Address Space Layout Randomization

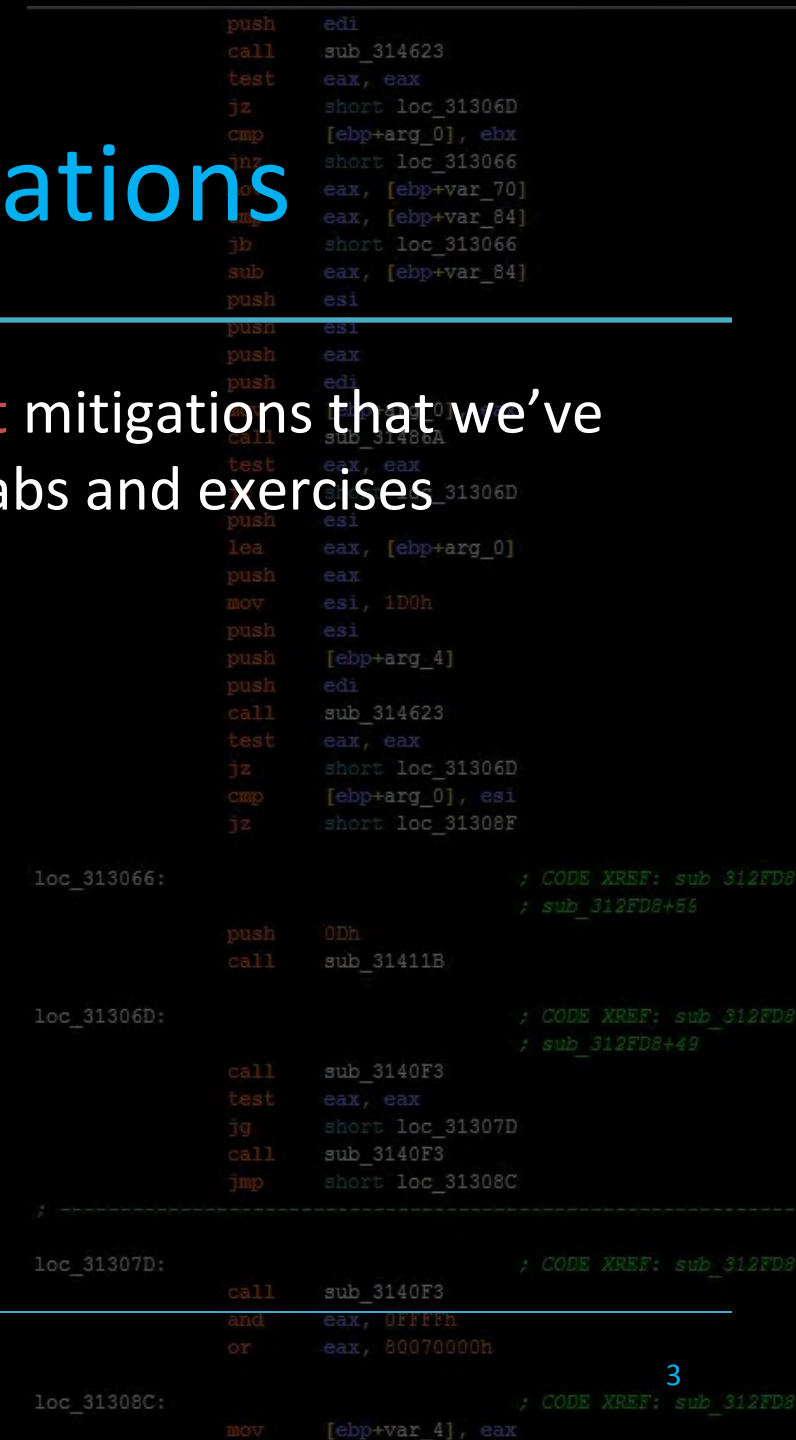
Modern Binary Exploitation
CSCI 4968 - Spring 2015
Patrick Biernat

Lecture Overview

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

Modern Exploit Mitigations

- Theres 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

- Theres 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

- Theres 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

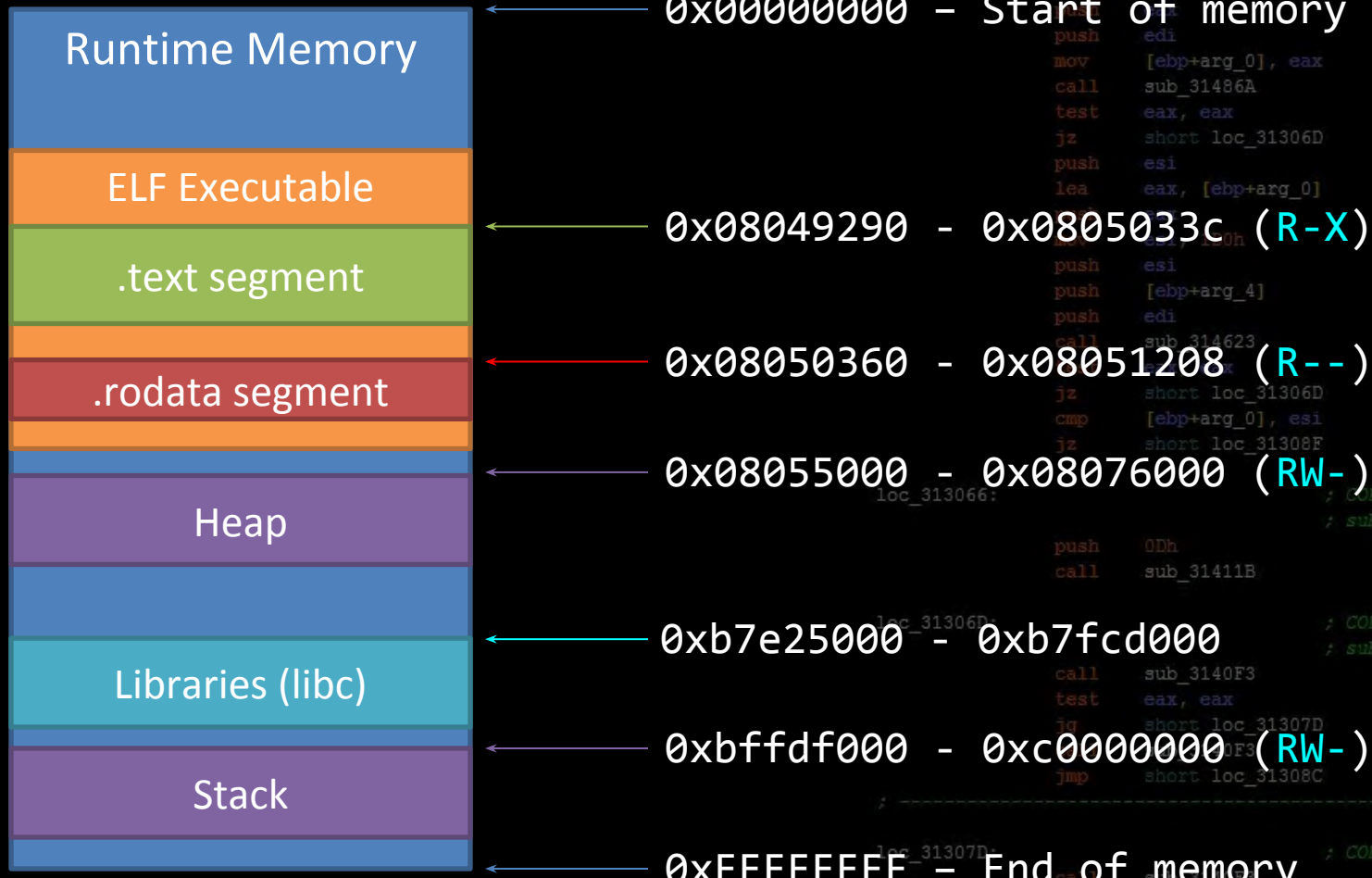
L: Layout

R: Randomization

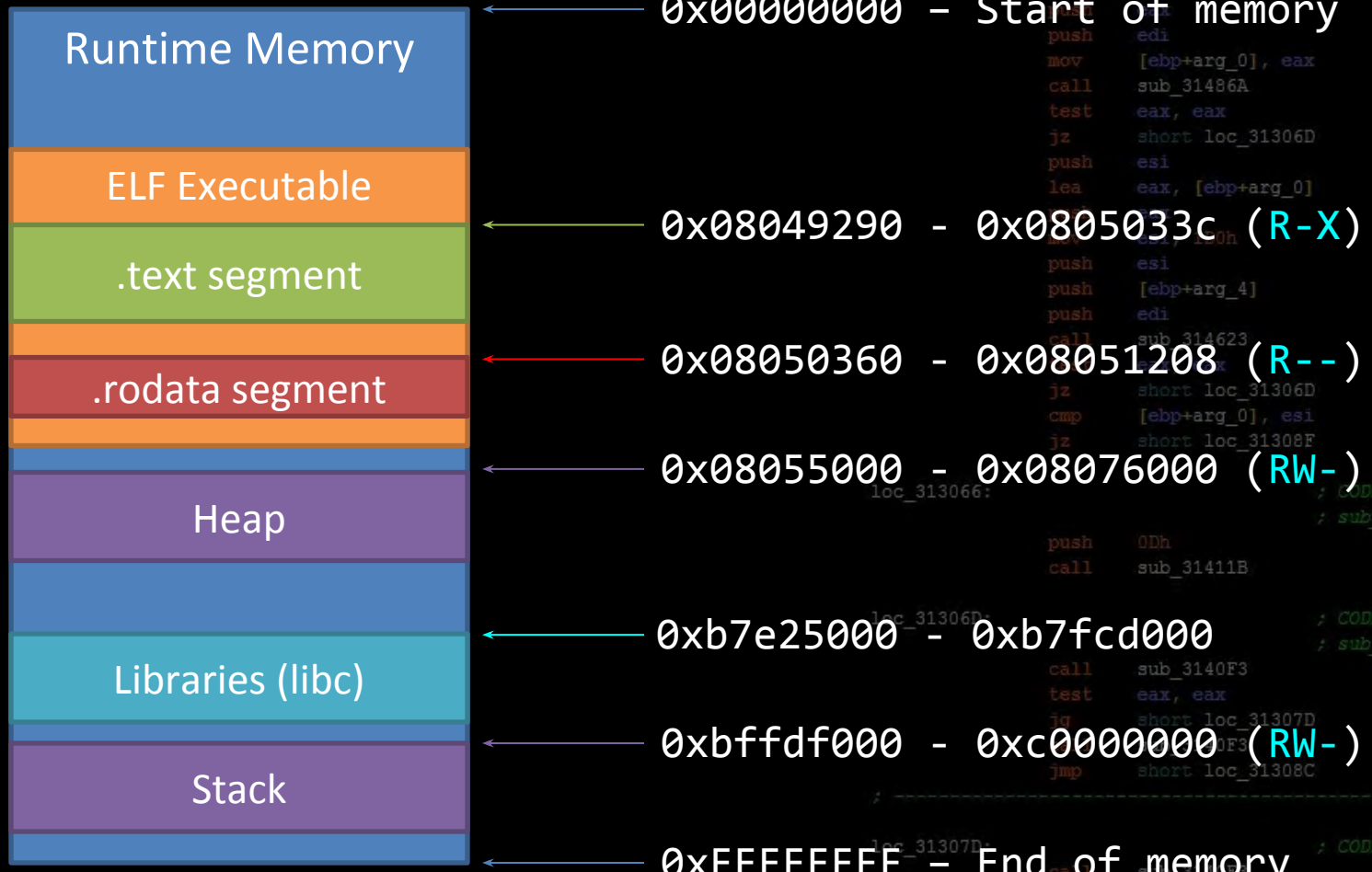
Course Terminology

- 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

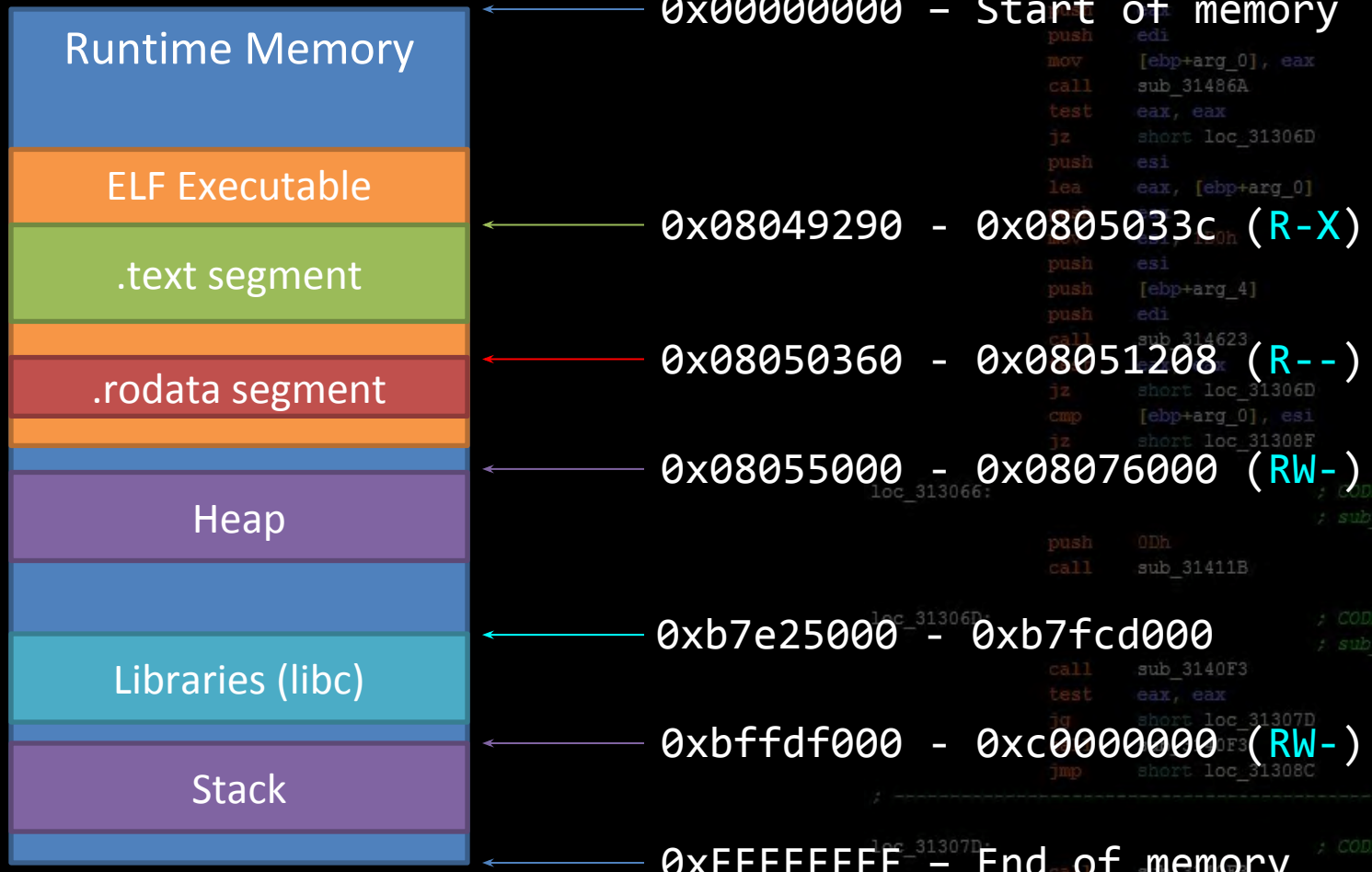
Runtime Process Without ASLR



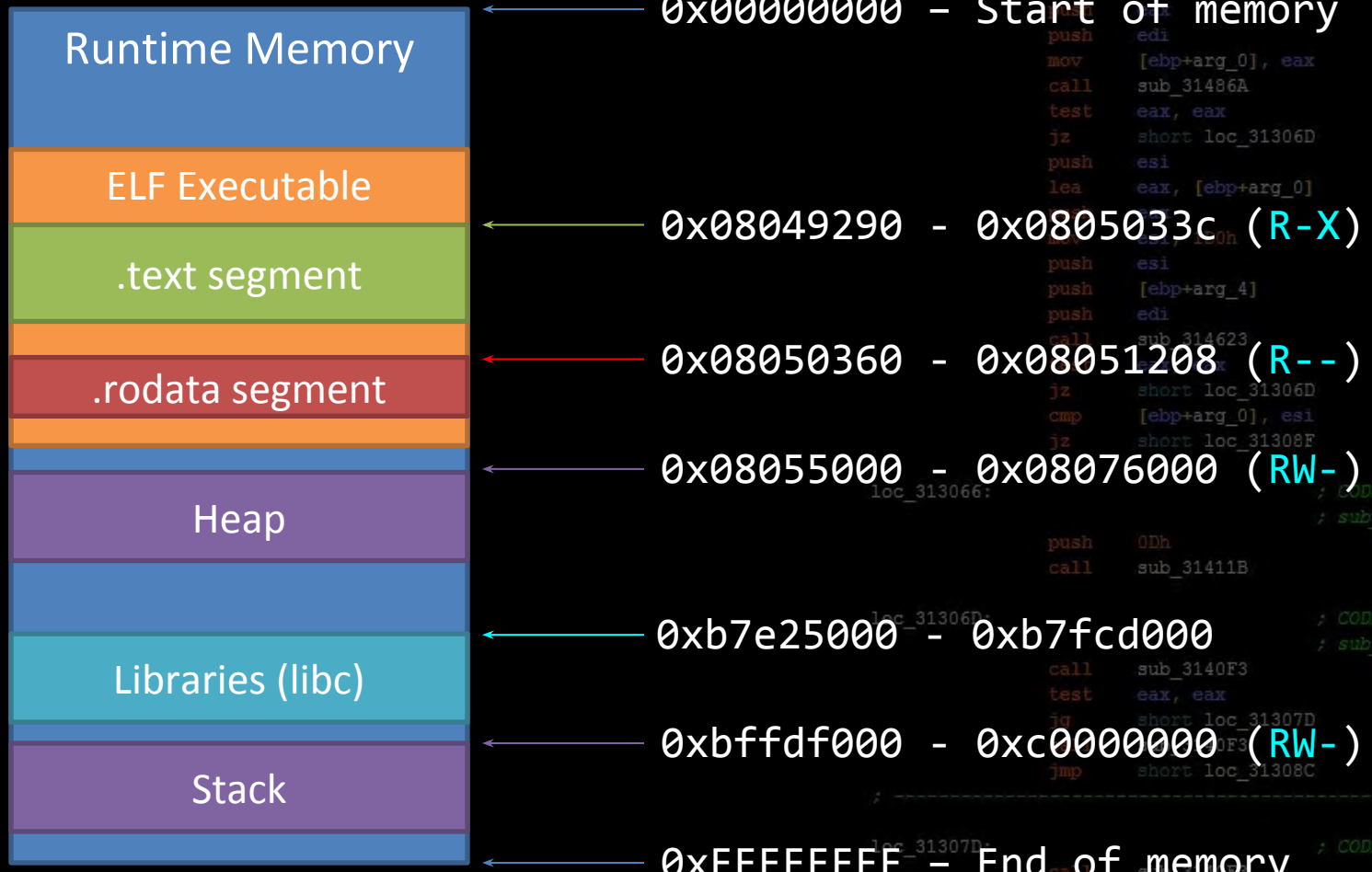
Run #1 Without ASLR



Run #2 Without ASLR



Run #3 Without ASLR



ya so, nothing changes...

```
push    edi
call    sub_314623
test    eax, eax
jz      short loc_31306D
cmp     [ebp+arg_0], ebx
jnz     short loc_313066
mov     eax, [ebp+var_70]
cmp     eax, [ebp+var_84]
jb      short loc_313066
sub     eax, [ebp+var_84]
push    esi
push    esi
push    eax
push    edi
mov     [ebp+arg_0], eax
call    sub_31486A
test    eax, eax
jz      short loc_31306D
push    esi
lea     eax, [ebp+arg_0]
push    eax
mov     esi, 1D0h
push    esi
push    [ebp+arg_4]
push    edi
call    sub_314623
test    eax, eax
jz      short loc_31306D
cmp     [ebp+arg_0], esi
jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+59
push    0Dh
call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
call    sub_3140F3
test    eax, eax
jg      short loc_31307D
call    sub_3140F3
jmp     short loc_31308C
; ---------------------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
call    sub_3140F3
and     eax, 0FFFFh
or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
mov     [ebp+var_4], eax
```

Runtime Process Without ASLR

0x00000000 – Start of memory

0x08049290 - 0x0805033c (R-X)

0x08050360 - 0x08051208 (R--)

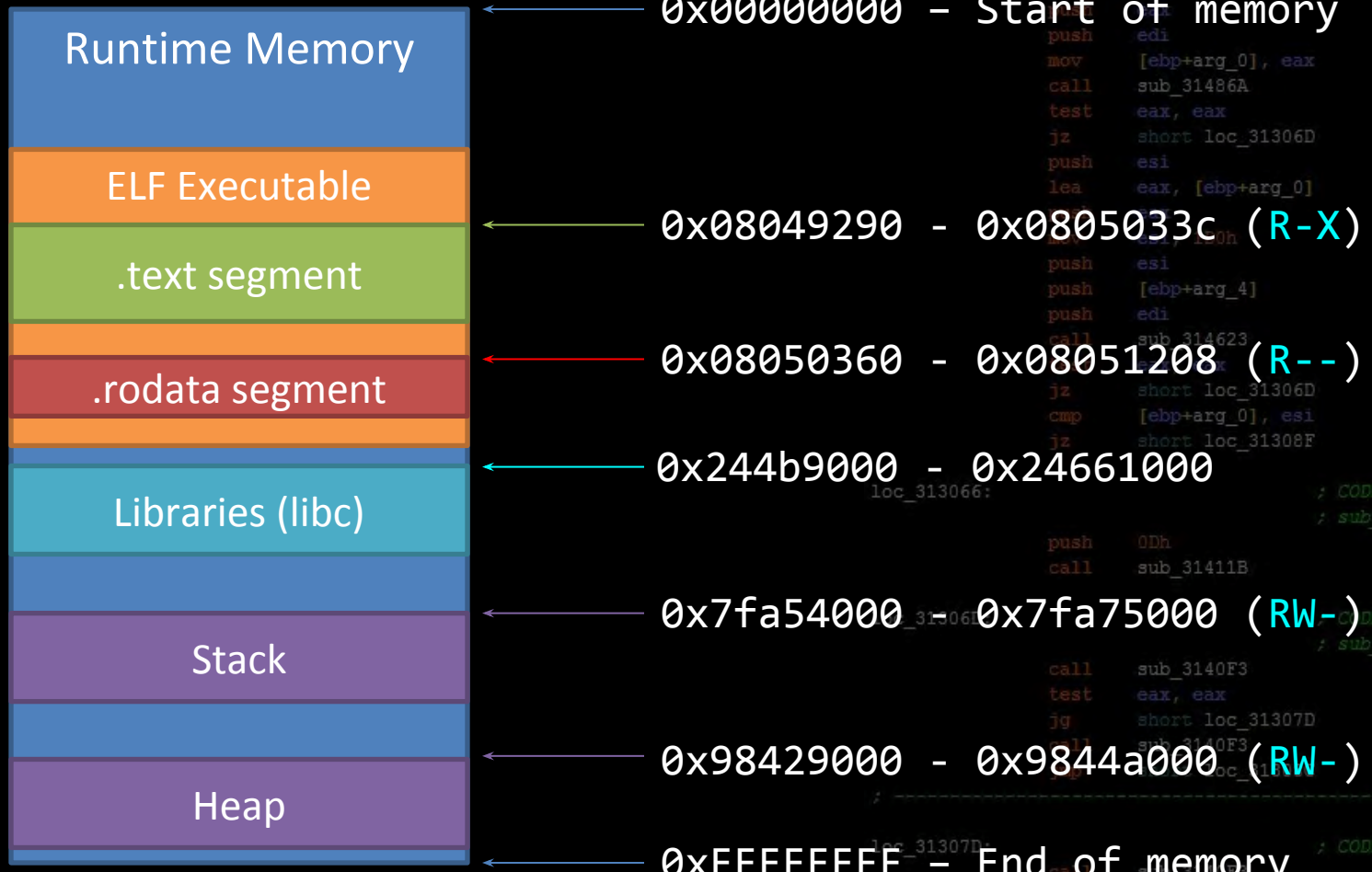
0x08055000 - 0x08076000 (RW-)

0xb7e25000 - 0xb7fcd000

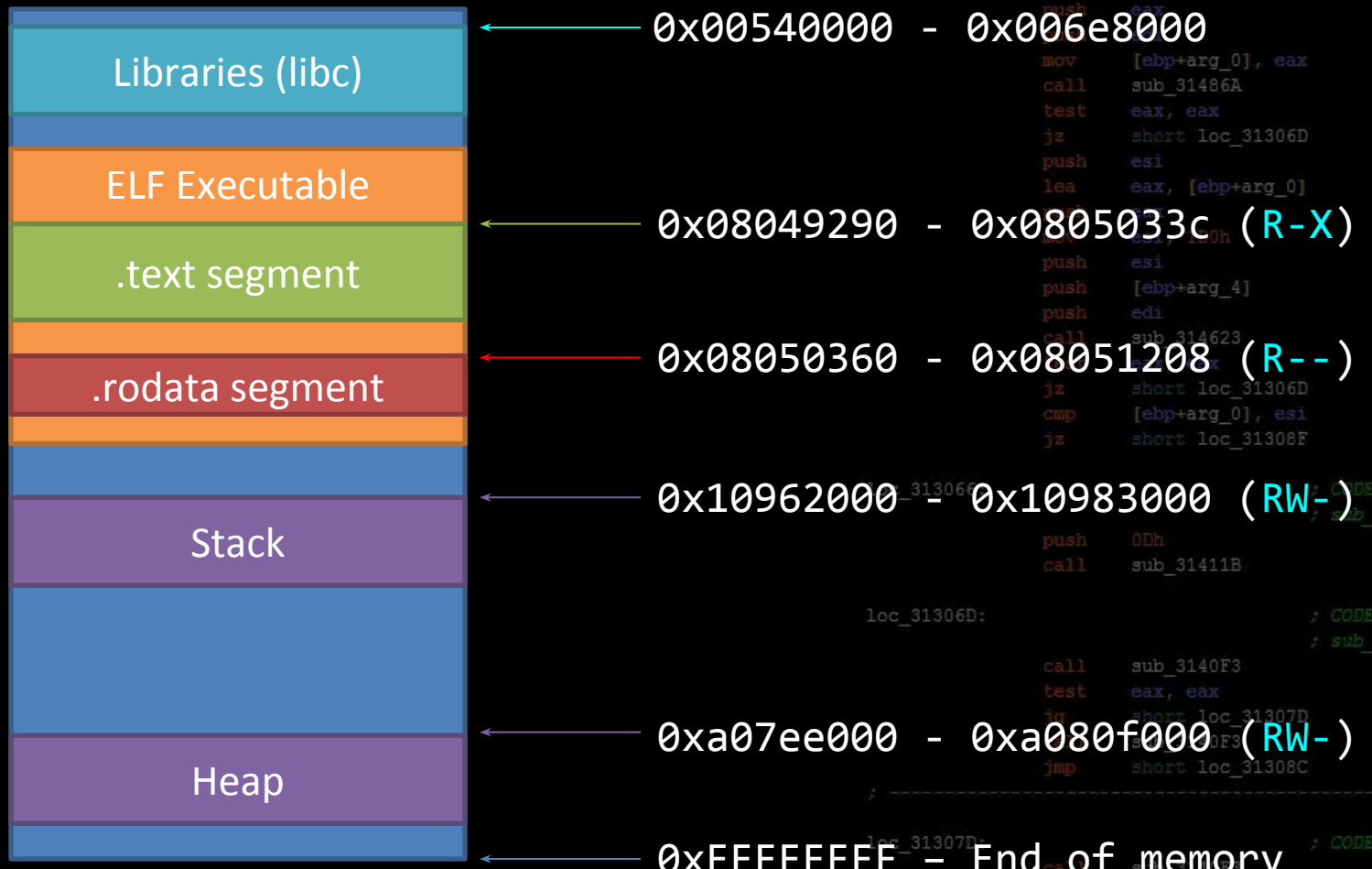
0xbffdf000 - 0xc0000000 (RW-)

0xFFFFFFFF – End of memory

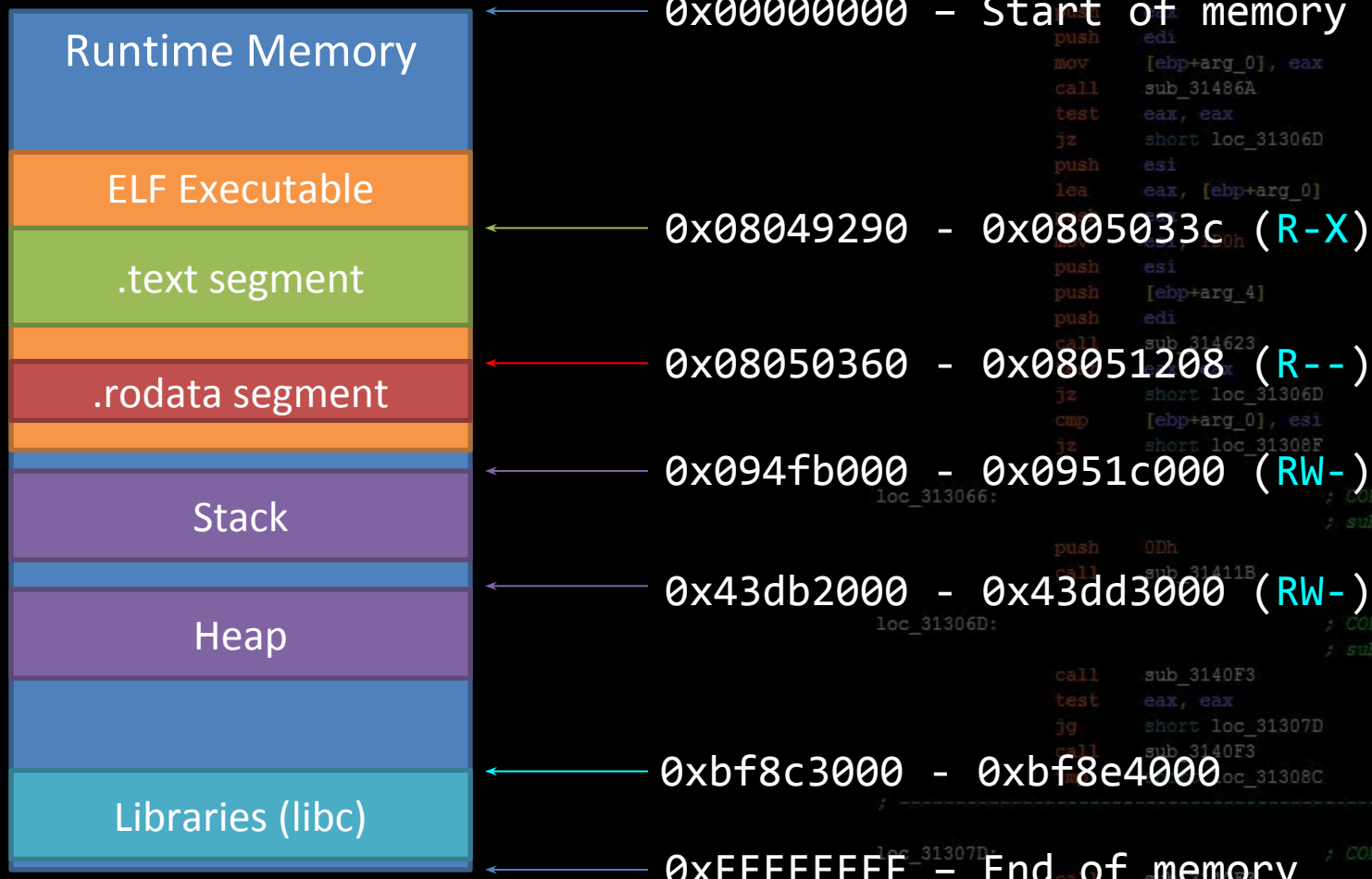
Run #1 With ASLR



Run #2 With ASLR



Run #3 With ASLR



ASLR in Action

> Open up a terminal.

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+59
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

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 :)

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+5E
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

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

```
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], ebx
                jnz     short loc_313066
                mov     eax, [ebp+var_70]
                cmp     eax, [ebp+var_84]
                jb      short loc_313066
                sub     eax, [ebp+var_84]
                push    esi
                push    esi
                push    eax
                push    edi
                mov     [ebp+arg_0], eax
                call    sub_31486A
                test    eax, eax
                jz      short loc_31306D
                push    esi
                lea     eax, [ebp+arg_0]
                push    eax
                mov     esi, 1D0h
                push    esi
                push    [ebp+arg_4]
                push    edi
                call    sub_314623
                test    eax, eax
                jz      short loc_31306D
                cmp     [ebp+arg_0], esi
                jz      short loc_31308F

loc_313066:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+55
                push    0Dh
                call    sub_31411B

loc_31306D:                             ; CODE XREF: sub_312FD8
                                        ; sub_312FD8+49
                call    sub_3140F3
                test    eax, eax
                jg      short loc_31307D
                call    sub_3140F3
                jmp     short loc_31308C
; ---------------------------------------------------------------------------

loc_31307D:                             ; CODE XREF: sub_312FD8
                call    sub_3140F3
                and     eax, 0FFFFh
                or      eax, 80070000h

loc_31308C:                             ; CODE XREF: sub_312FD8
                mov     [ebp+var_4], eax
```

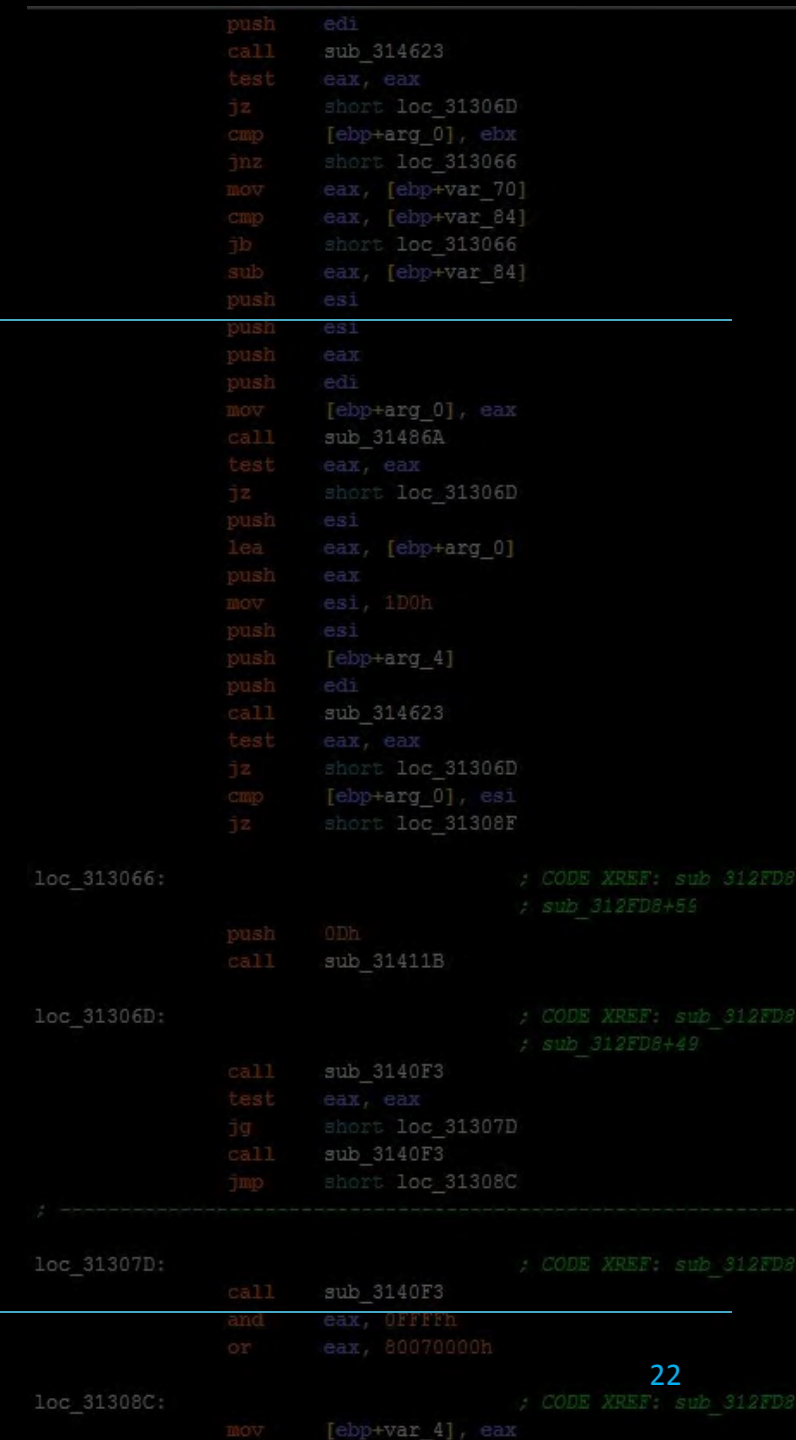
ASLR in Action

> Open up a terminal.

> Type “cat /proc/self/maps”

> Repeat a few times :)

- Stack Address Changes
- Heap Address Changes



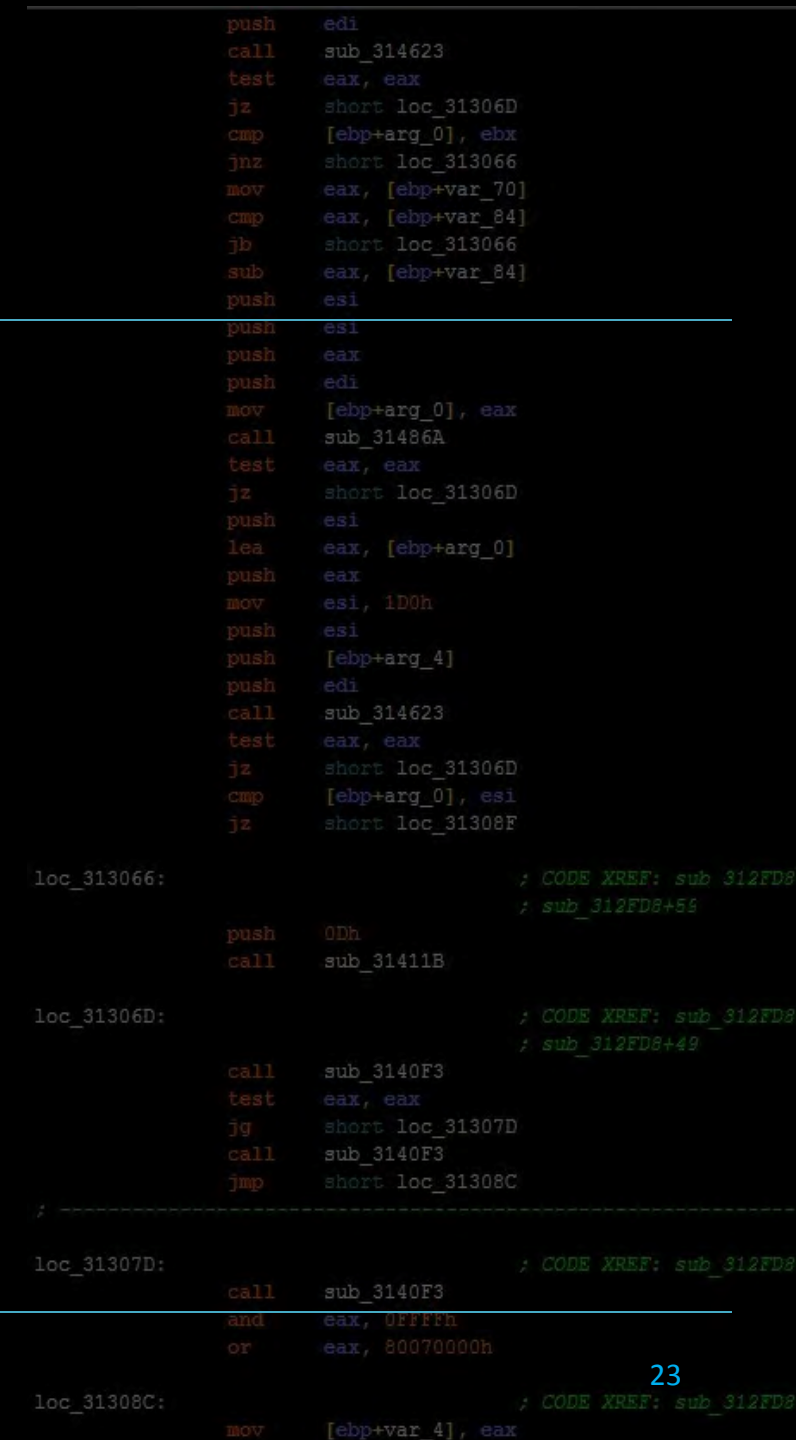
ASLR in Action

> Open up a terminal.

> Type “cat /proc/self/maps”

> Repeat a few times :)

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

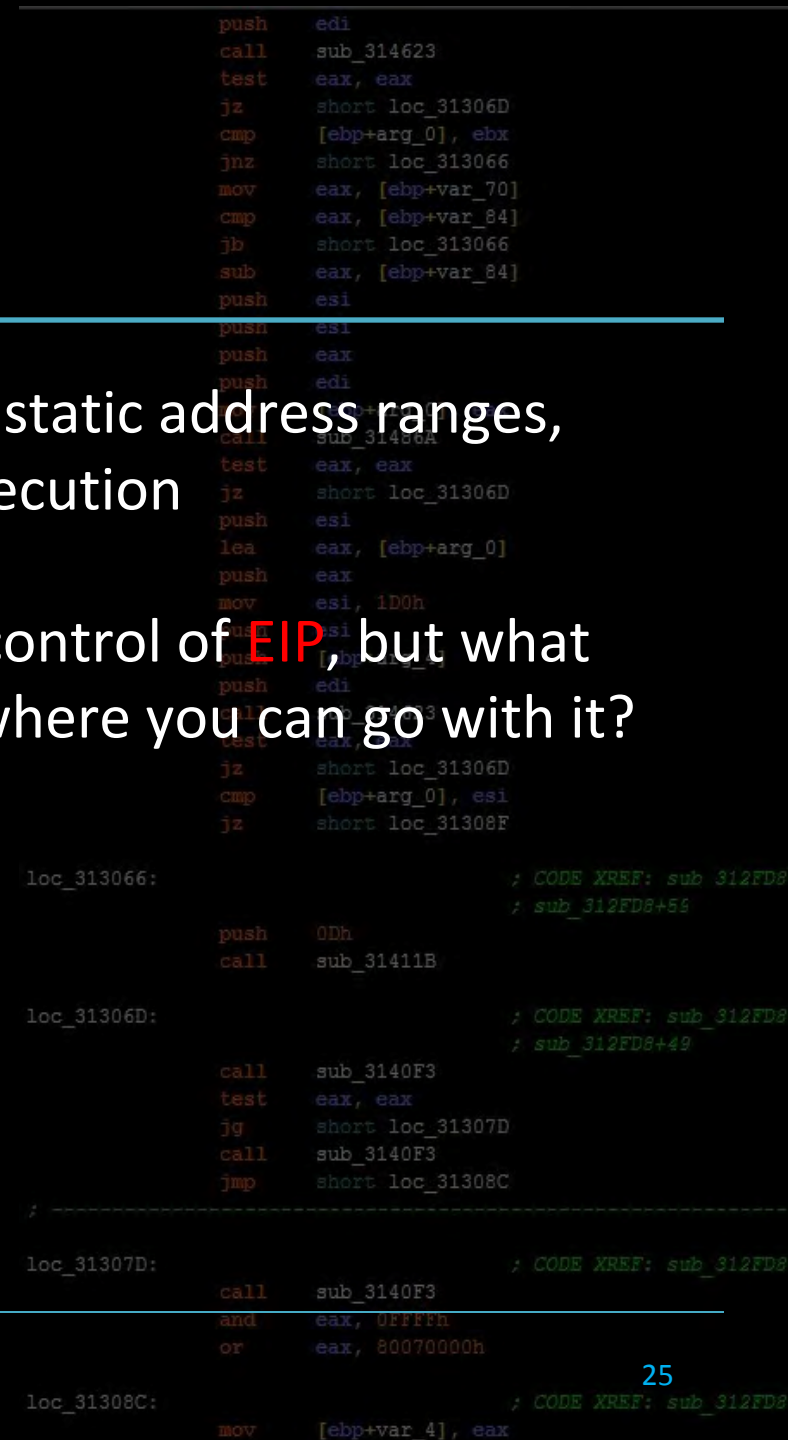


ASLR Basics

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

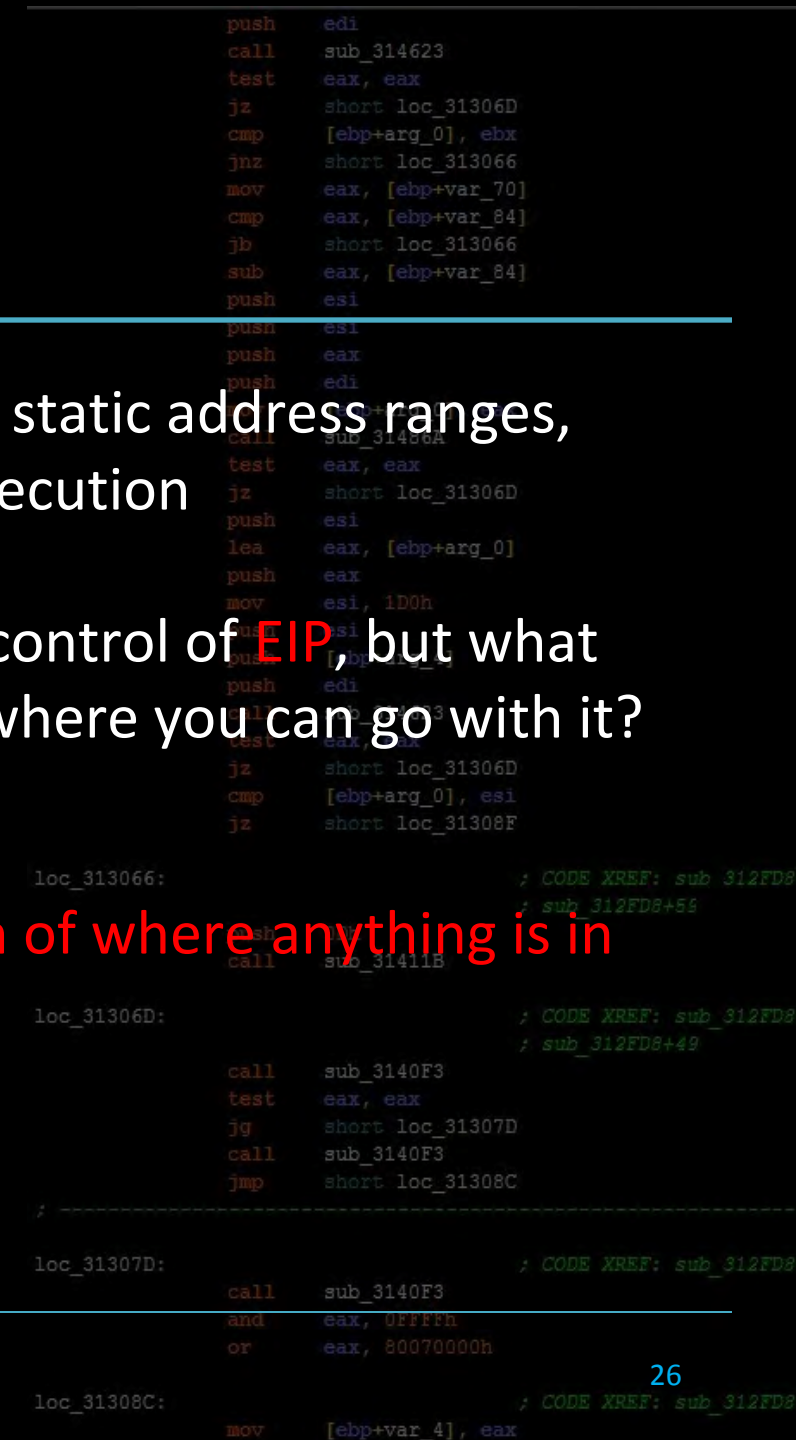
ASLR Basics

- 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 1st, 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)

History of ASLR

- When was ASLR implemented?
  - May 1st, 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)

perspective: markus is accepted to RPI

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

0x00000000 – Start of memory

0x08049290 - 0x0805033c (R-X)

0x08050360 - 0x08051208 (R--)

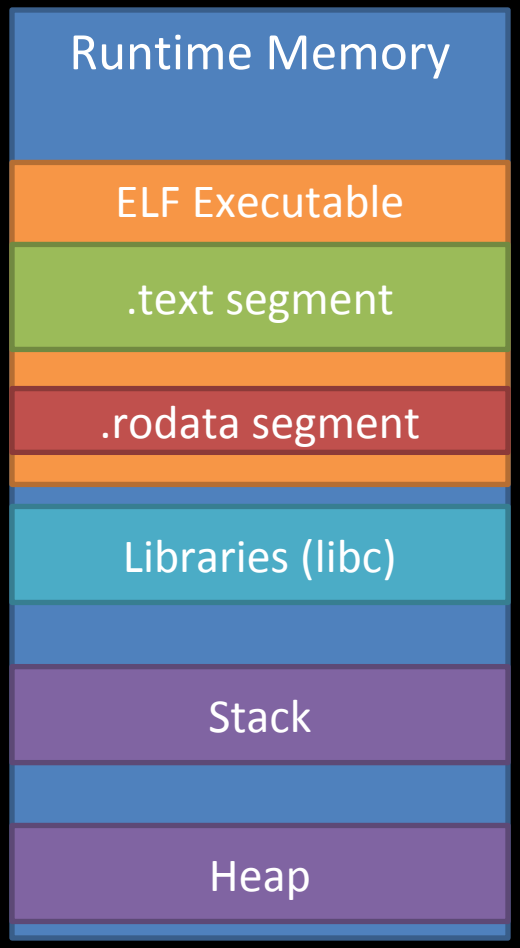
0x08055000 - 0x08076000 (RW-)

0xb7e25000 - 0xb7fcd000

0xbffdf000 - 0xc0000000 (RW-)

0xFFFFFFFF – End of memory

Run #1 With ASLR



0x00000000 – Start of memory

0x08049290 - 0x0805033c (R-X)

wat r u doin ELF

0x08050360 - 0x08051208 (R--)

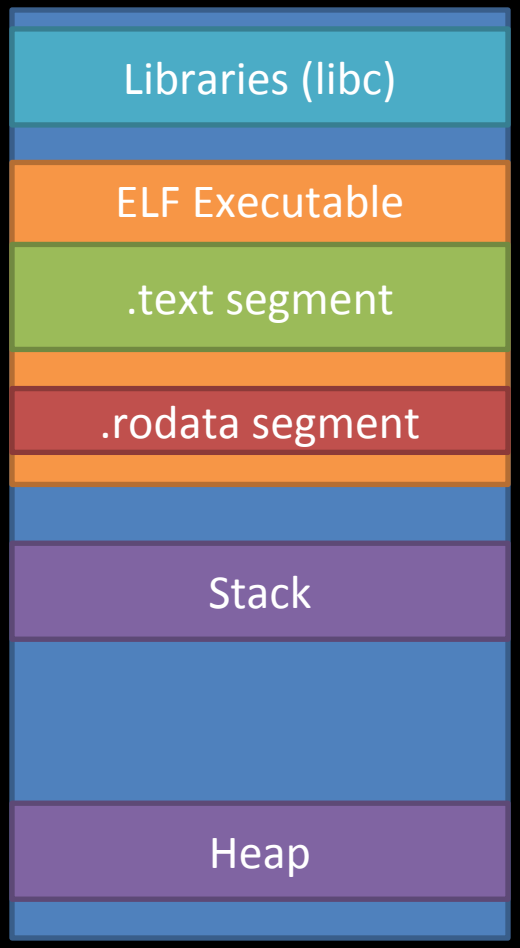
0x244b9000 - 0x24661000

0x7fa54000 - 0x7fa75000 (RW-)

0x98429000 - 0x9844a000 (RW-)

0xFFFFFFFF – End of memory

Run #2 With ASLR



0x00540000 - 0x006e8000

0x08049290 - 0x0805033c (R-X)

plz ELF...

0x08050360 - 0x08051208 (R--)

0x10962000 - 0x10983000 (RW-)

0xa07ee000 - 0xa080f000 (RW-)

0xFFFFFFFF – End of memory

Run #3 With ASLR

0x00000000 – Start of memory

0x08049290 - 0x0805033c (R-X)

stahppp q_q

0x08050360 - 0x08051208 (R--)

0x094fb000 - 0x0951c000 (RW-)

0x43db2000 - 0x43dd3000 (RW-)

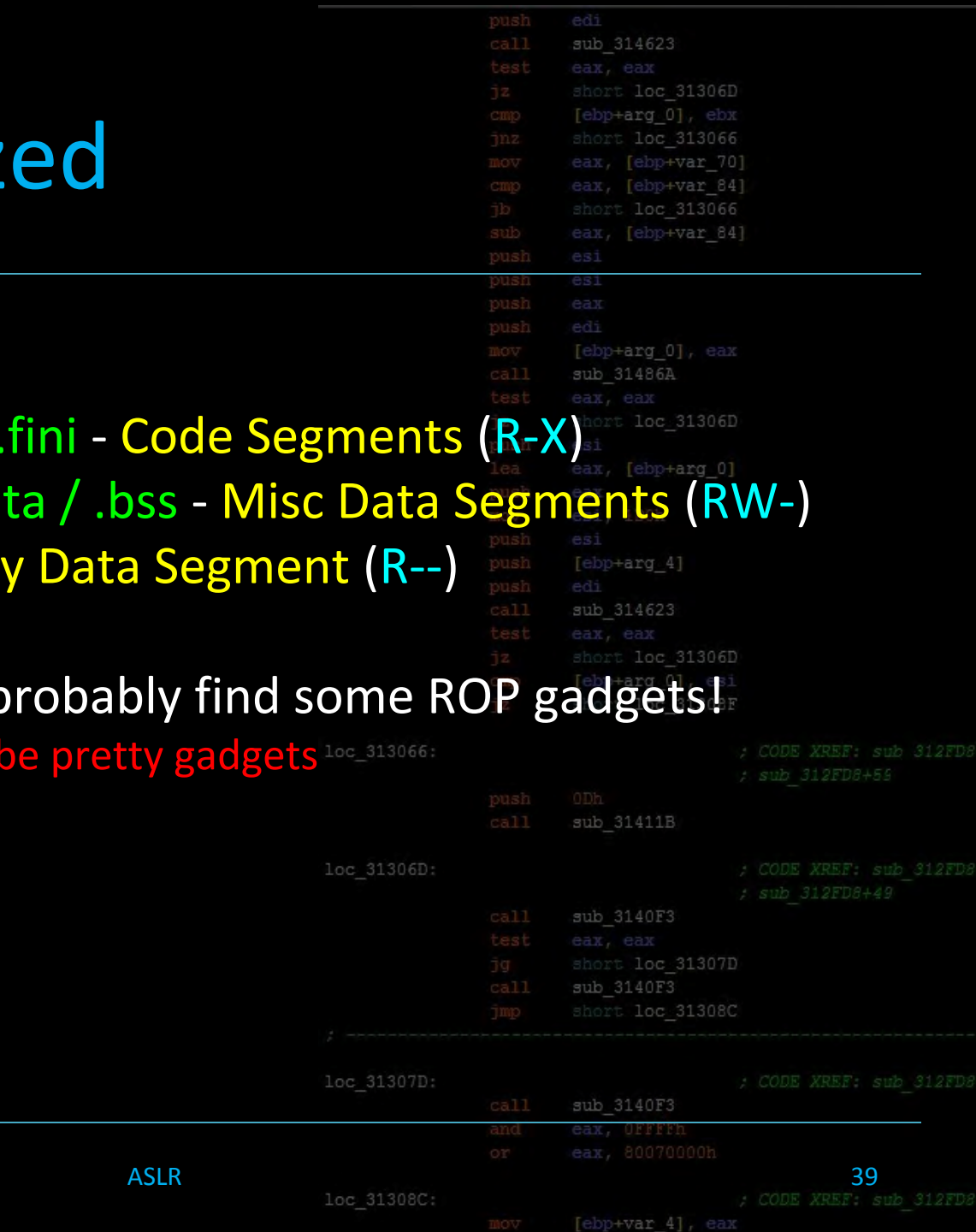
0xbf8c3000 - 0xbf8e4000

0xFFFFFFFF – End of memory

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 flag, you are not taking full advantage of ASLR

Checking for PIE

- Most binaries aren't actually compiled as PIE

```
doom@ubuntu:~$ checksec --file /bin/bash
RELRO           STACK CANARY      NX            PIE           RPATH      RUNPATH      FILE
Partial RELRO   Canary found      NX enabled    No PIE        No RPATH   No RUNPATH   /bin/bash
doom@ubuntu:~$ checksec --file /bin/ping
RELRO           STACK CANARY      NX            PIE           RPATH      RUNPATH      FILE
Partial RELRO   Canary found      NX enabled    No PIE        No RPATH   No RUNPATH   /bin/ping
doom@ubuntu:~$ checksec --file /usr/sbin/sshd
RELRO           STACK CANARY      NX            PIE           RPATH      RUNPATH      FILE
Full RELRO      Canary found      NX enabled    PIE enabled   No RPATH   No RUNPATH   /usr/sbin/sshd
doom@ubuntu:~$
doom@ubuntu:~$
```

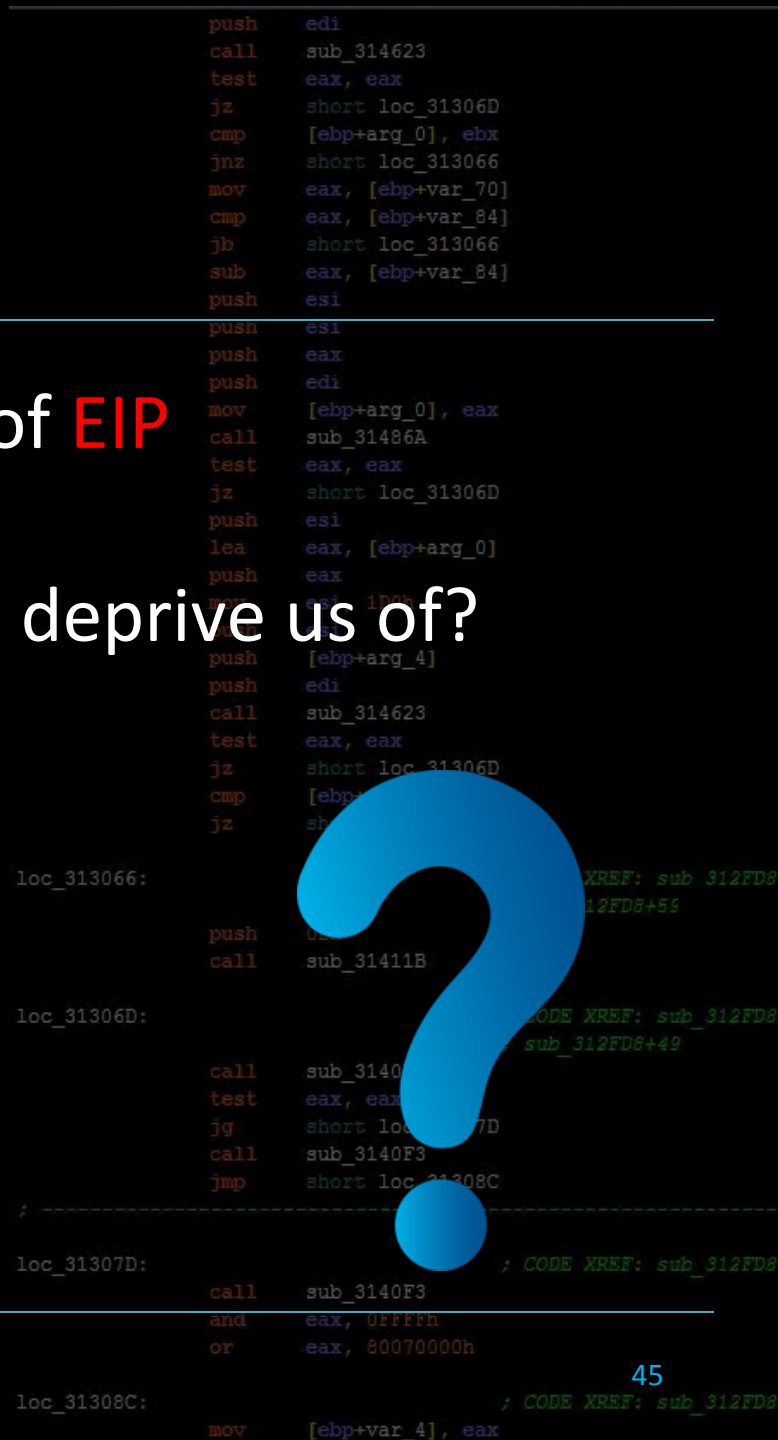
- Generally only on remote services, as you don't want your server to get owned

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?

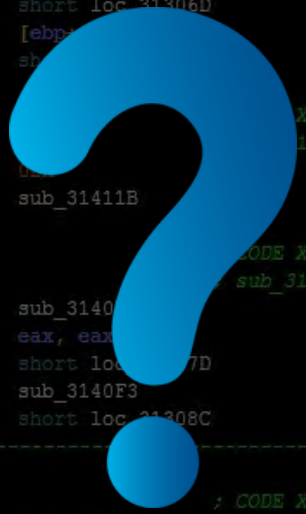


Bypassing ASLR

- Assume you can get control of EIP
- What information does ASLR deprive us of?
  - You don't know the address of ANYTHING

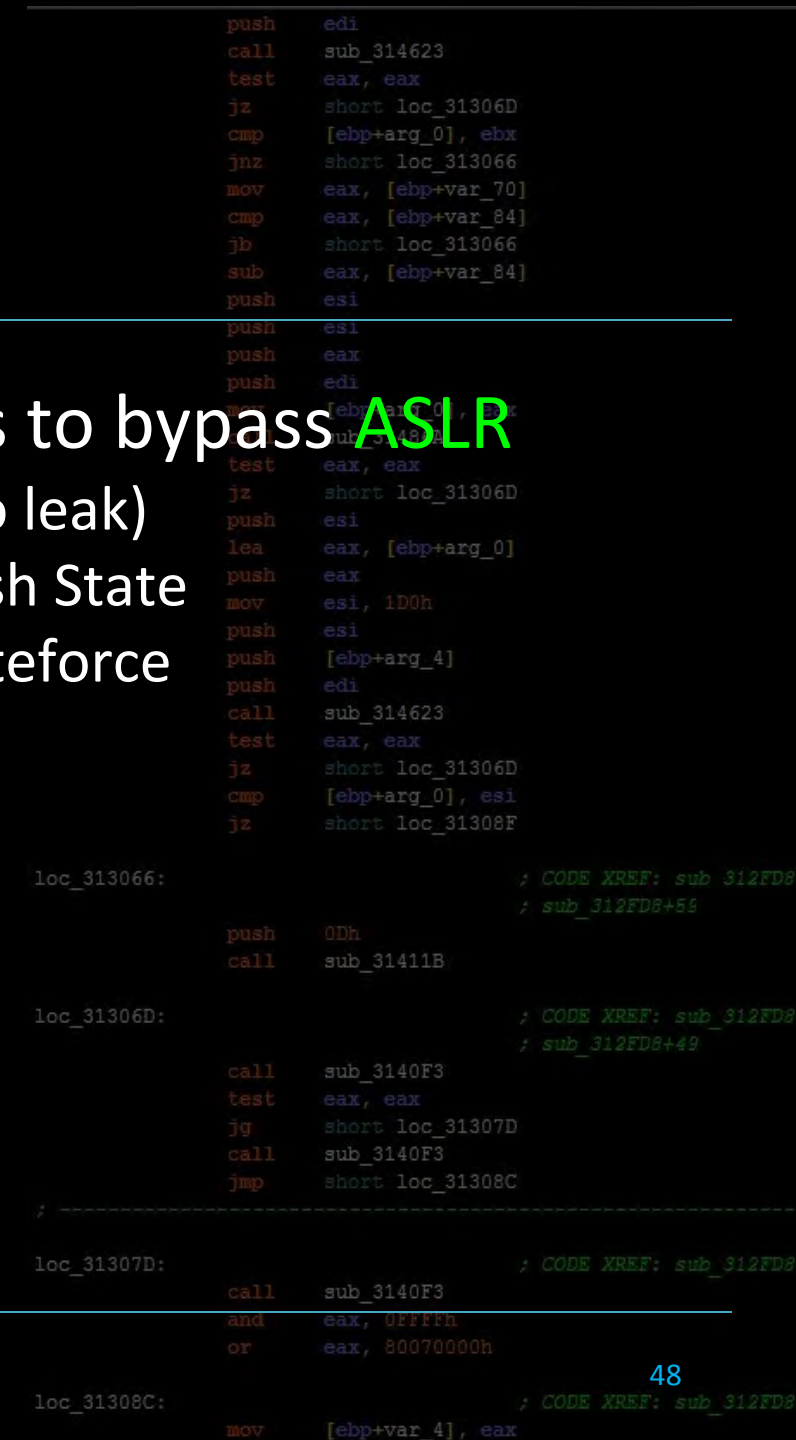
Bypassing ASLR

- 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

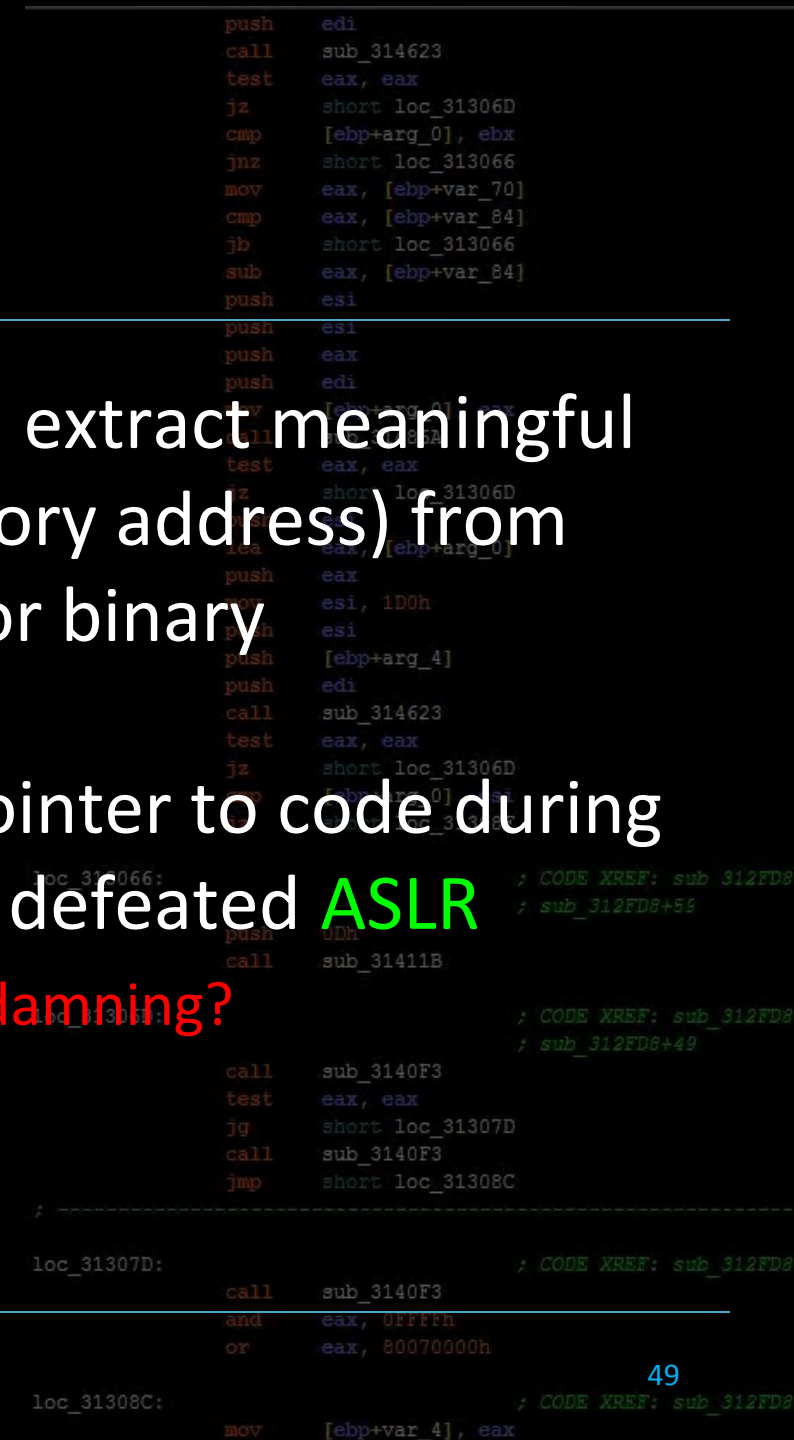
- There's a few common ways to bypass ASLR
  - Information disclosure (aka info leak)
  - Partial address overwrite + Crash State
  - Partial address overwrite + Bruteforce



What are Info Leaks?

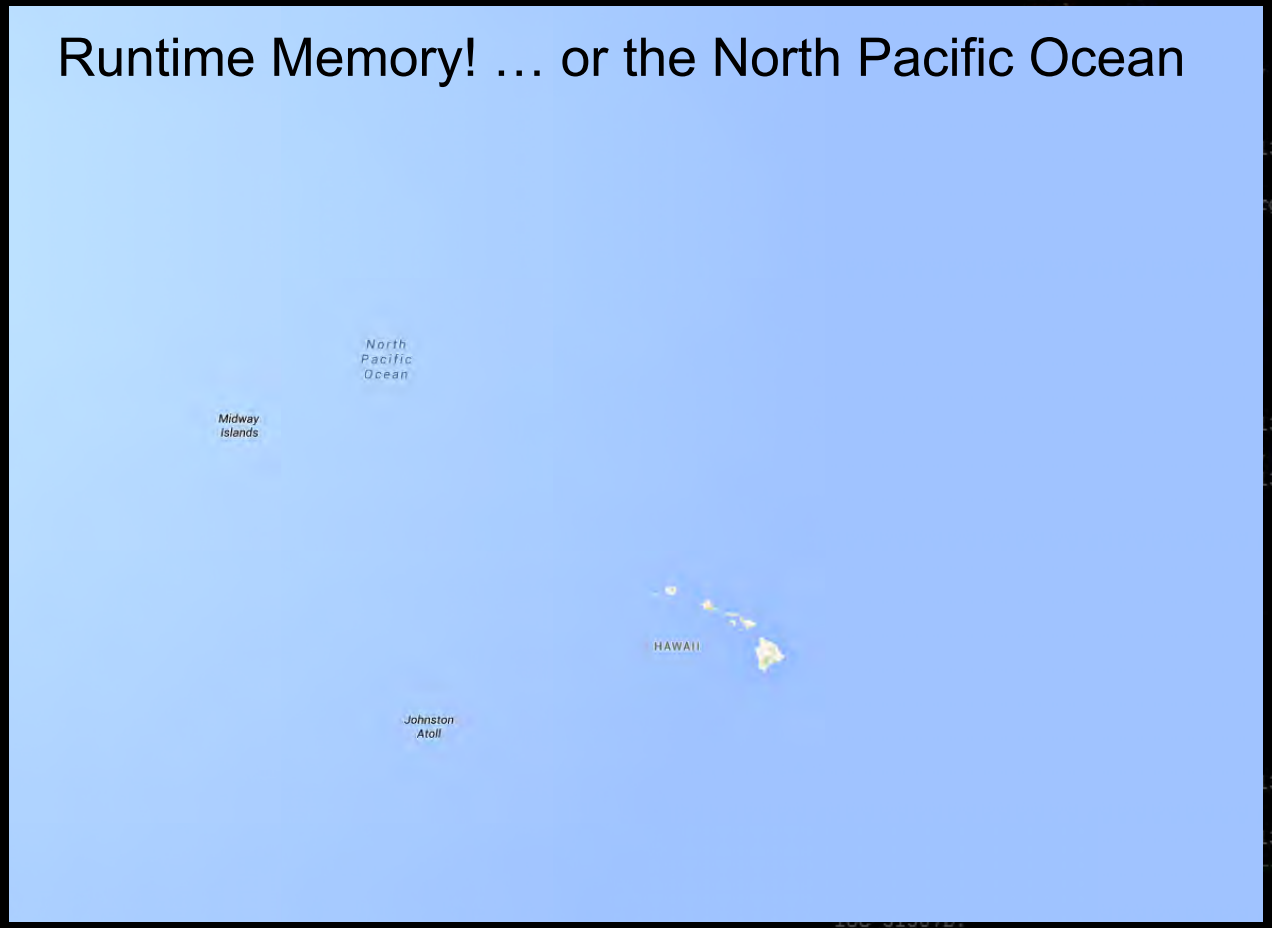
- An info leak is when you can extract meaningful information (such as a memory address) from the ASLR protected service or binary

- 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?



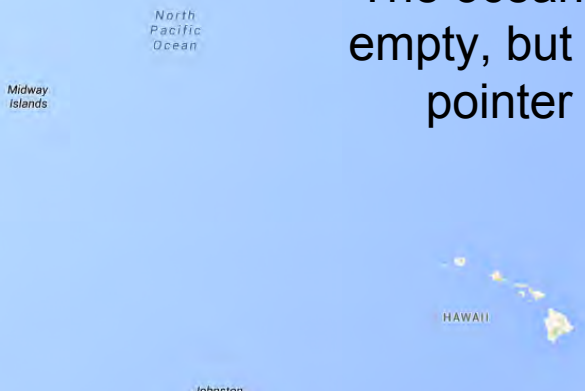
Death by Pointer

Runtime Memory! … or the North Pacific Ocean



Death by Pointer

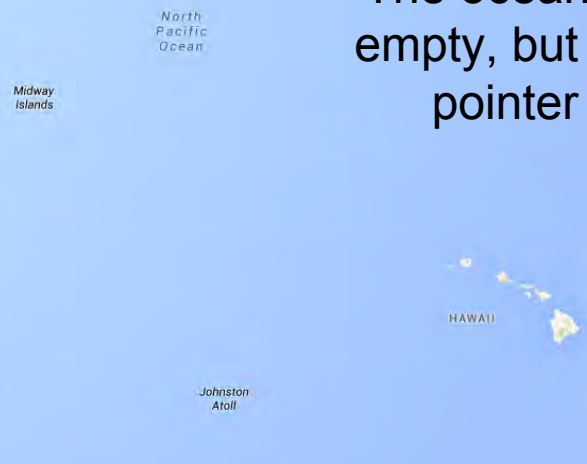
Runtime Memory! … or the North Pacific Ocean



The ocean is so vast and empty, but once you get a pointer to Hawaii...

Death by Pointer

Runtime Memory! … or the North Pacific Ocean



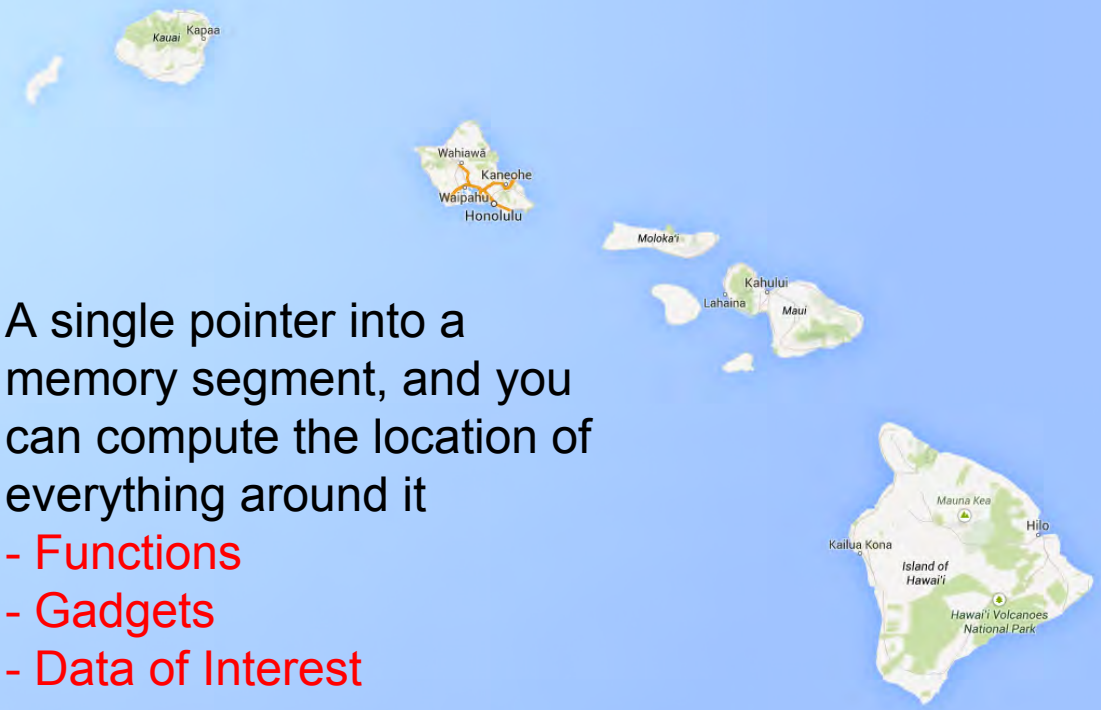
The ocean is so vast and empty, but once you get a pointer to Hawaii...

executable code!

Death by Pointer

Death by Pointer

Everything becomes relative



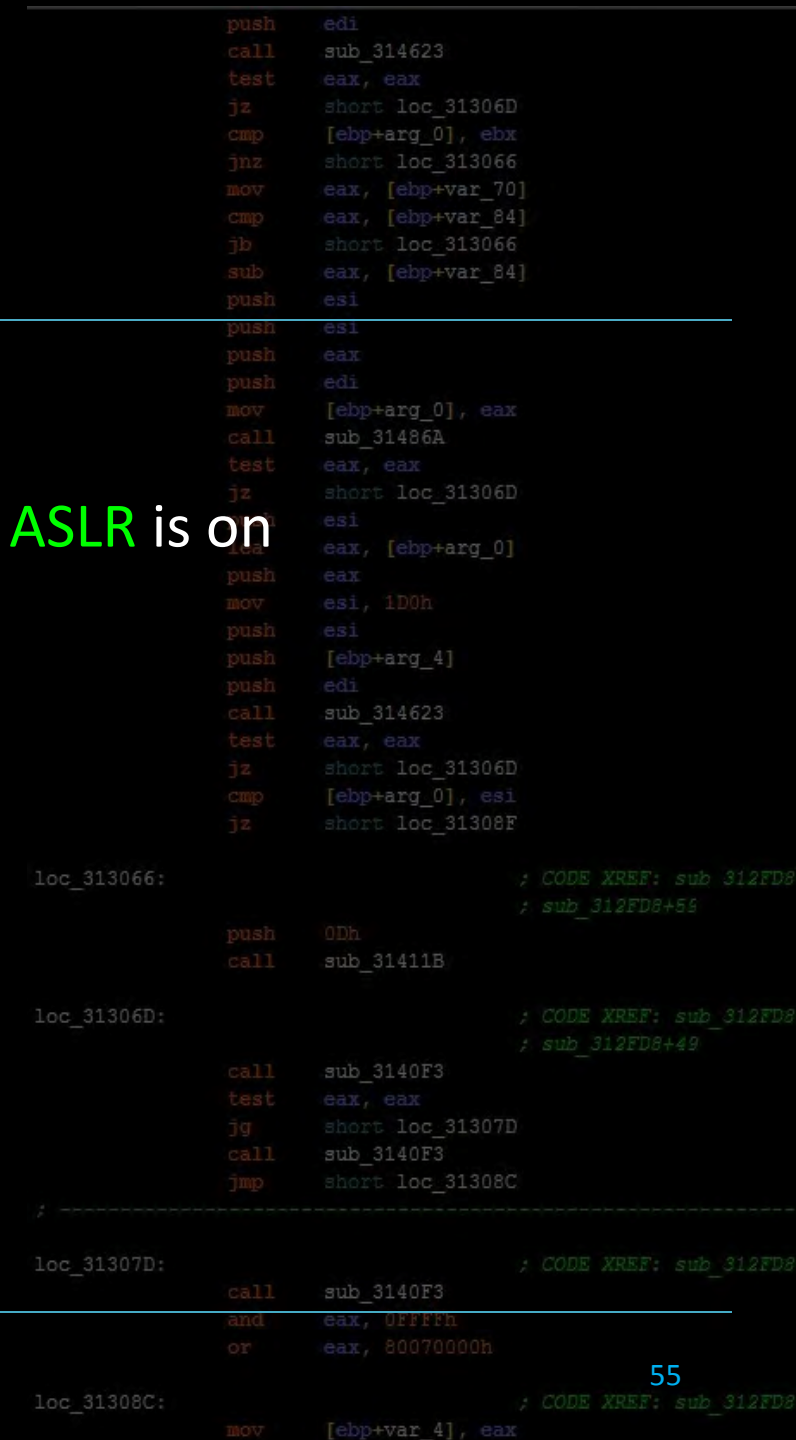
A single pointer into a memory segment, and you can compute the location of everything around it

- Functions
- Gadgets
- Data of Interest

Using Info Leaks

By Example:

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



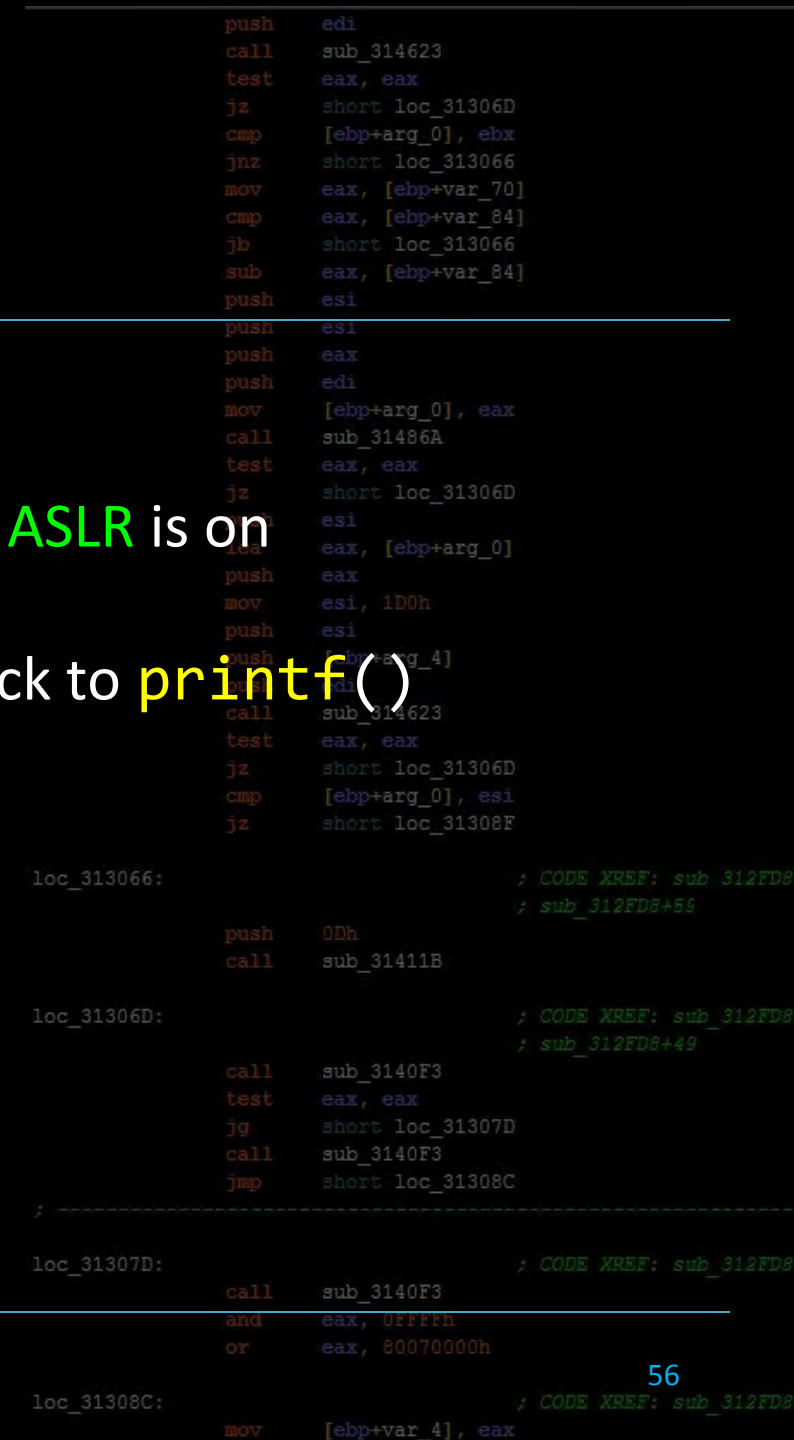
Using Info Leaks

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)
```

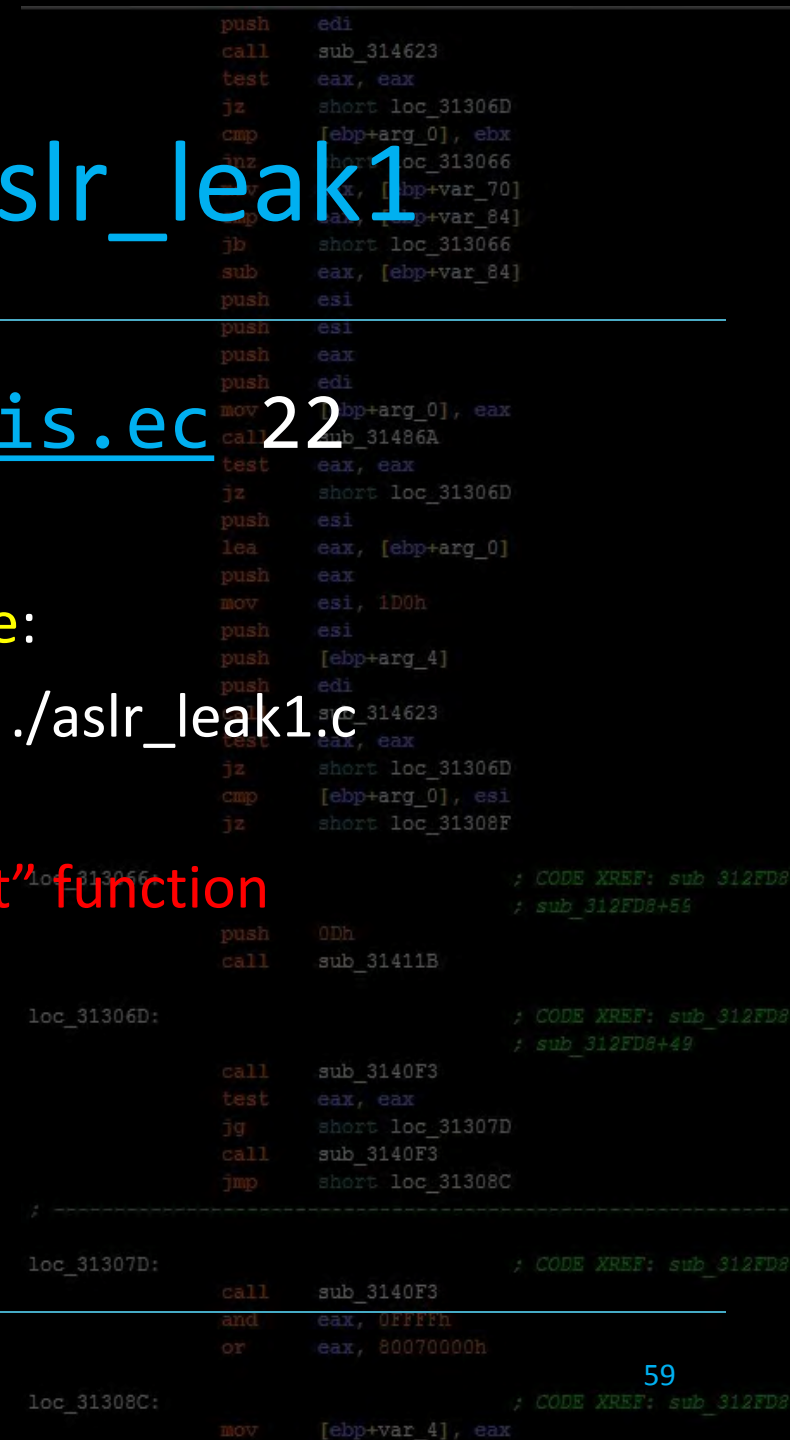
/levels/lecture/aslr/aslr_leak1

ssh lecture@warzone.rpis.ec 22

Fully Position Independent Executable:

gcc -pie -fPIE -fno-stack-protector ./aslr_leak1.c

Force it to execute the “i_am_rly_leet” function

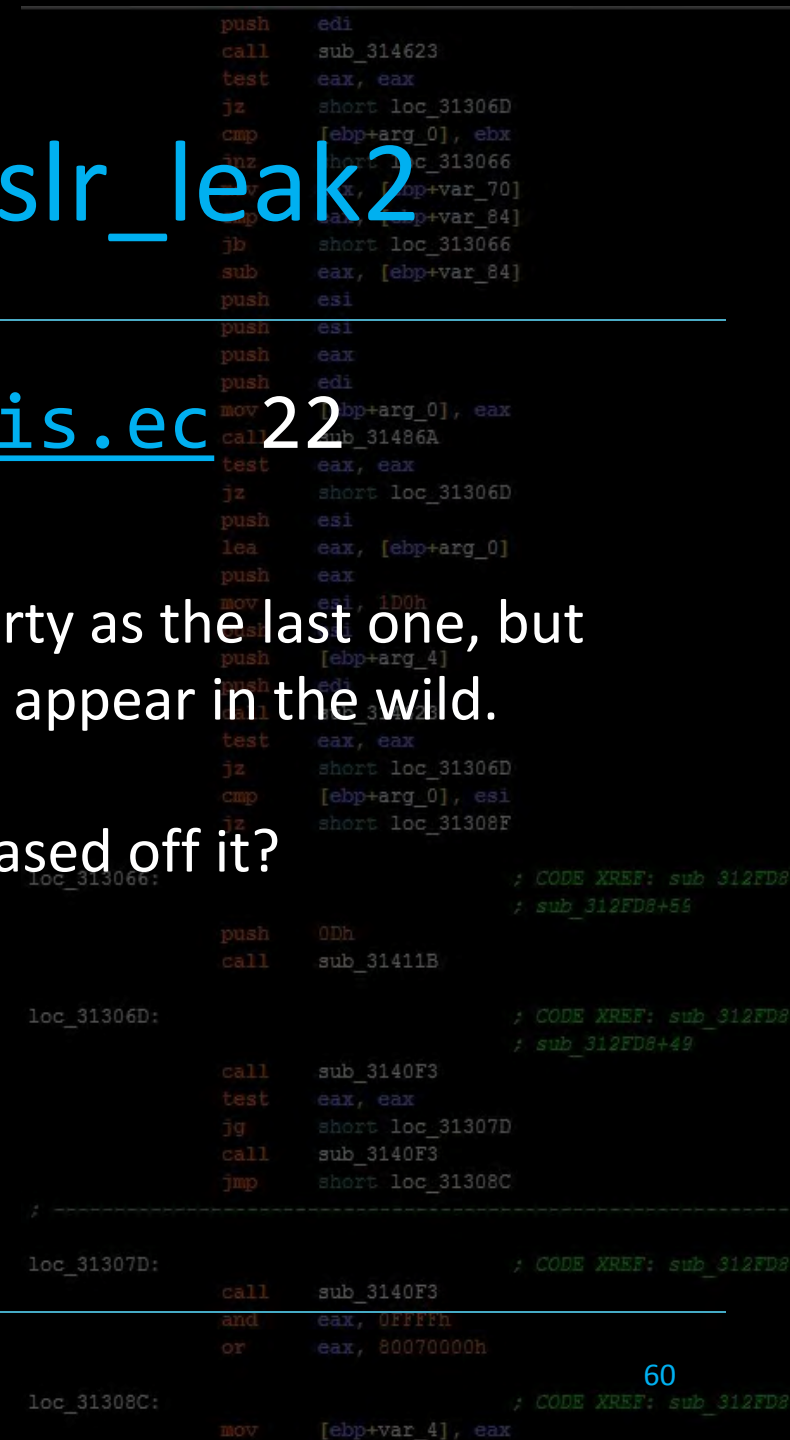


/levels/lecture/aslr/aslr_leak2

ssh lecture@warzone.rpis.ec 22

The exercise is equally as small and dirty as the last one, but this is typically how an infoleak might appear in the wild.

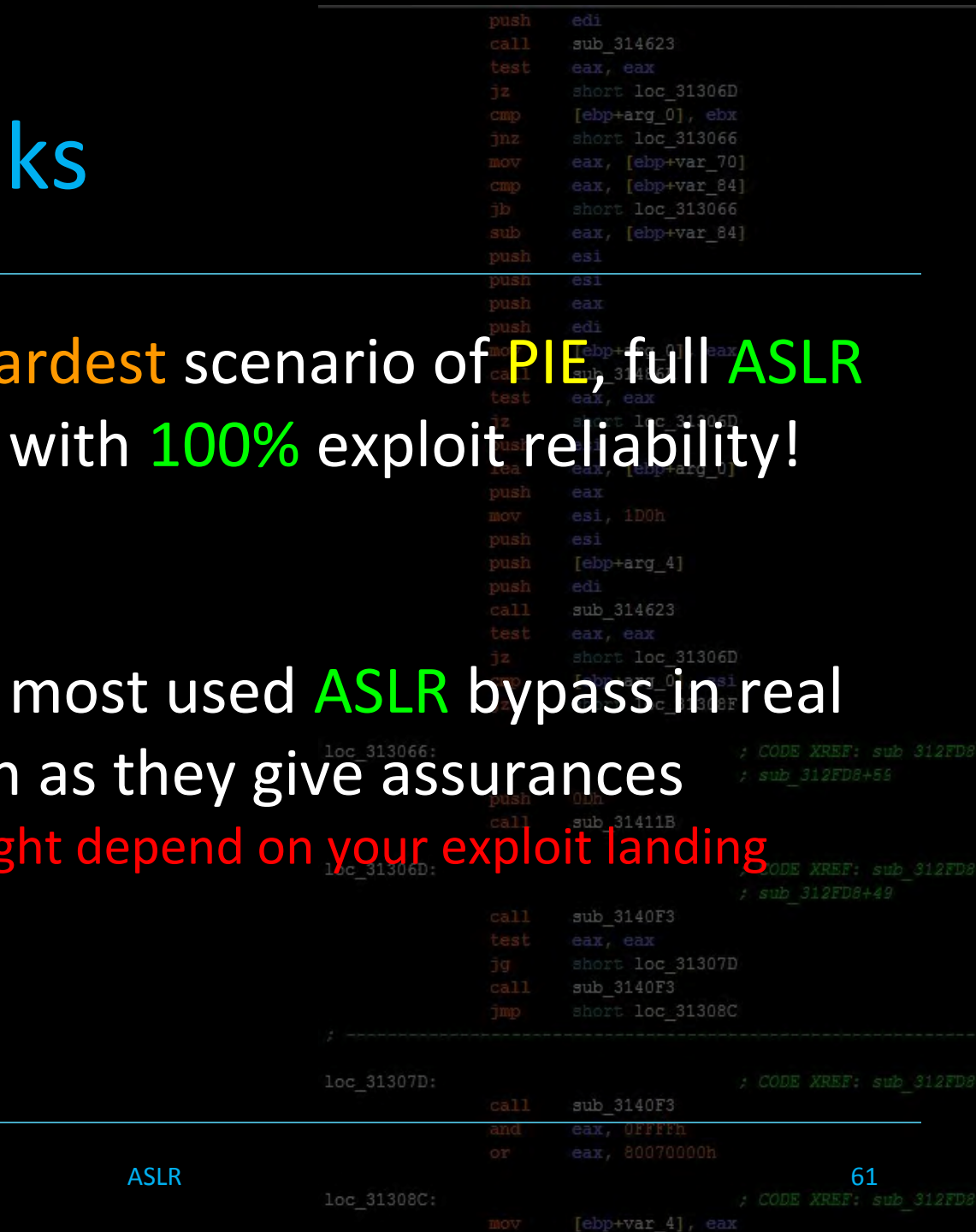
Can you parse it? Build a ROP chain based off it?



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
  - Someone’s life might depend on your exploit landing



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

2^4 bits of bruteforce gives you 64kb of range around the addr

2^{12} 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:

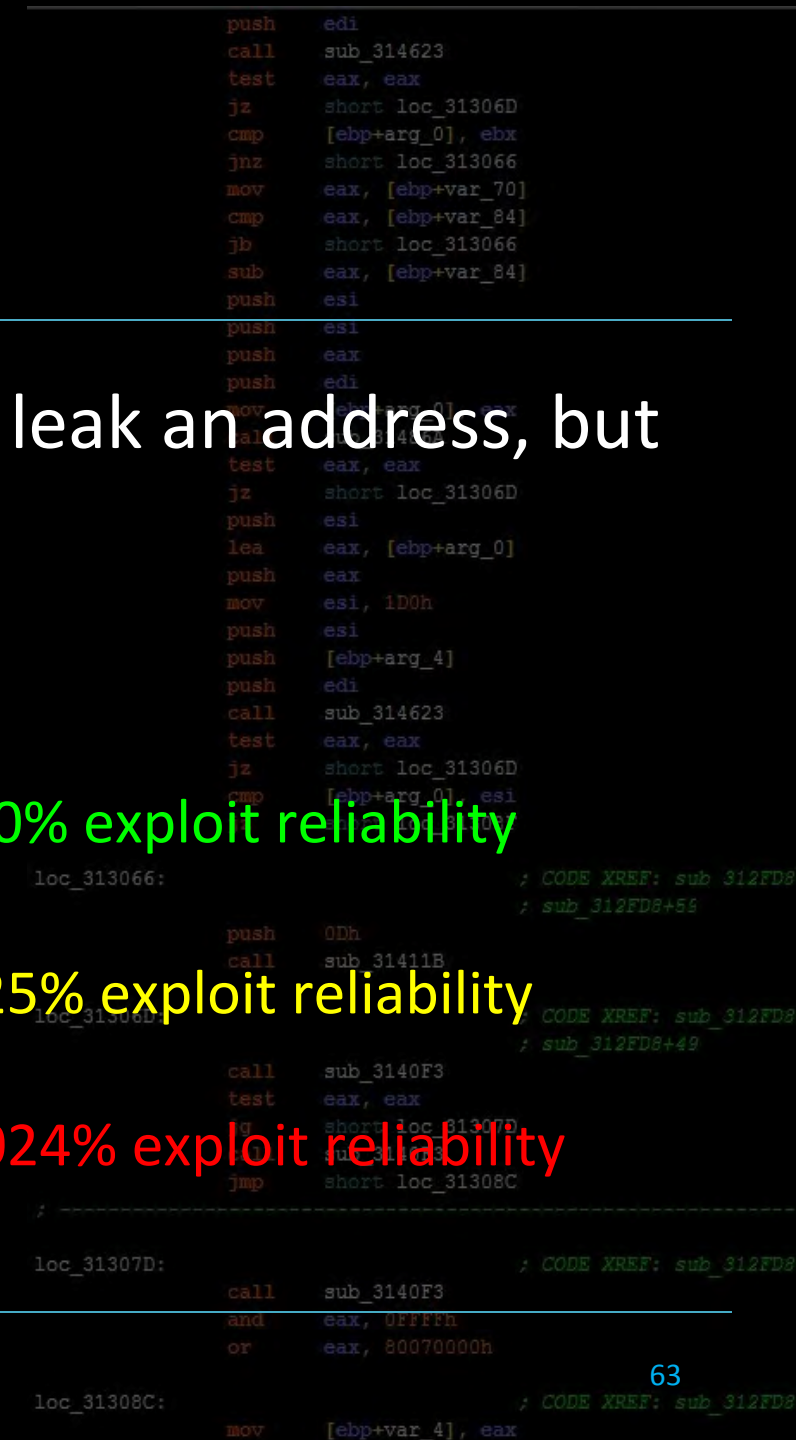
  0xb756b132 — 100% exploit reliability

  0xb758e132

  0xb75e5132 — 6.25% exploit reliability

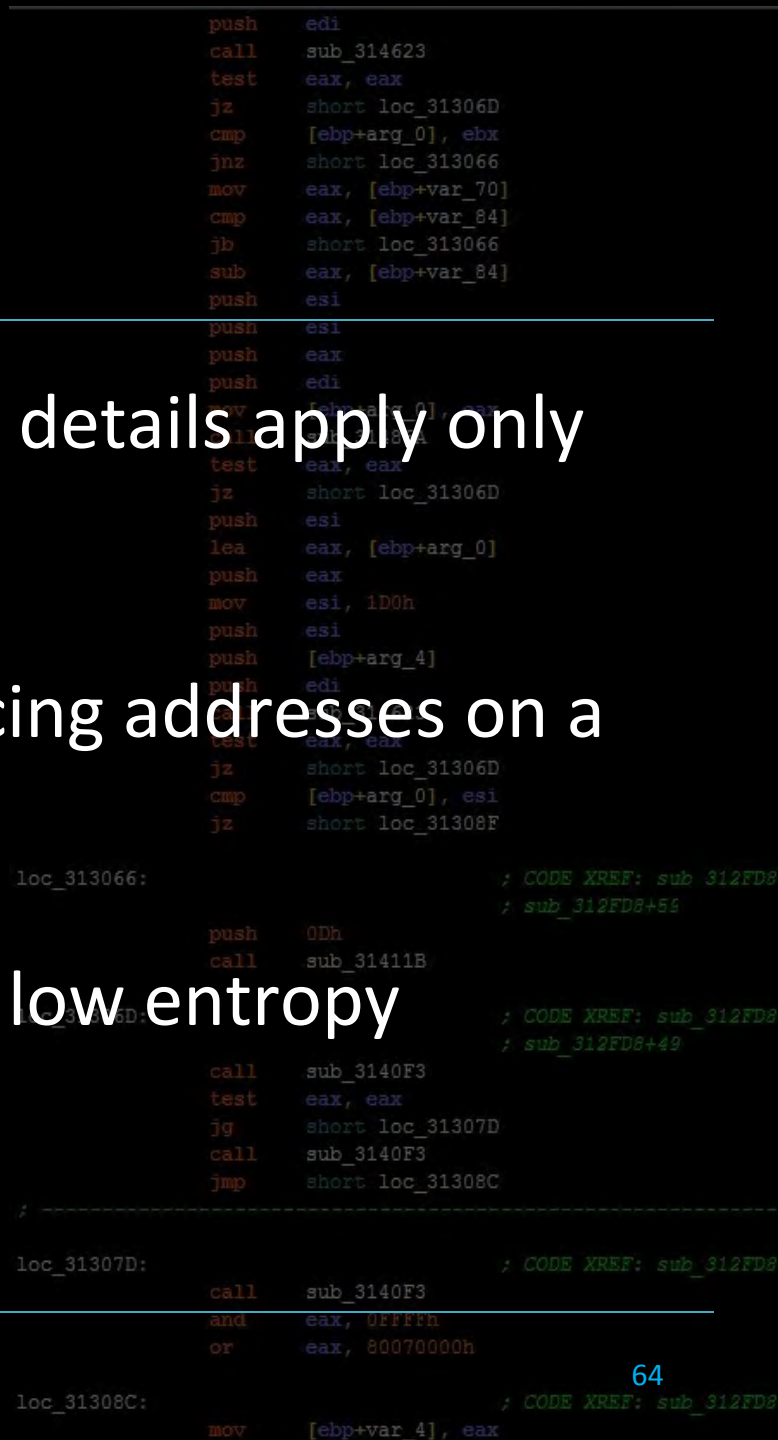
  0xb754d132

  0xb75cf132 — 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?

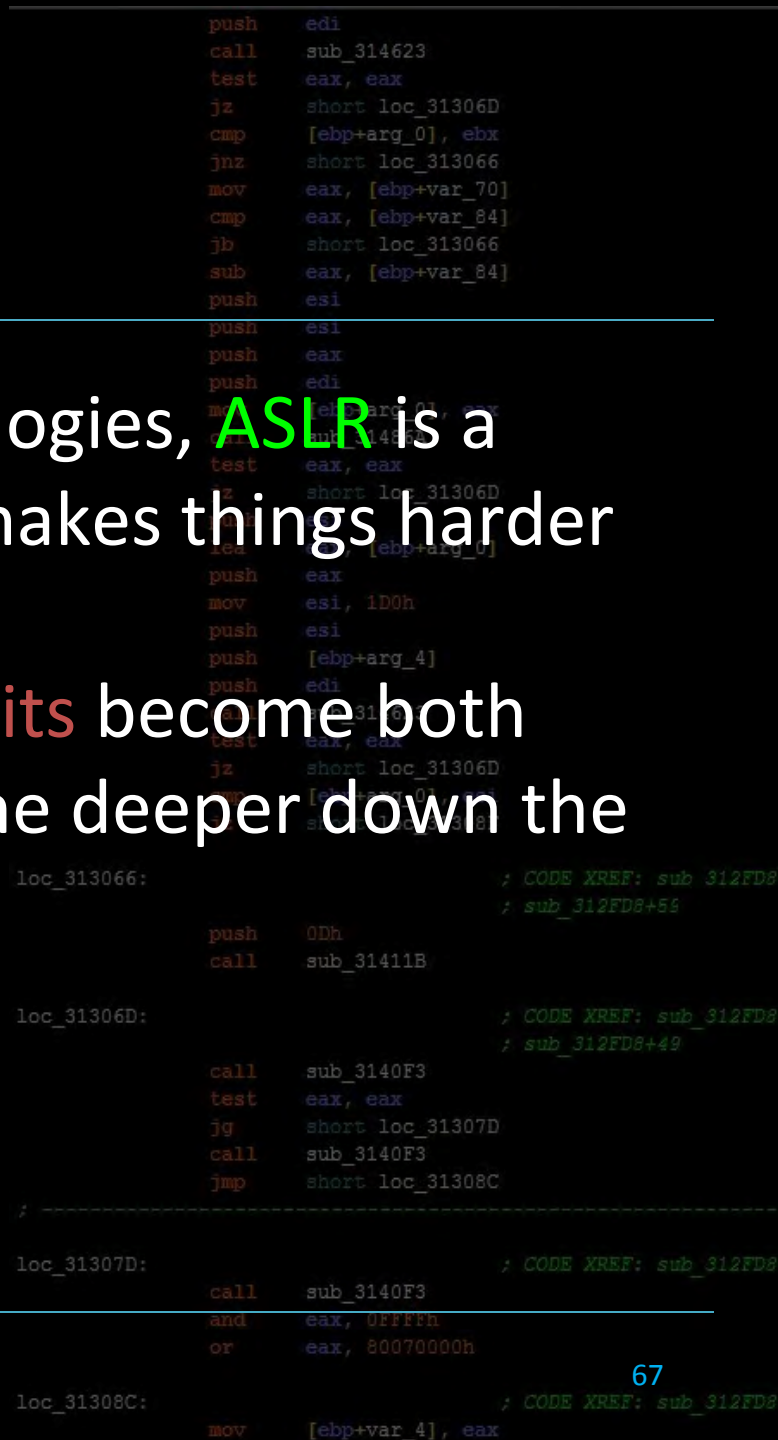
- Even if you can't easily leak some data address out of a register or off the stack, there's nothing that's stopping you from using it for stuff
  - As always: get creative

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