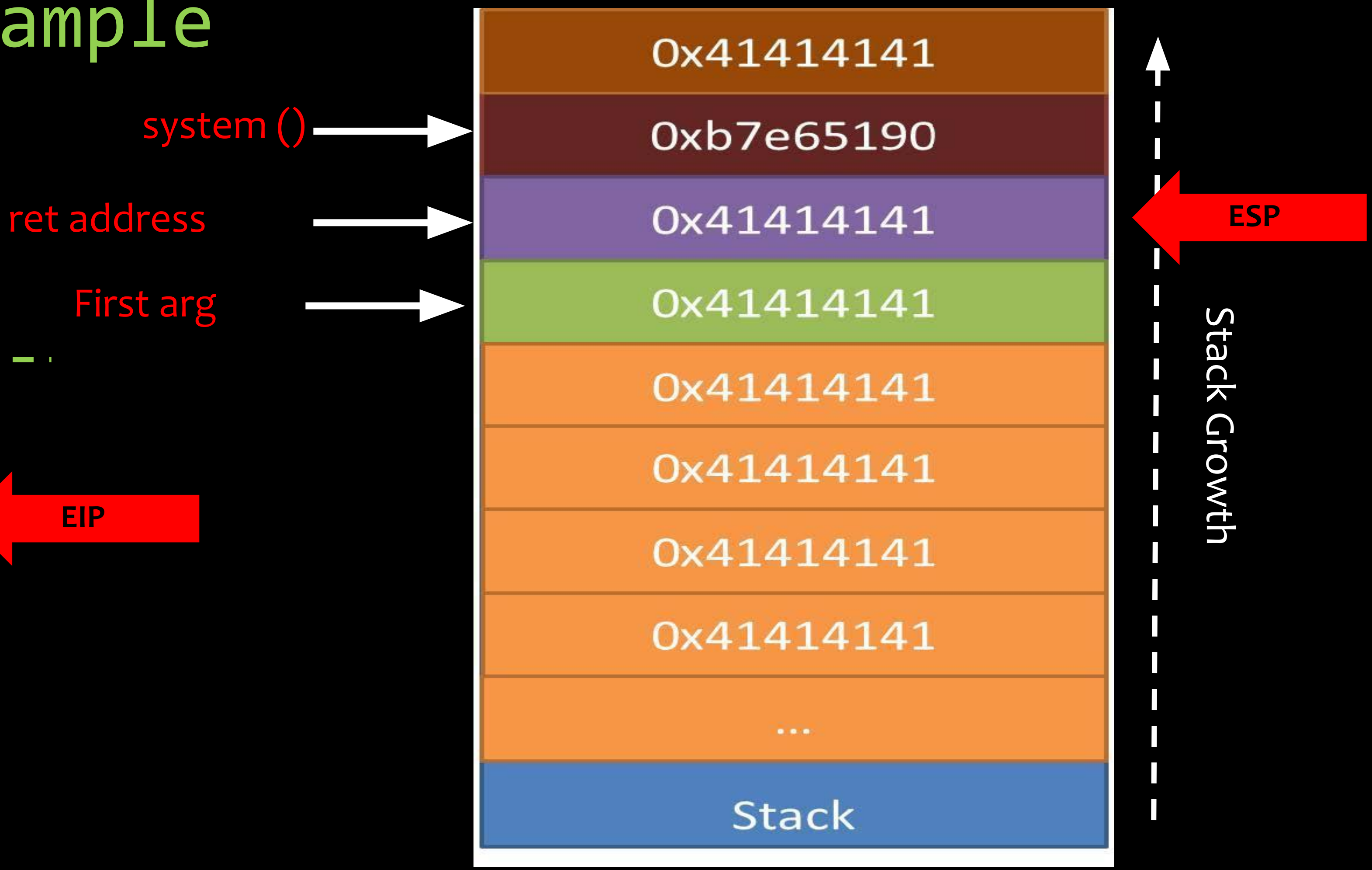
0xb7e65191: sub esp, 8

0xb7e65194: mov eax, DWORD PTR

[esp+0x10]

...

EIP



# ret2libc example

system ()'s
Stack frame
ret address

First arg

0x08045430: ret

system()

0xb7e65190: push ebx

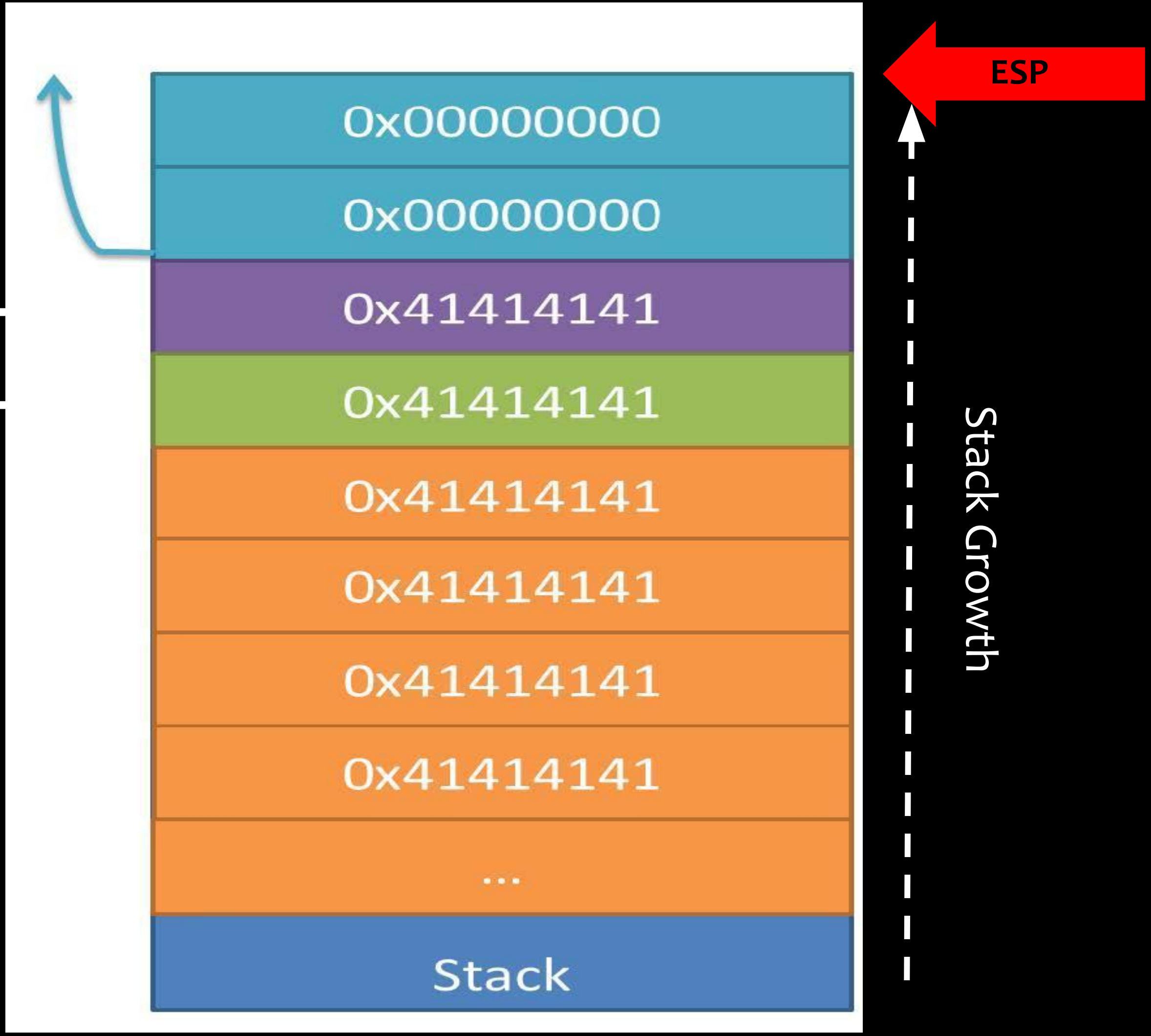
0xb7e65191: sub esp, 8

0xb7e65194: mov eax, DWORD PTR

[esp+0x10]

...

EIP



# ret2libc example

0x08045430: ret

system()

0xb7e65190: push ebx

0xb7e65191: sub esp, 8

0xb7e65194: mov eax, DWORD PTR [esp+0x10]

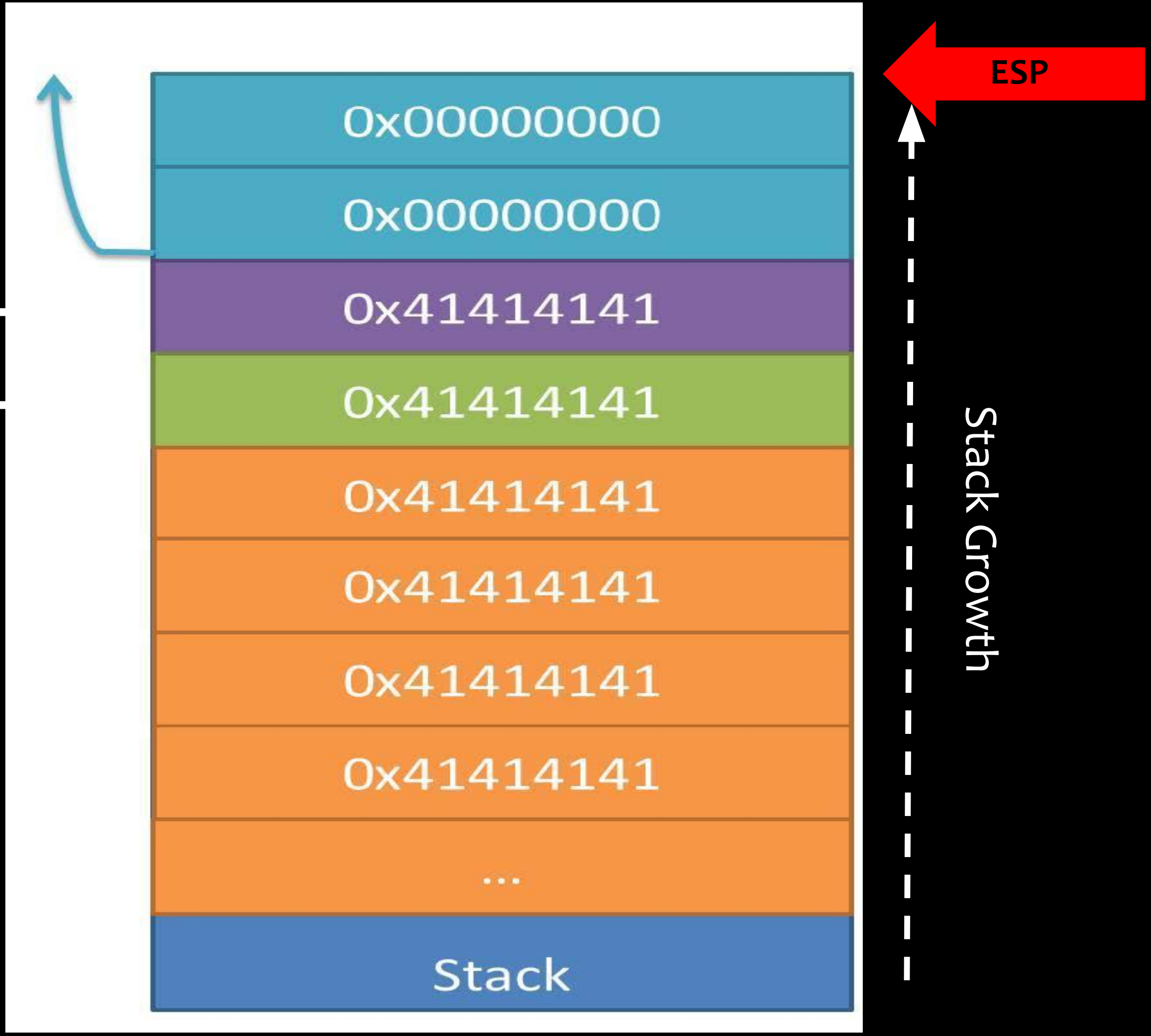
EIP

...

system ()'s Stack frame

ret address

First arg



# REWIND

# ret2libc example

system ()

0x08045430: ret

EIP

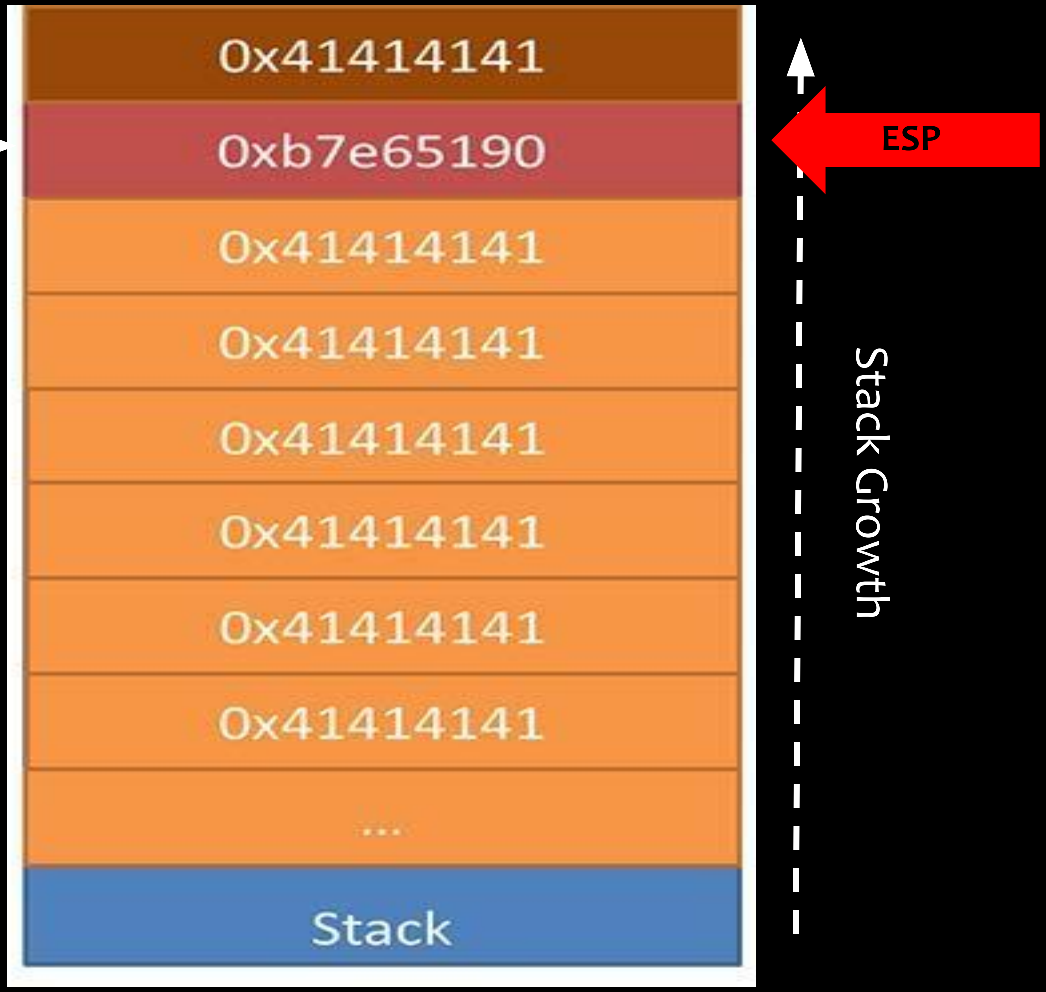
system()

0xb7e65190: push ebx

0xb7e65191: sub esp, 8

0xb7e65194: mov eax, DWORD PTR [esp+0x10]

...



# ret2libc example

0x08045430: ret

EIP

---

system()

0xb7e65190: push ebx

0xb7e65191: sub esp, 8

0xb7e65194: mov eax, DWORD PTR

[esp+0x10]

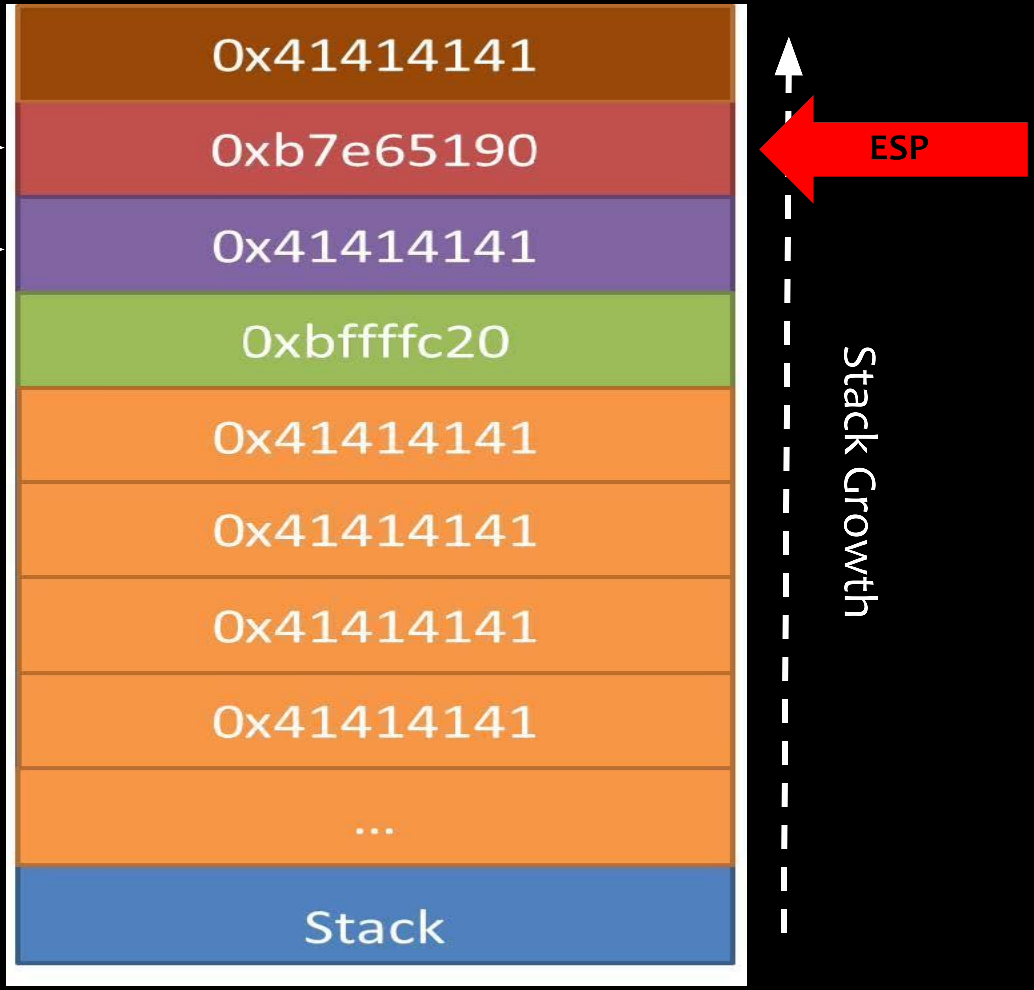
...

system ()'s
Stack frame

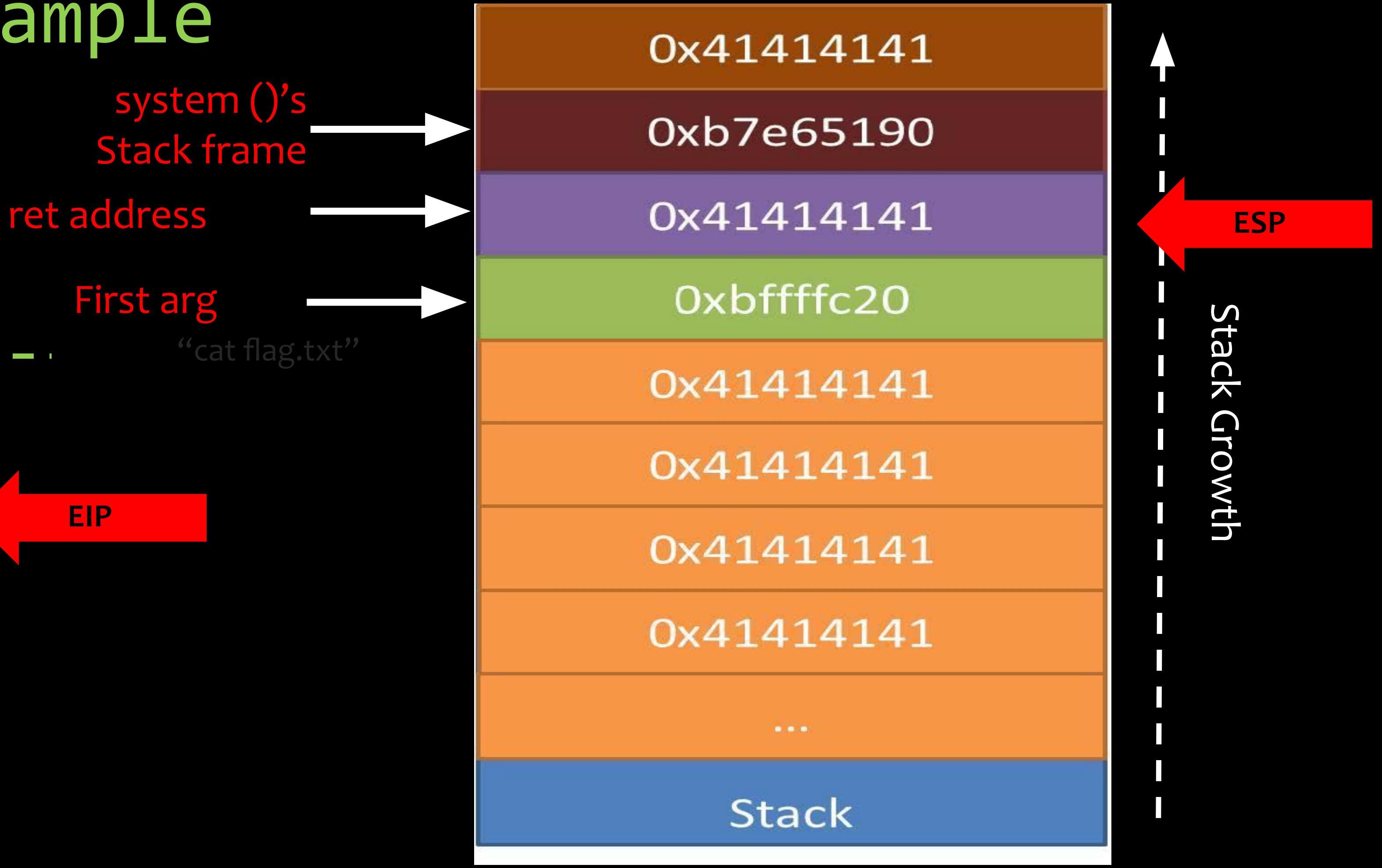
ret address

First arg

"cat flag.txt"



# ret2libc example




0x08045430: ret

---

system()

0xb7e65190: push ebx

EIP

0xb7e65191: sub esp, 8

0xb7e65194: mov eax, DWORD PTR [esp+0x10]

...

# ret2libc example

0x08045430: ret

- - - - - - - - - - - - - - - - - - - -

system()

0xb7e65190: push ebx

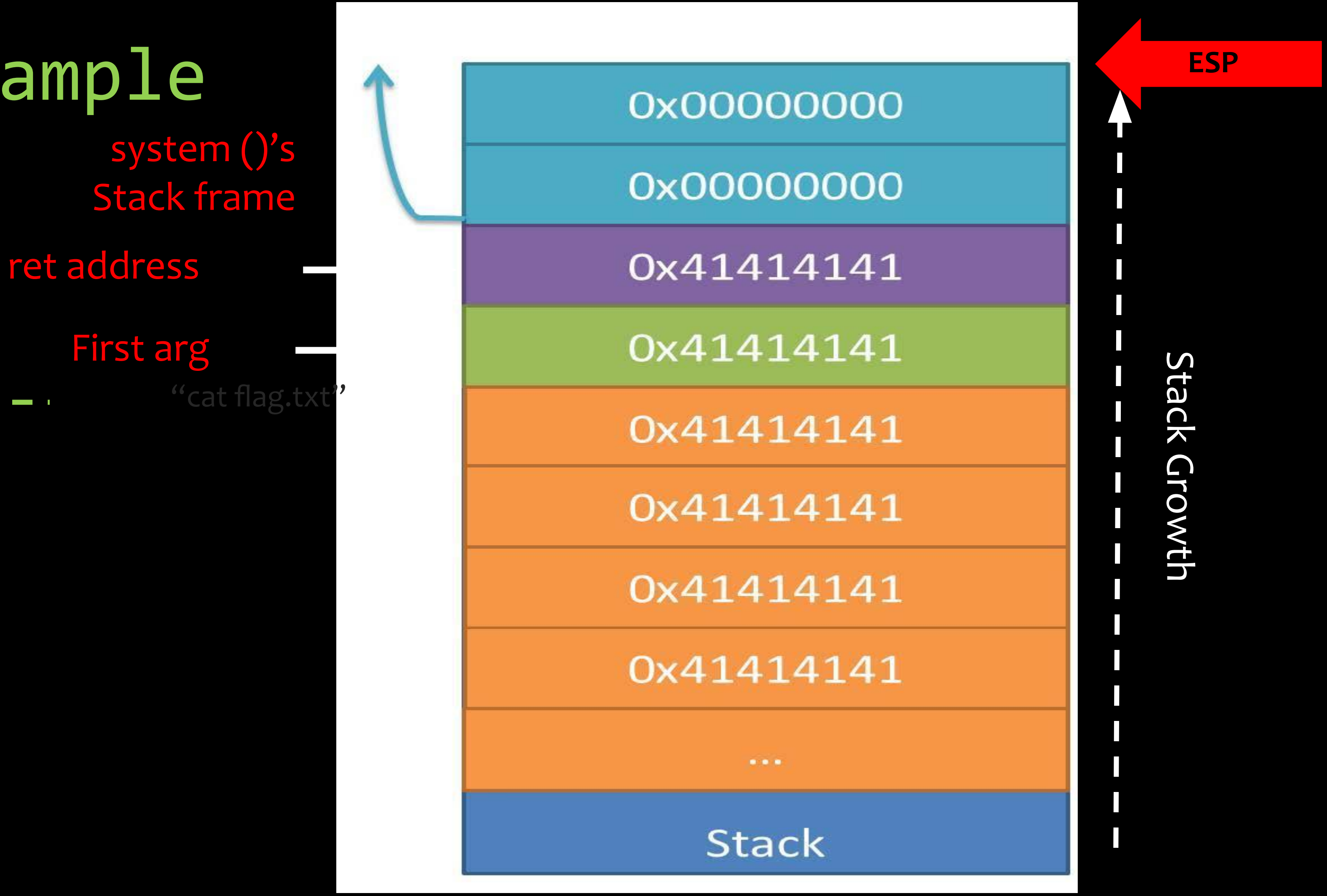
0xb7e65191: sub esp, 8

0xb7e65194: mov eax, DWORD PTR

[esp+0x10]

EIP

...



# ret2libc example

system ()'s
Stack frame

ret address

First arg

0x08045430: ret

WOW_U_got_th3_fl4g_such_h4ck3r

sy

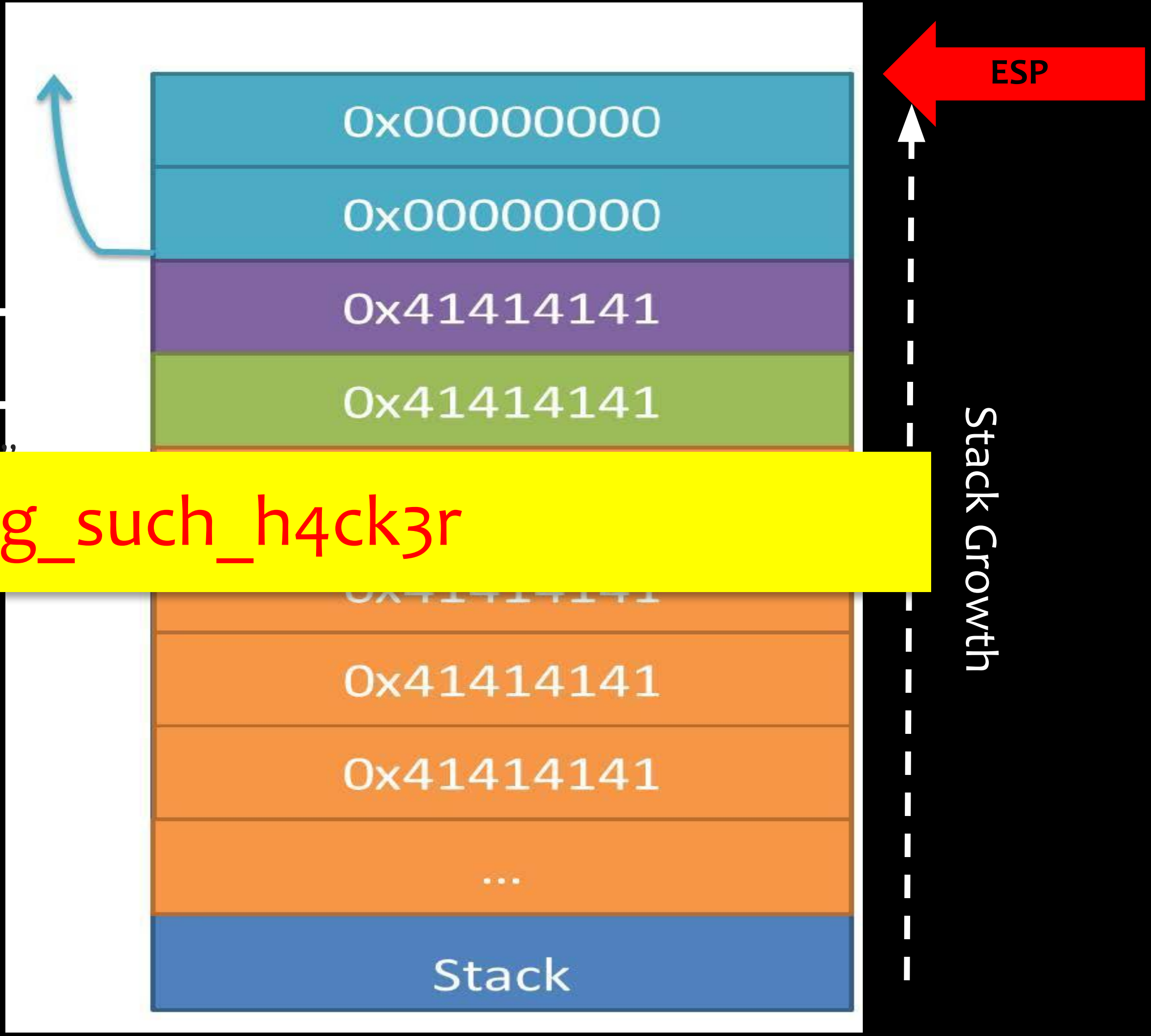
0xb7e65190: push ebx

0xb7e65191: sub esp, 8

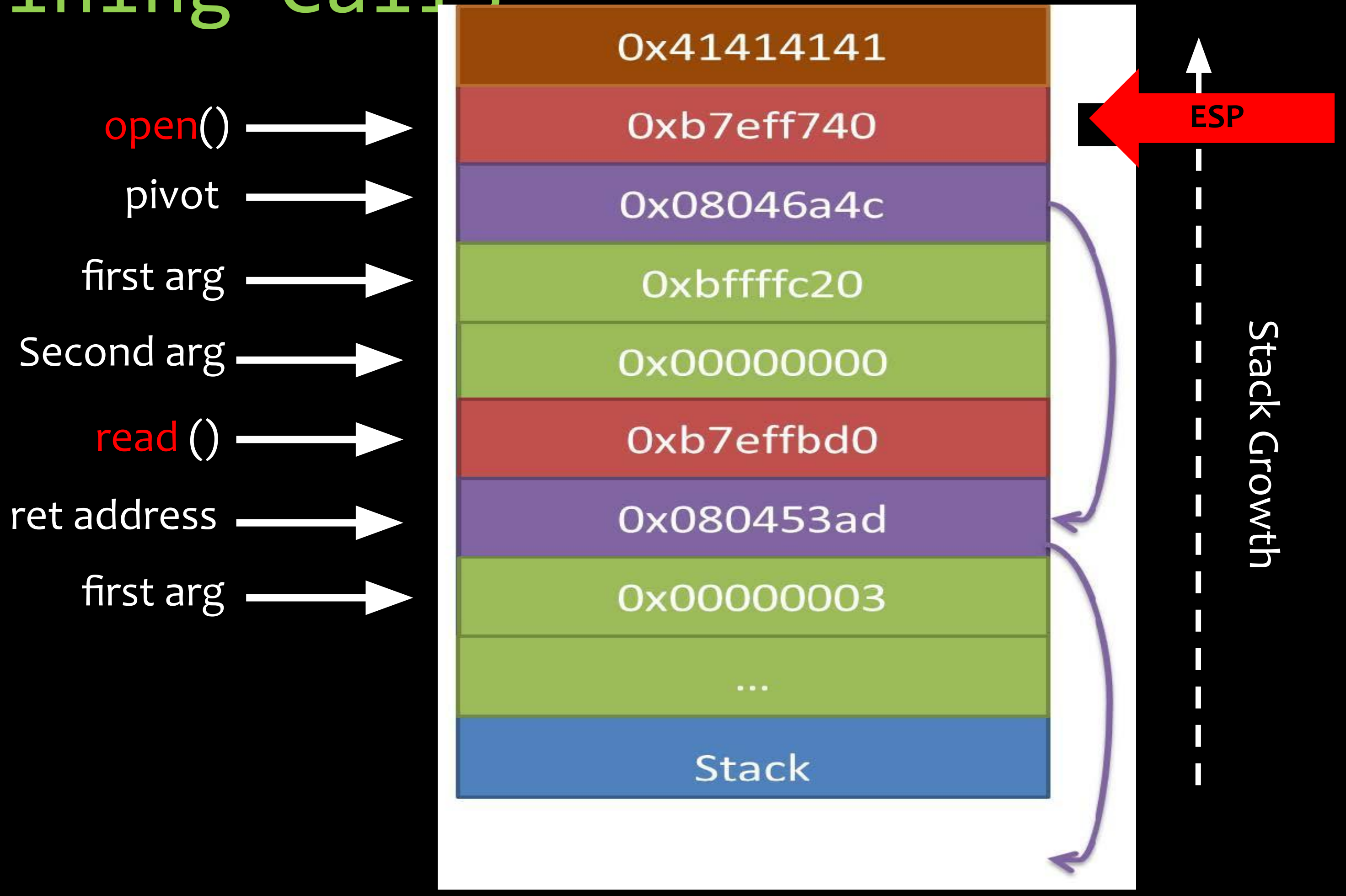
0xb7e65194: mov eax, DWORD PTR [esp+0x10]

EIP

...



# Chaining Calls



# Am I out of time?

- Probably
  - The  rest of this slide deck is adapted from Modern Binary Exploitation from RPISEC
- The first two challenges can be solved with the first part of this slide deck.
  - https://drive.google.com/open?id=0B5Sor8VFNaEEVE9qMk1JWGZWTzQ

# Address Space Layout Random Randomization

ASLR

# Lecture Overview

1. Introducing ASLR
2. Position Independent Executables
3. Bypassing ASLR, Examples
4. Conclusion

# Modern Exploit Mitigations

- There's a number of modern exploit mitigations that we've generally been turning off for the labs and exercises
  - DEP
  - ASLR
  - Stack Canaries
  - … ?

# Modern Exploit Mitigations

- There's a number of modern exploit mitigations that we've generally been turning off for the labs and exercises
  - DEP
  - ASLR
  - Stack Canaries
  - … ?
- We turned on DEP and introduced ROP last lab

# Modern Exploit Mitigations

- There's a number of modern exploit mitigations that we've generally been turning off for the labs and exercises
  - DEP
  - ASLR
  - Stack Canaries
  - … ?
- We turned on DEP and introduced ROP last lab
- Today we turn ASLR back on for the remainder of the course

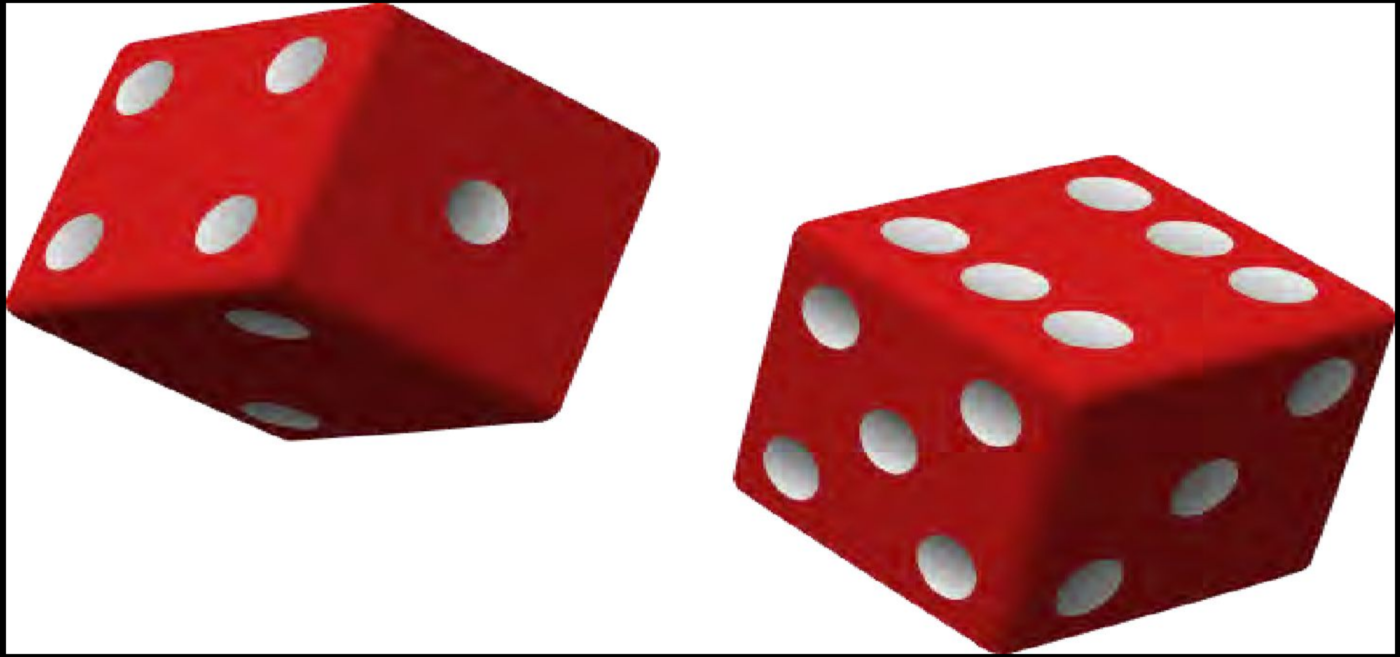
# What is ASLR?

A: Address

S: Space

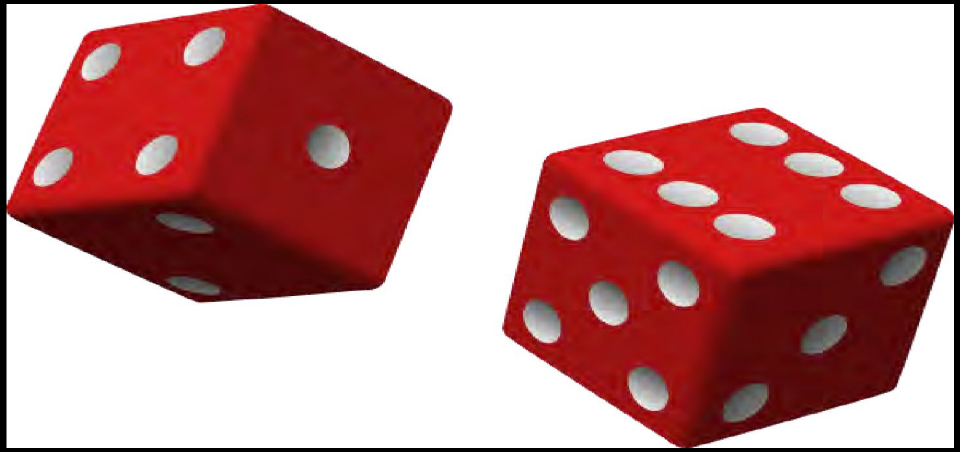
L: Layout

R: Randomization



# Course Terminology

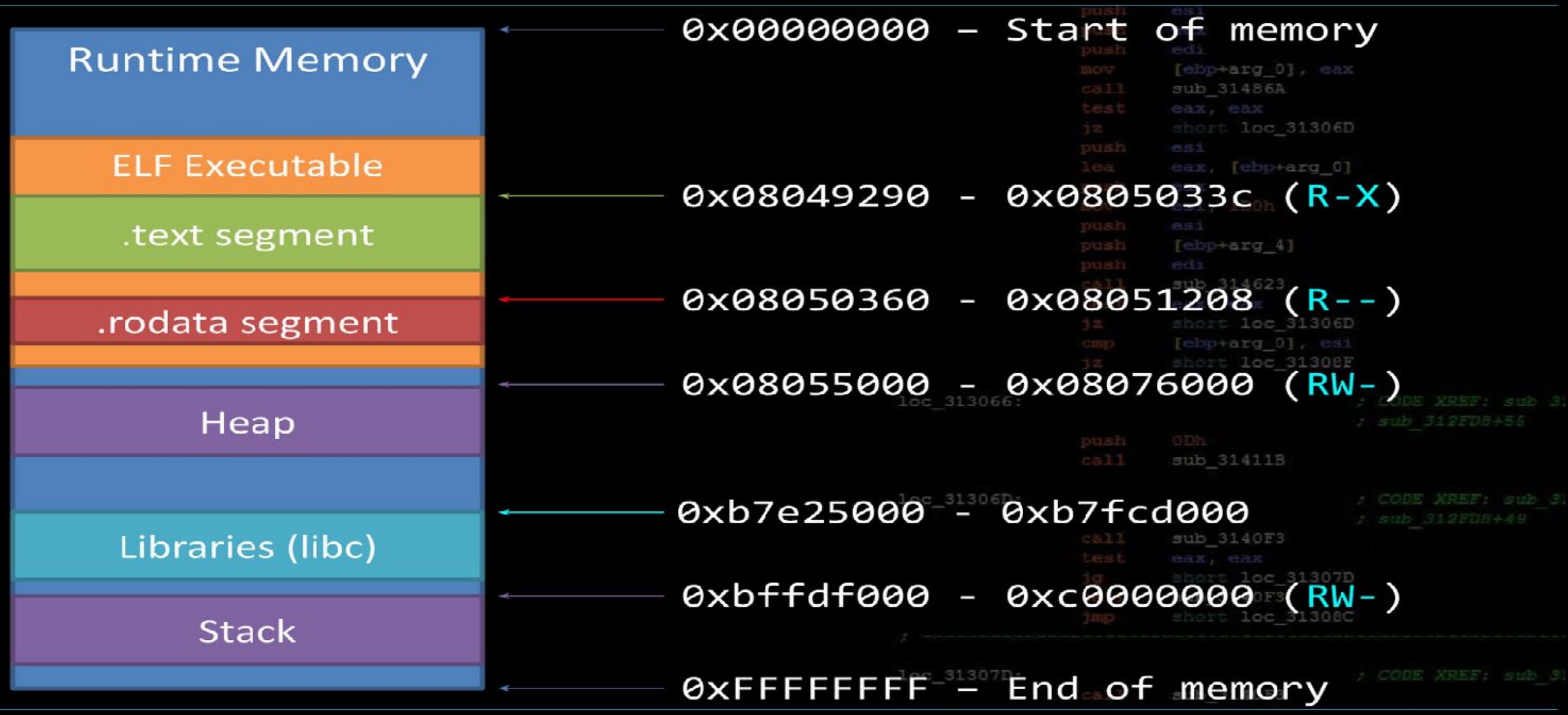
- Address Space Layout Randomization
  - An exploit mitigation technology used to ensure that address ranges for important memory segments are random for every execution
  - Meant to mitigate exploits leveraging hardcoded stack, heap, code, libc addresses
  - Known as ASLR for short



# Runtime Process Without ASLR

# Run #1 Without ASLR

# Run #2 Without ASLR

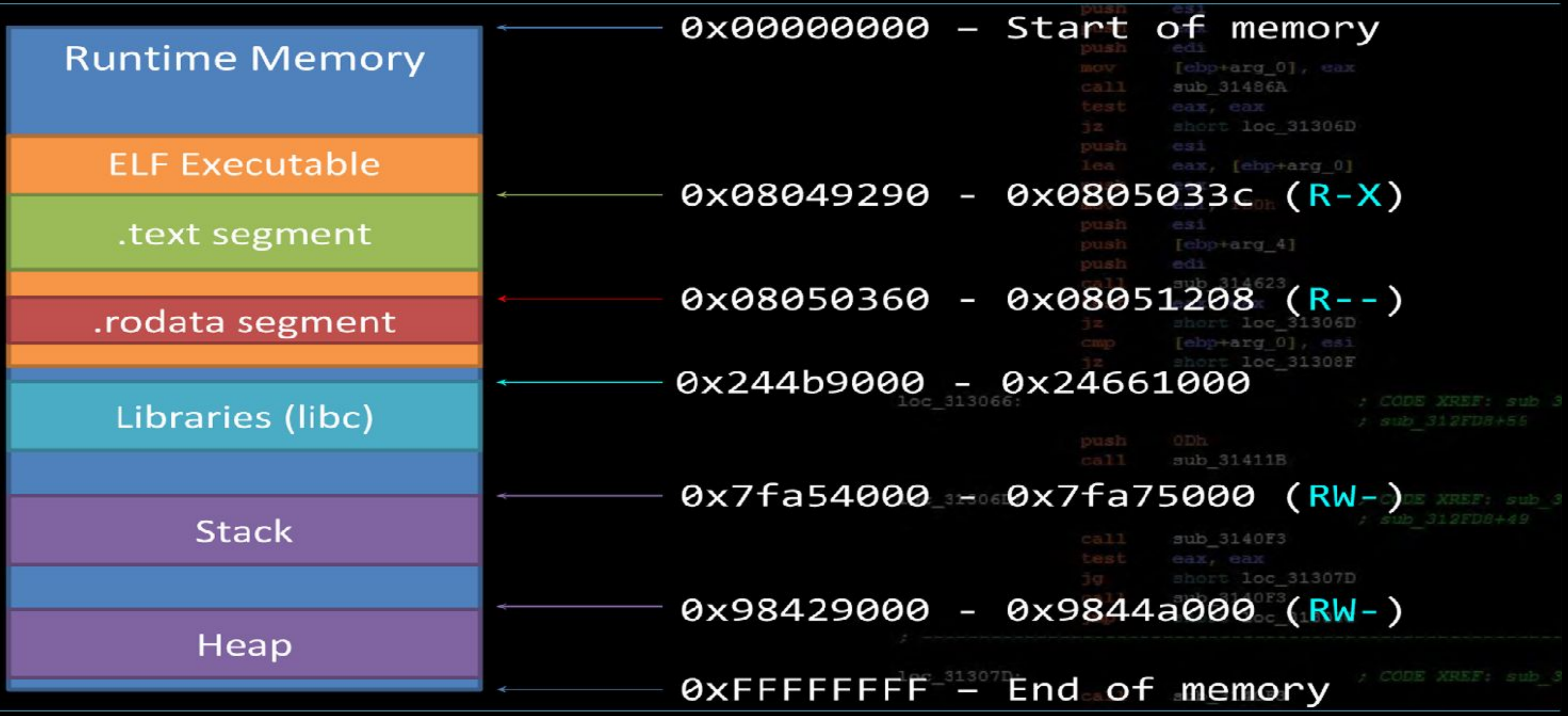


# Run #3 Without ASLR

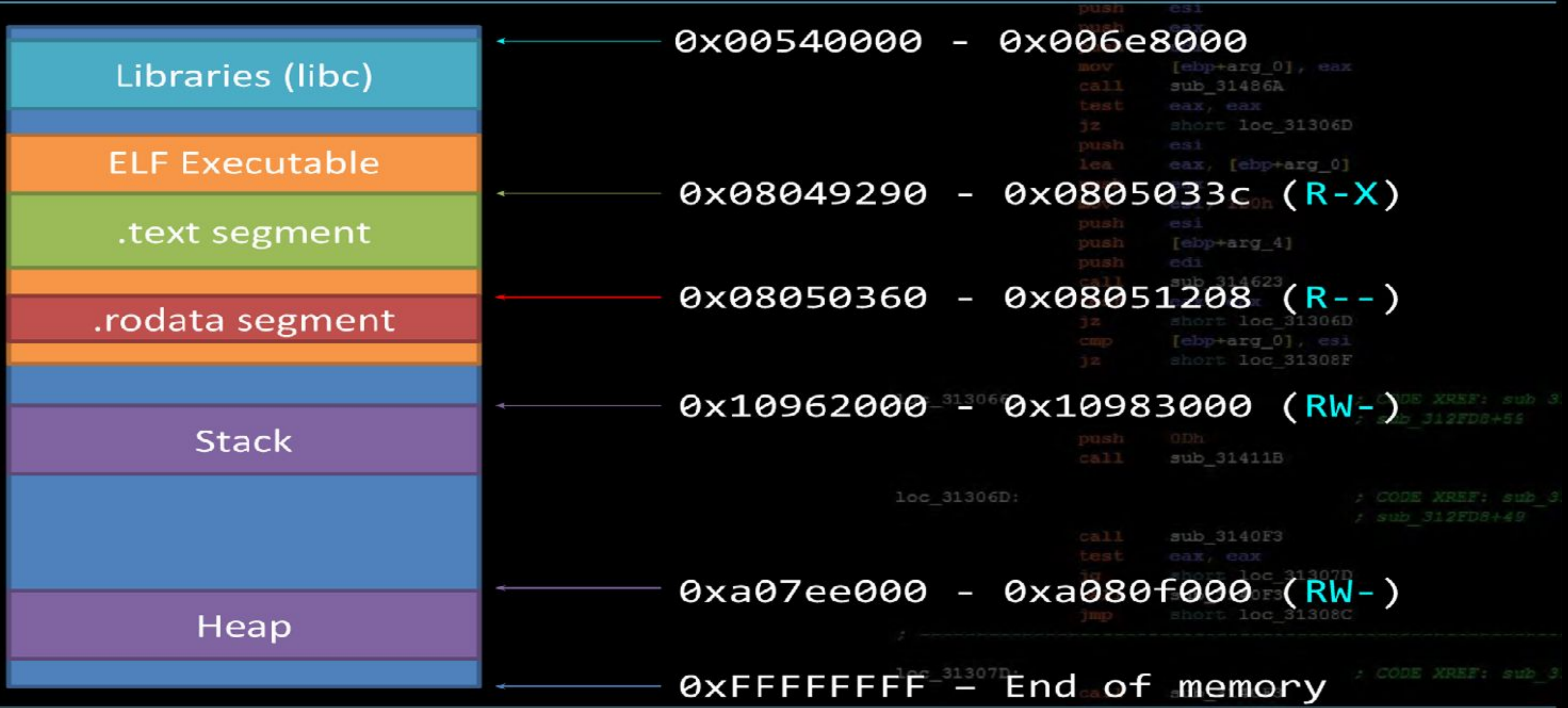
ya so, nothing changes …

# Runtime Process With~~out~~ ASLR

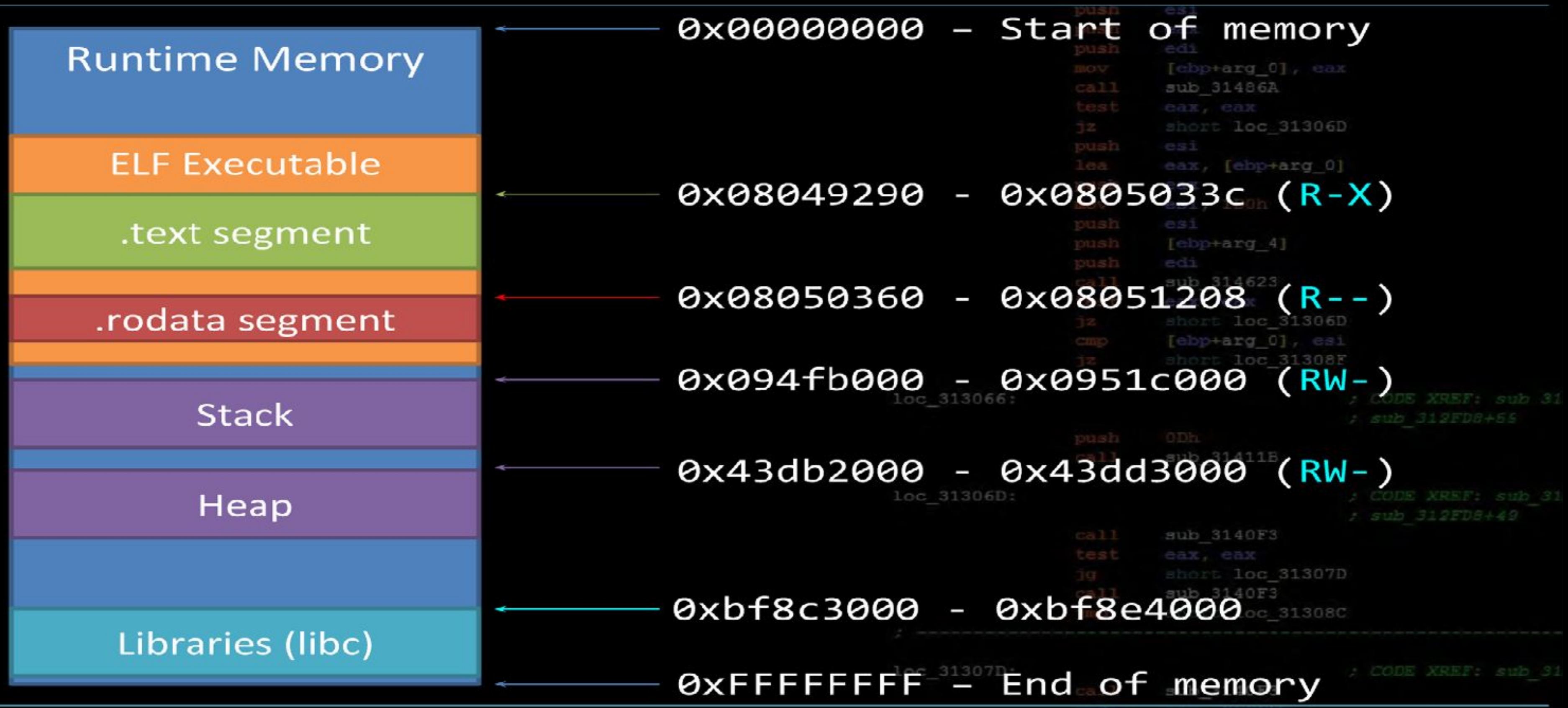
# Run #1 With ASLR



# Run #2 With ASLR



# Run #3 With ASLR



# ASLR in Action

> Open up a terminal.

# ASLR in Action

> Open up a terminal.

> Type "cat /proc/self/maps"

# ASLR in Action

> Open up a terminal.

> Type “cat /proc/self/maps”

> Repeat a few times :)

# ASLR in Action

> Open up a terminal.

> Type "cat /proc/self/maps"

> Repeat a few times :)

You'll see lots of lines like this:

```
bfe49000-bfe6a000 rw-p 00000000 00:00 0      [stack]
...
bfa23000-bfa44000 rw-p 00000000 00:00 0      [stack]
...
bfdab000-bfdcc000 rw-p 00000000 00:00 0      [stack]
```

# ASLR in Action

> Open up a terminal.

> Type “cat /proc/self/maps”

> Repeat a few times :)

- Stack Address Changes

# ASLR in Action

> Open up a terminal.

> Type “cat /proc/self/maps”

> Repeat a few times :)

- Stack Address Changes
- Heap Address Changes

# ASLR in Action

> Open up a terminal.

> Type “cat /proc/self/maps”

> Repeat a few times :)

- Stack Address Changes
- Heap Address Changes
- Library Addresses Change

# ASLR Basics

- Memory segments are no longer in static address ranges, rather they are unique for every execution

# ASLR Basics

- Memory segments are no longer in static address ranges, rather they are unique for every execution
- A simple stack smash may get you control of EIP, but what does it matter if you have no idea where you can go with it?

# ASLR Basics

- Memory segments are no longer in static address ranges, rather they are unique for every execution
- A simple stack smash may get you control of EIP, but what does it matter if you have no idea where you can go with it?
  - The essence of ASLR
- You must work with no expectation of where anything is in memory anymore

# History of ASLR

- When was ASLR implemented?
  - May $1^{st}$, 2004 – OpenBSD 3.5 (mmap)
  - June 17th, 2005 – Linux Kernel 2.6.12 (stack, mmap)
  - January 30th, 2007 – Windows Vista (full)
  - October 26th, 2007 – Mac OSX 10.5 Leopard (sys libraries)
  - October 21st, 2010 – Windows Phone 7 (full)
  - March 11th, 2011 – iPhone IOS 4.3 (full)
  - July 20th, 2011 – Mac OSX 10.7 Lion (full)

Reminder:

**Security is rapidly evolving**

# Checking for ASLR

```
$ cat / proc/sys/kernel/randomize_va_space
```

# Checking for ASLR

```
$ cat / proc/sys/kernel/randomize_va_space
2
```

# Checking for ASLR

$ cat / proc/sys/kernel/randomize_va_space

2

---

0: No ASLR

1: Conservative Randomization

(Stack, Heap, Shared Libs, PIE, mmap(), VDRO)

2: Full Randomization
(Conservative Randomization + memory managed via brk())

# Lecture Overview

1. Introducing ASLR
2. Position Independent Executables
3. Bypassing ASLR, Examples
4. Conclusion

# ELF's and ASLR

On Linux, not everything is randomized …

# Runtime Process With ASLR

# Run #1 With ASLR

# Run #2 With ASLR

# Run #3 With ASLR

# Not Randomized

- Main ELF Binary
  - .text / .plt / .init / .fini - Code Segments (R-X)
  - .got / .got.plt / .data / .bss - Misc Data Segments (RW-)
  - .rodata - Read Only Data Segment (R--)
- At minimum, we can probably find some ROP gadgets!
  - Warning: They won’t be pretty gadgets

# Course Terminology

- Position Independent Executable
  - Executables compiled such that their base address does not matter, ‘position independent code’
  - Shared Libs /must/ be compiled like this on modern Linux
    - eg: libc
  - Known as PIE for short

# Applying ASLR to ELF's

- To make an executable position independent, you must compile it with the flags -pie-fPIE

```
$ gcc -pie -fPIE  -o tester  tester.c
```

# Applying ASLR to ELF's

- To make an executable position independent, you must compile it with the flags -pie-fPIE

```
$ gcc -pie -fPIE  -o tester  tester.c
```

- Without these flags, you are not taking full advantage of ASLR

# Checking for PIE

- Most binaries aren't actually compiled as PIE

- Generally only on remote services, as
  you don't want your server to get owned

# Lecture Overview

1. Introducing ASLR
2. Position Independent Executables
3. Bypassing ASLR, Examples
4. Conclusion

# Bypassing ASLR

- Assume you can get control of EIP
- What information does ASLR deprive us of?

# Bypassing ASLR

- Assume you can get control of EIP

- What information does ASLR deprive us of?
  - You don't know the address of ANYTHING

# Bypassing ASLR

- Assume you can get control of EIP
- What information does ASLR deprive us of?
  - You don't know the address of ANYTHING
- How can we get that information?
  - Or work around it?

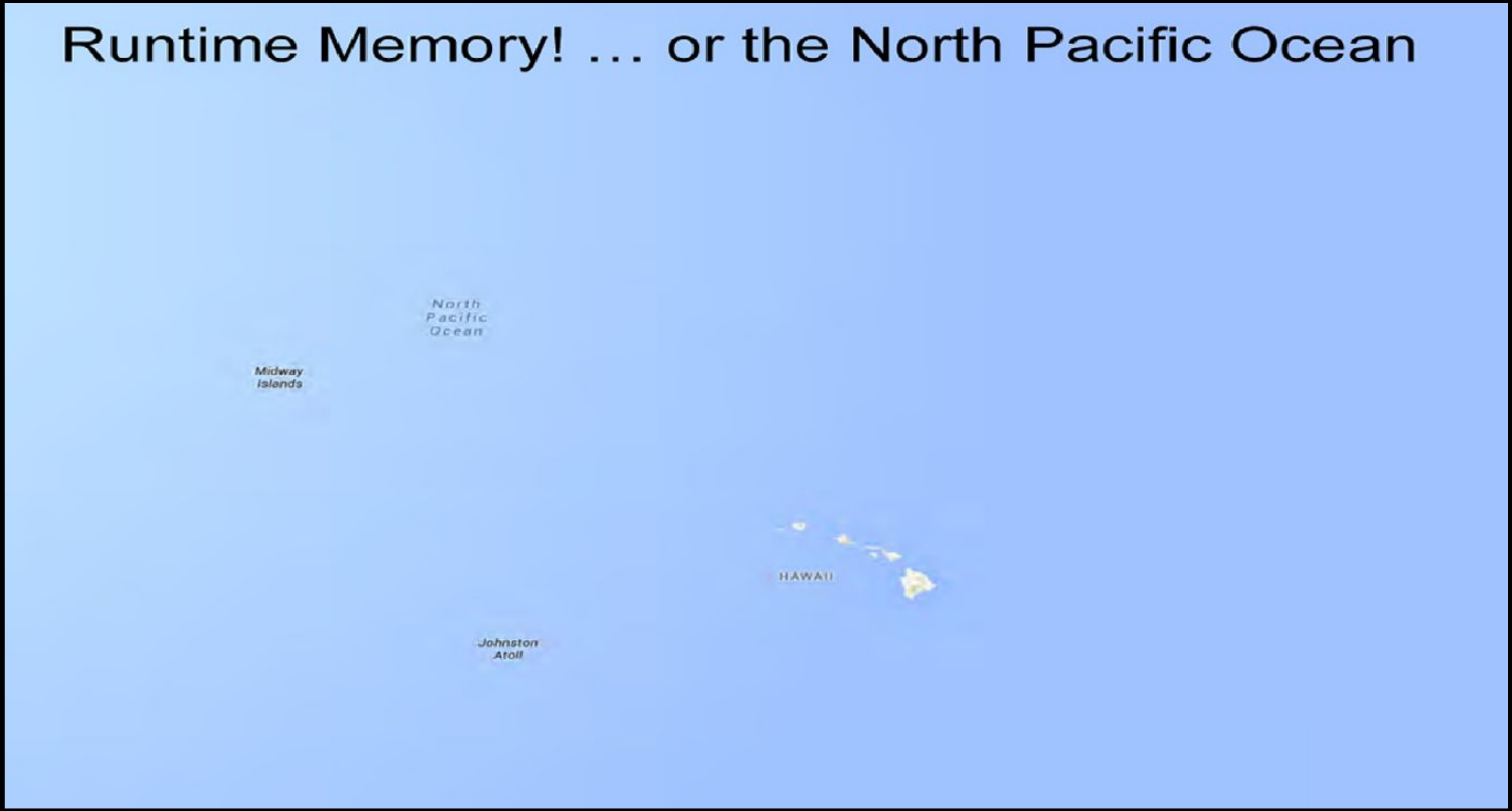
# Bypassing ASLR

- There's a few common ways to bypass ASLR
  - Information disclosure (aka info leak)
  - Partial address overwrite + Crash State
  - Partial address overwrite + Bruteforce

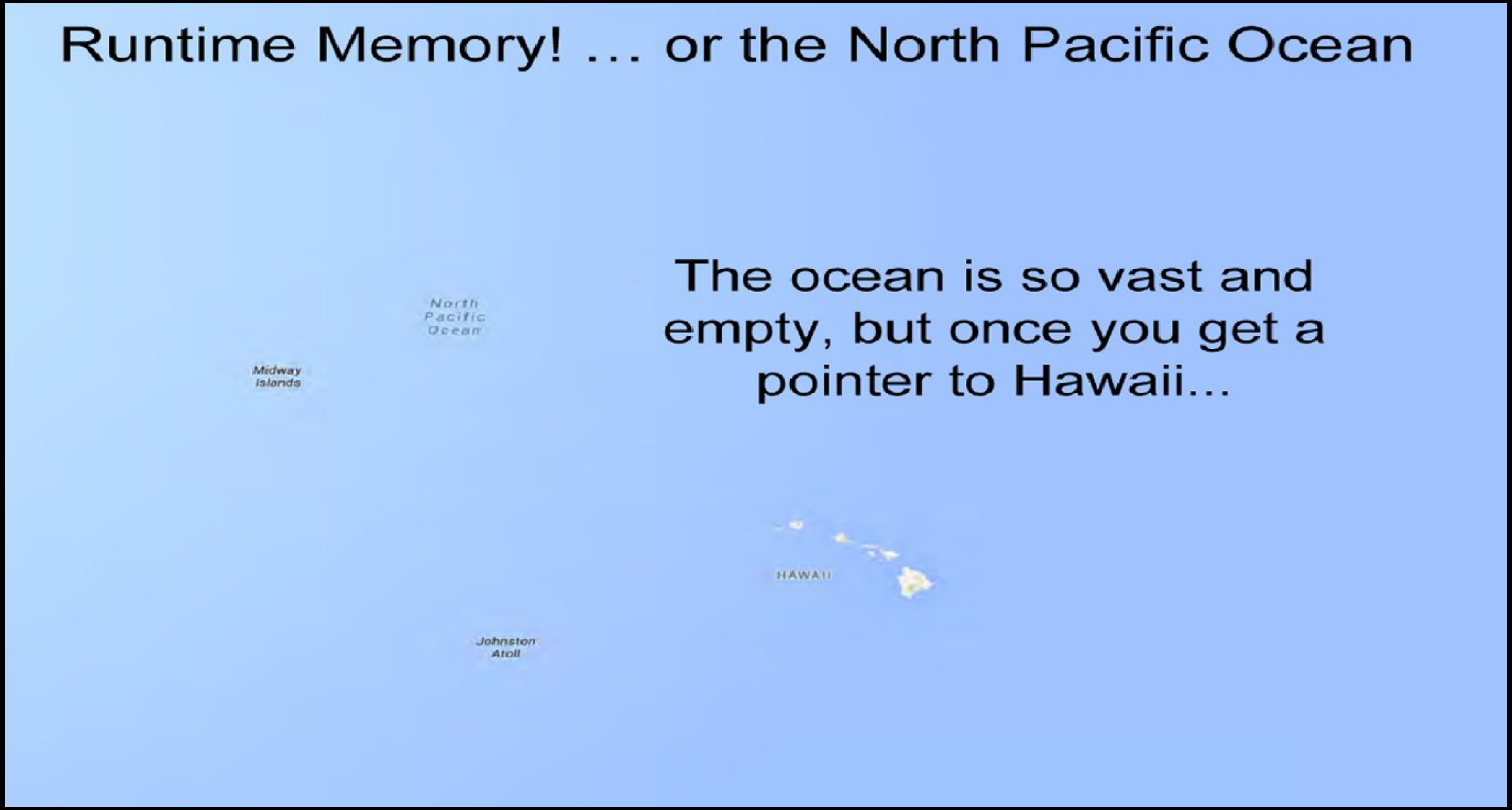
# What are Info Leaks?

- An info leak is when you can extract meaningful information (such as a memory address) from the ASLR protected service or binary
- If you can leak any sort of pointer to code during your exploit, you have likely defeated ASLR
  - Why is a single pointer leak so damning?

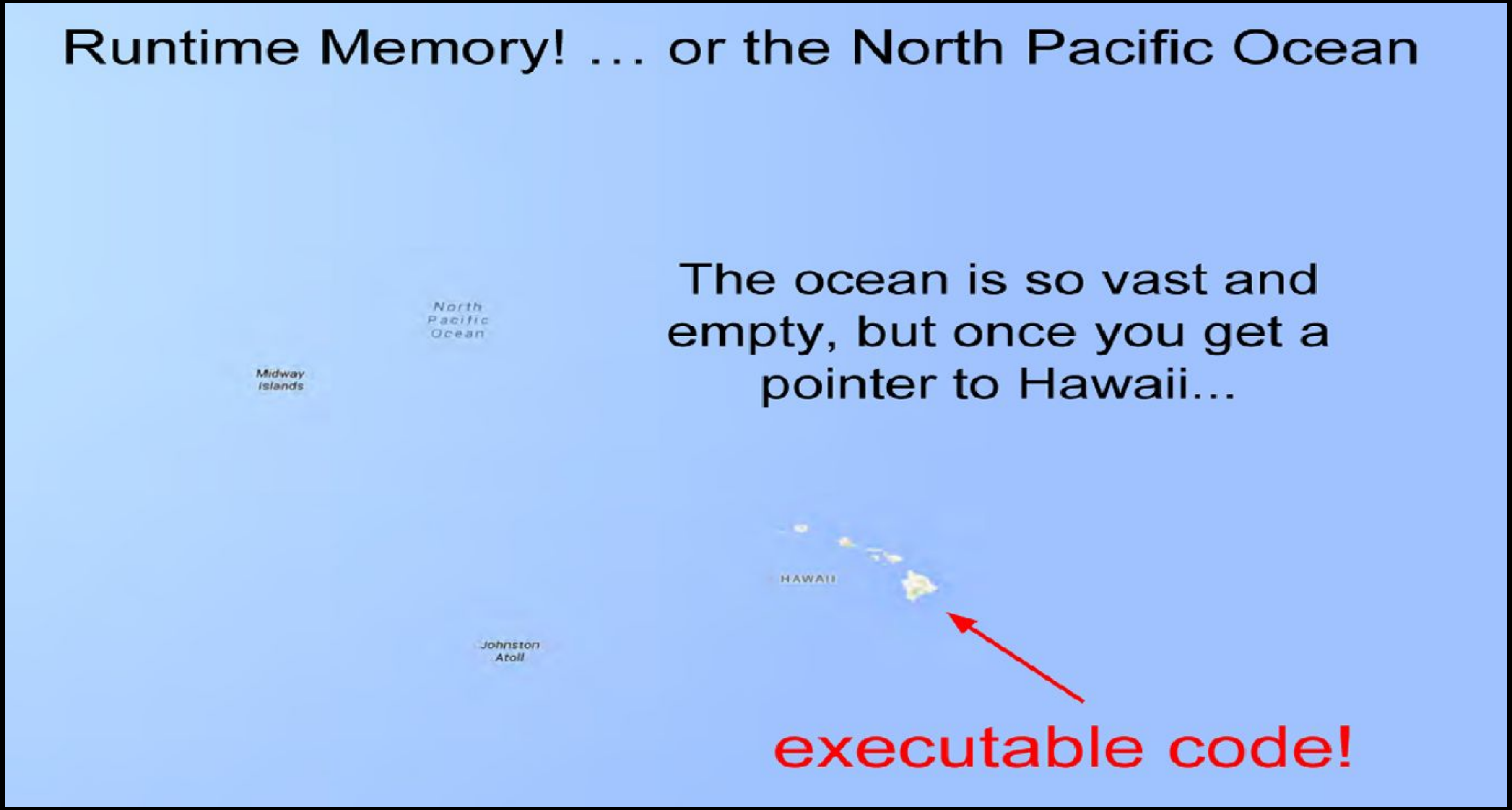
# Death by Pointer



# Death by Pointer



# Death by Pointer



# Death by Pointer

# Death by Pointer

# Using Info Leaks

By Example:

- You have a copy of the libc binary, ASLR is on

# Using Info Leaks

By Example:

- You have a copy of the libc binary, ASLR is on
- You've leaked a pointer off the stack to printf()
  printf() is @ 0xb7e72280

# Using Info Leaks

By Example:

- You have a copy of the libc binary, ASLR is on
- You've leaked a pointer off the stack to printf()
  printf() is @ 0xb7e72280
- Look at the libc binary, how far away is system() from printf()?
  system() is -0xD0F0 bytes away from printf()

# Using Info Leaks

By Example:

- You have a copy of the libc binary, ASLR is on
- You've leaked a pointer off the stack to printf()
  printf() is @ 0xb7e72280
- Look at the libc binary, how far away is system() from printf()?
  system() is -0xD0F0 bytes away from printf()

therefore system() is at @0xb7e65190
(0xb7e65190-0xD0F0)

# Using Info Leaks

- Can be used on hardest scenario of PIE, full ASLR
  - Usually comes with 100% exploit reliability!
  - ‘it just works’

- Info leaks are the most used ASLR bypass in real world exploitation as they give assurances

# Partial Overwrites

- Assume you have no way to leak an address, but you can overwrite one

from multiple runs:

```
0xb756b132
0xb758e132
0xb75e5132
0xb754d132
0xb75cf132
```

Guaranteed 255 byte ROP/ret range around that address

24 bits of bruteforce gives you 64kb of range around the addr

212 bits of bruteforce will give you ROP/ret across all of libc

# Partial Overwrites

- Assume you have no way to leak an address, but you can overwrite one

from multiple runs:

```
0xb756b132
0xb758e132
0xb75e5132
0xb754d132
0xb75cf132
```

100% exploit reliability

6.25% exploit reliability

0.024% exploit reliability

# Bruteforcing

- Note that these bruteforcing details apply only to Ubuntu 32bit
- Don’t bother to try bruteforcing addresses on a 64bit machine of any kind
- Ubuntu ASLR is rather weak, low entropy

# ASLR Tips

- What does your crash state look like?
  - What's in the registers?
  - What's on the stack around you?

- Even if you can't easily leak some data address out of a register or off the stack, there's nothing that's stopping you from using it for stuff
  - As always: get creative

# Lecture Overview

1. Introducing ASLR
2. Position Independent Executables
3. Bypassing ASLR, Examples
4. Conclusion

# In Closing

- Like other mitigation technologies, ASLR is a ‘tack on’ solution that only makes things harder
- The vulnerabilities and exploits become both more complex and precise the deeper down the rabbit hole we go

# Modern Exploit Mitigations

- DEP & ASLR are the two main pillars of modern exploit mitigation technologies

- Congrats, being able to bypass these mean that you’re probably capable of writing exploits for real vulnerabilities