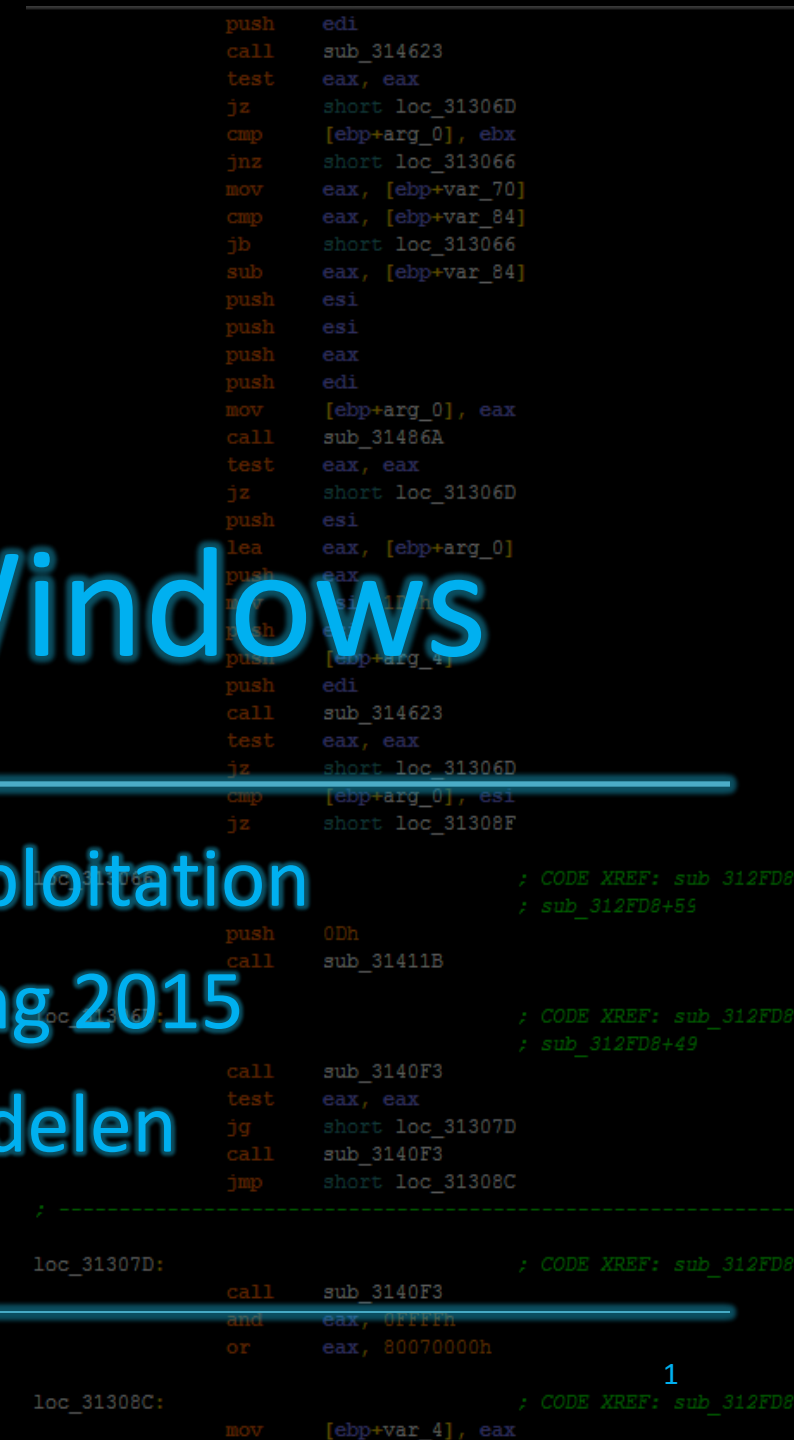
x64, ARM, Windows

Modern Binary Exploitation

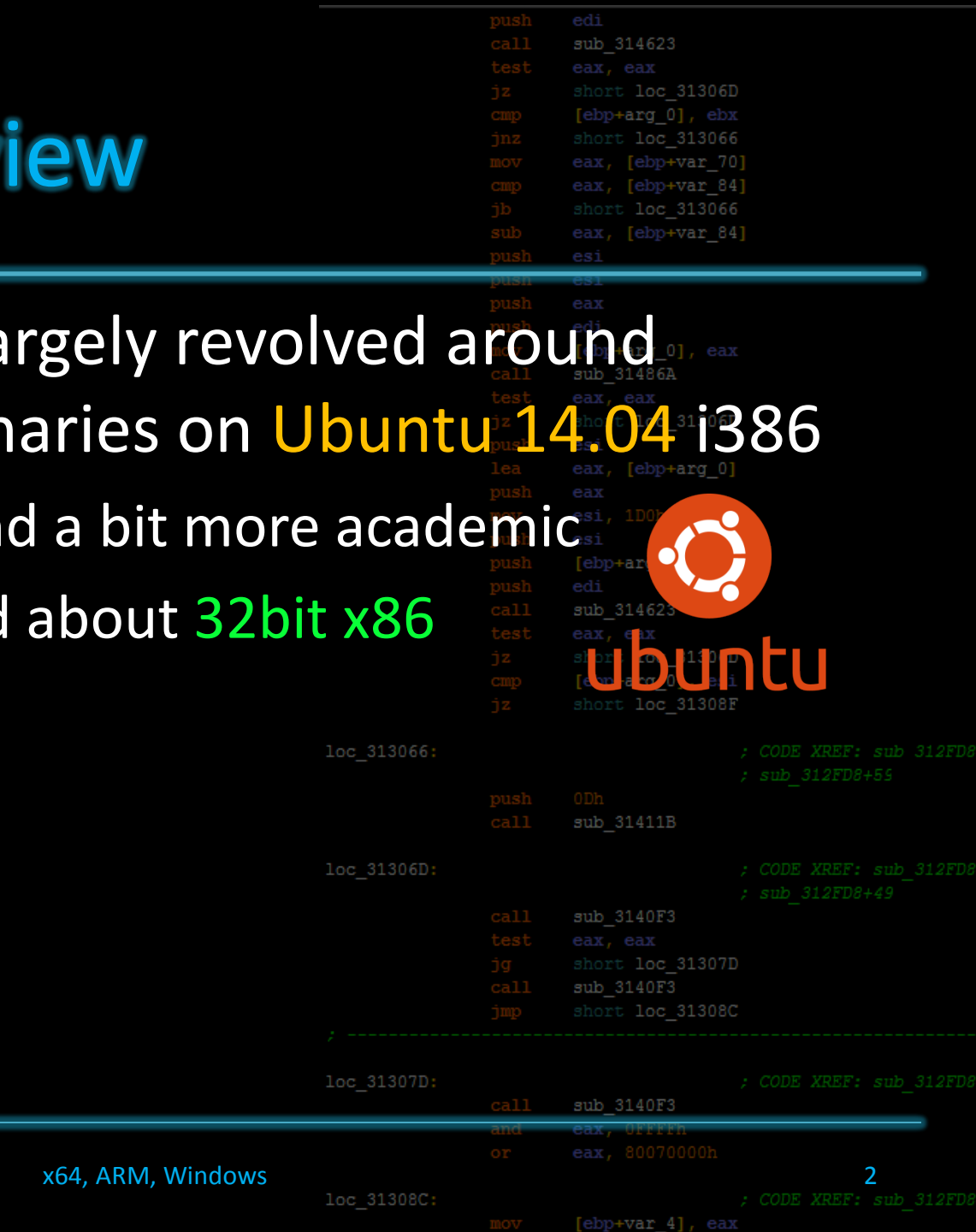
CSCI 4968 - Spring 2015

Markus Gaasedelen



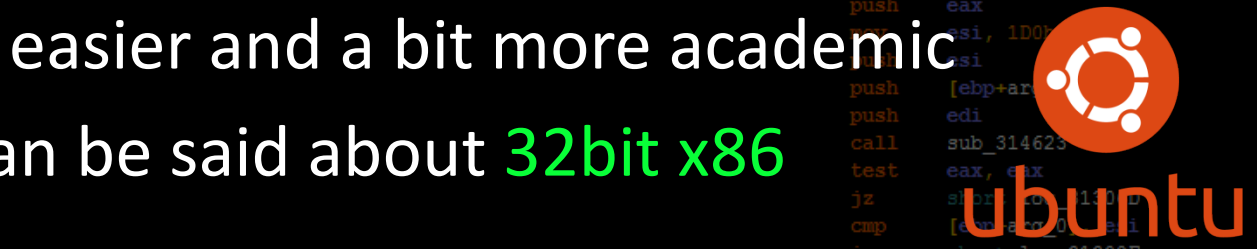
Lecture Overview

- This course has largely revolved around exploiting x86 binaries on Ubuntu 14.04 i386
  - Linux is easier and a bit more academic
  - Same can be said about 32bit x86



Lecture Overview

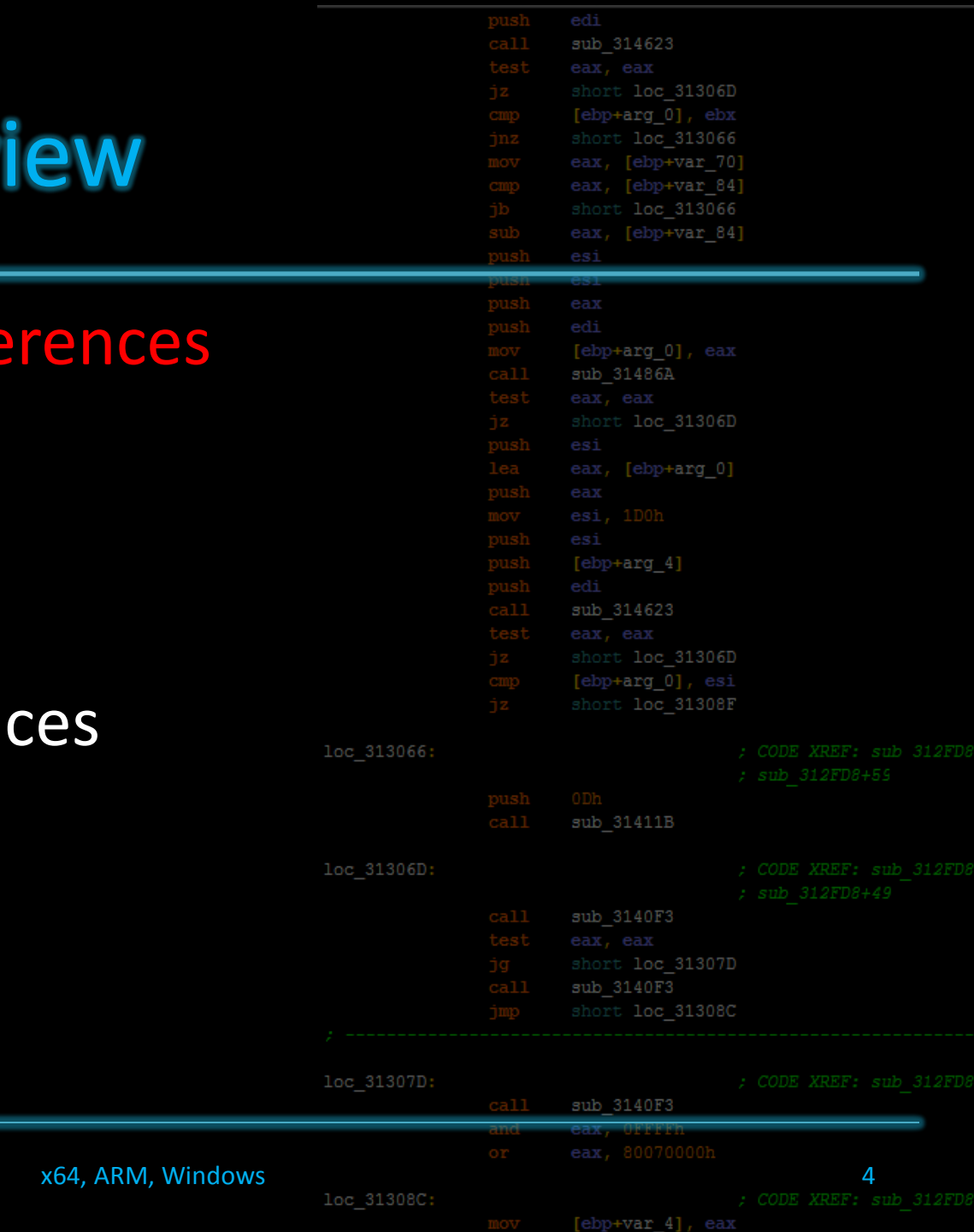
- This course has largely revolved around exploiting x86 binaries on Ubuntu 14.04 i386
  - Linux is easier and a bit more academic
  - Same can be said about 32bit x86



- But how does exploitation change for x86_64 systems? ARM devices? How about Windows?

Lecture Overview

- Architecture Differences
  - x86
  - x86_64
  - ARM
- Platform Differences
  - Windows



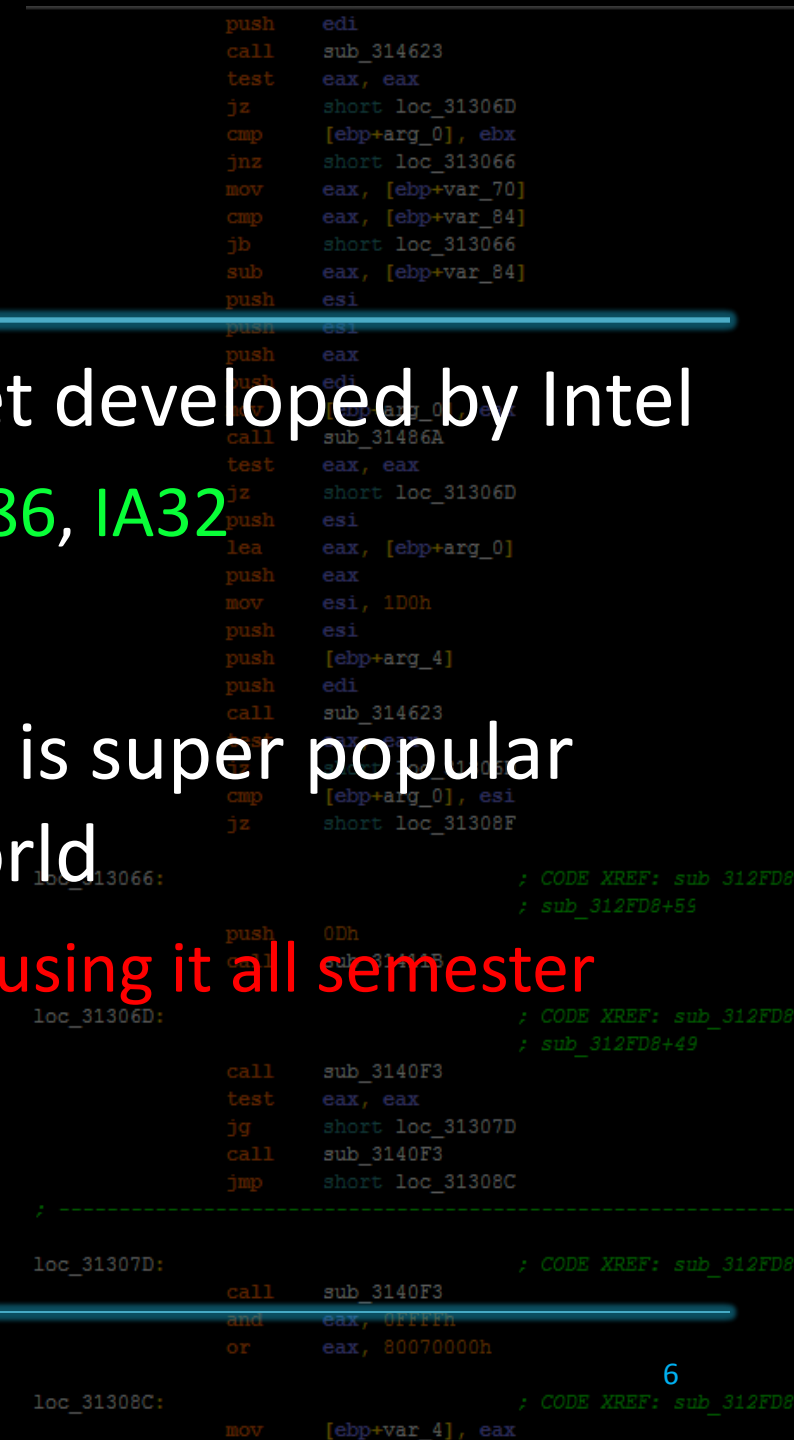
x86 Overview

- x86 is a 32bit instruction set developed by Intel
  - Sometimes known as x32, x86, IA32

x86 Overview

- x86 is a 32bit instruction set developed by Intel
  - Sometimes known as x32, x86, IA32

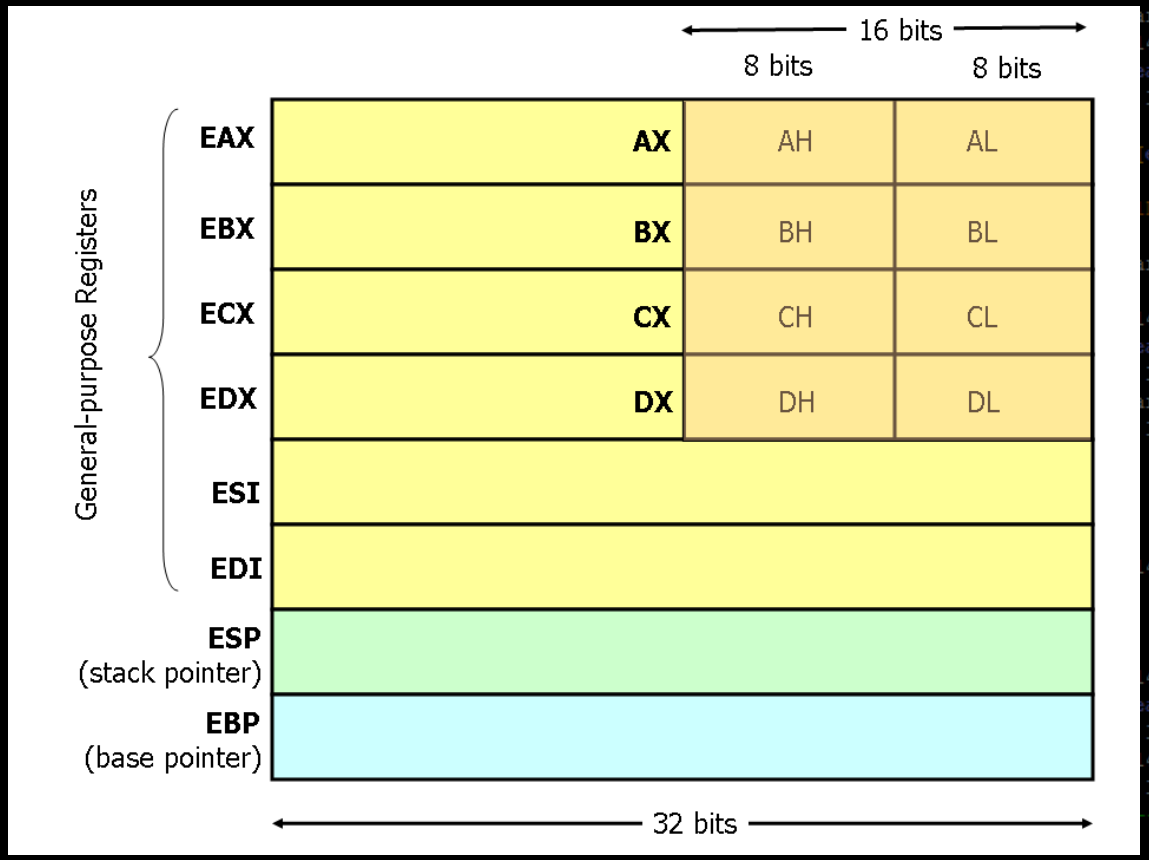
- It's a CISC architecture that is super popular and used all around the world
  - yadayadayada, you've been using it all semester



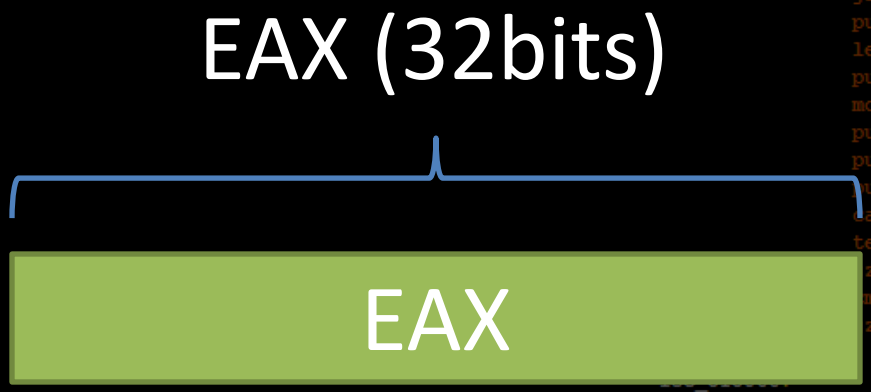
x86 CPU


```
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    eax
push    edi
        [ebp+arg_0], eax
        sub_31486A
        eax, eax
        short loc_31306D
        esi
        eax, [ebp+arg_0]
        eax
        esi, 1D0h
        esi
        [ebp+arg_4]
        edi
        sub_314623
        eax, eax
        short loc_31306D
        [ebp+arg_0], esi
        short loc_31308F

                        ; CODE XREF: sub_312FD8
                        ; sub_312FD8+55
        0Dh
        sub_31411B

                        ; CODE XREF: sub_312FD8
                        ; sub_312FD8+49
        sub_3140F3
        eax, eax
        short loc_31307D
        sub_3140F3
        short loc_31308C

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

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

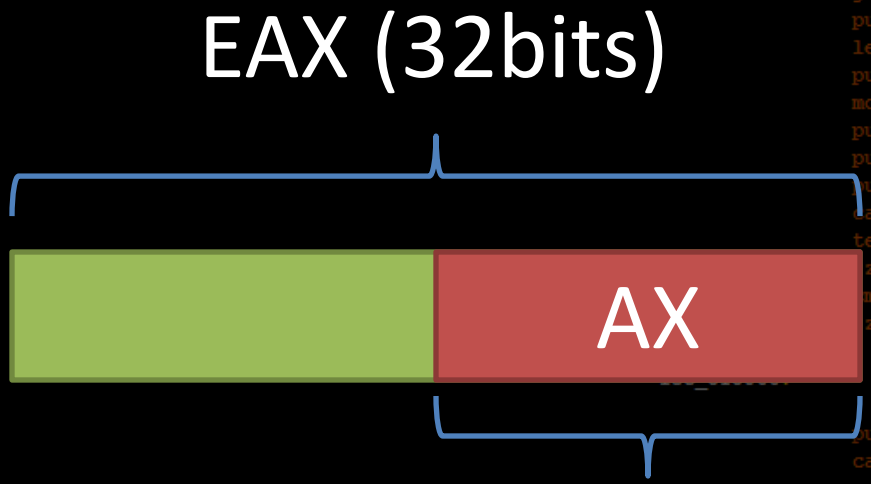
x86 Registers



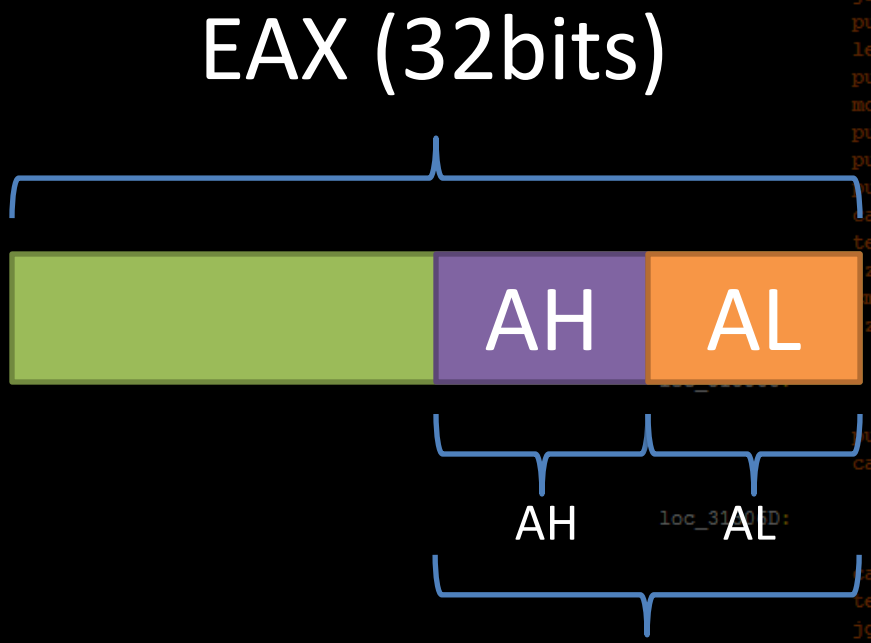
x86 Registers



x86 Registers

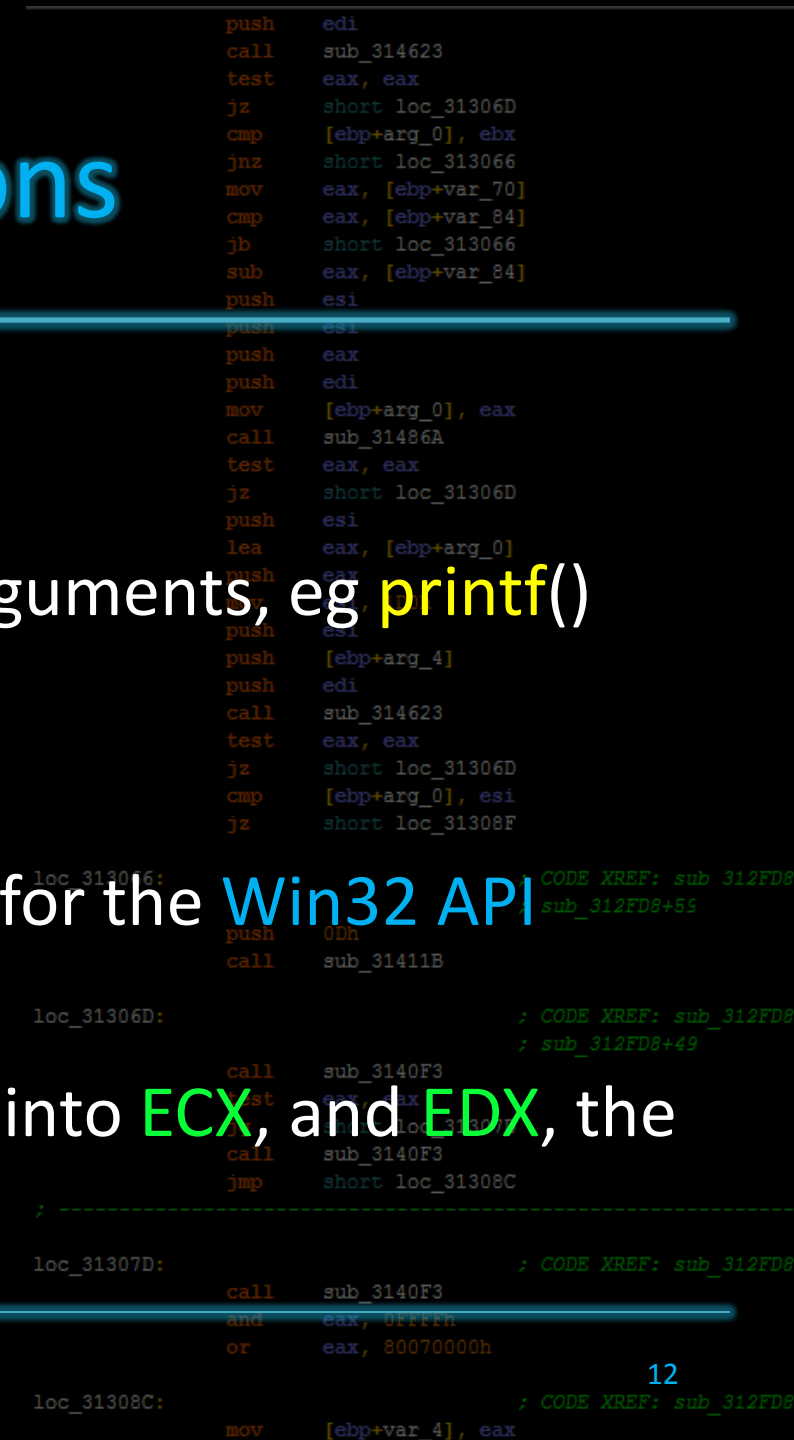


x86 Registers



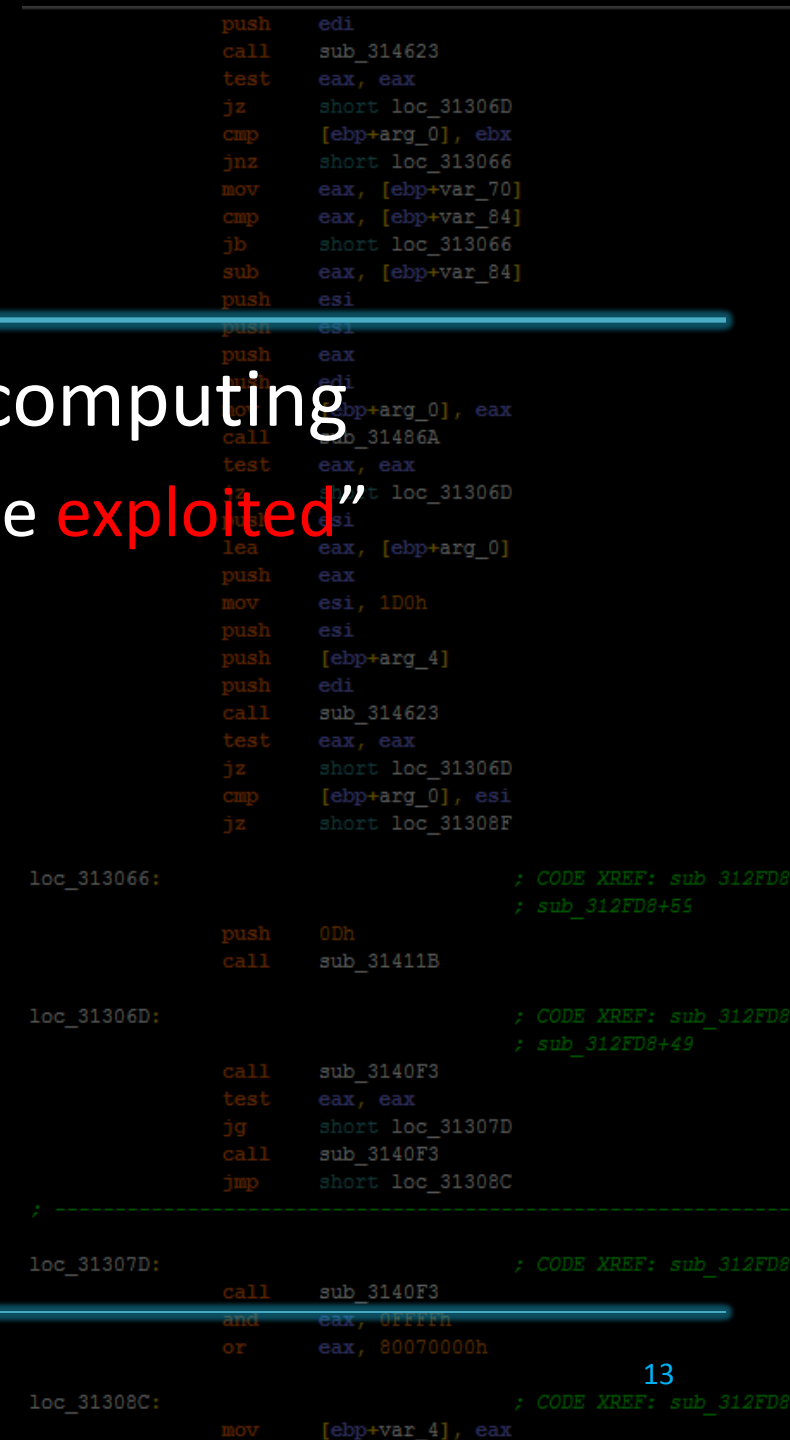
x86 Calling Conventions

- cdecl
  - Caller cleans up the stack
  - Unknown or variable # of arguments, eg printf()
- stdcall
  - Callee cleans up the stack
  - Standard calling convention for the Win32 API
- fastcall
  - First two arguments are put into ECX, and EDX, the rest are put onto the stack



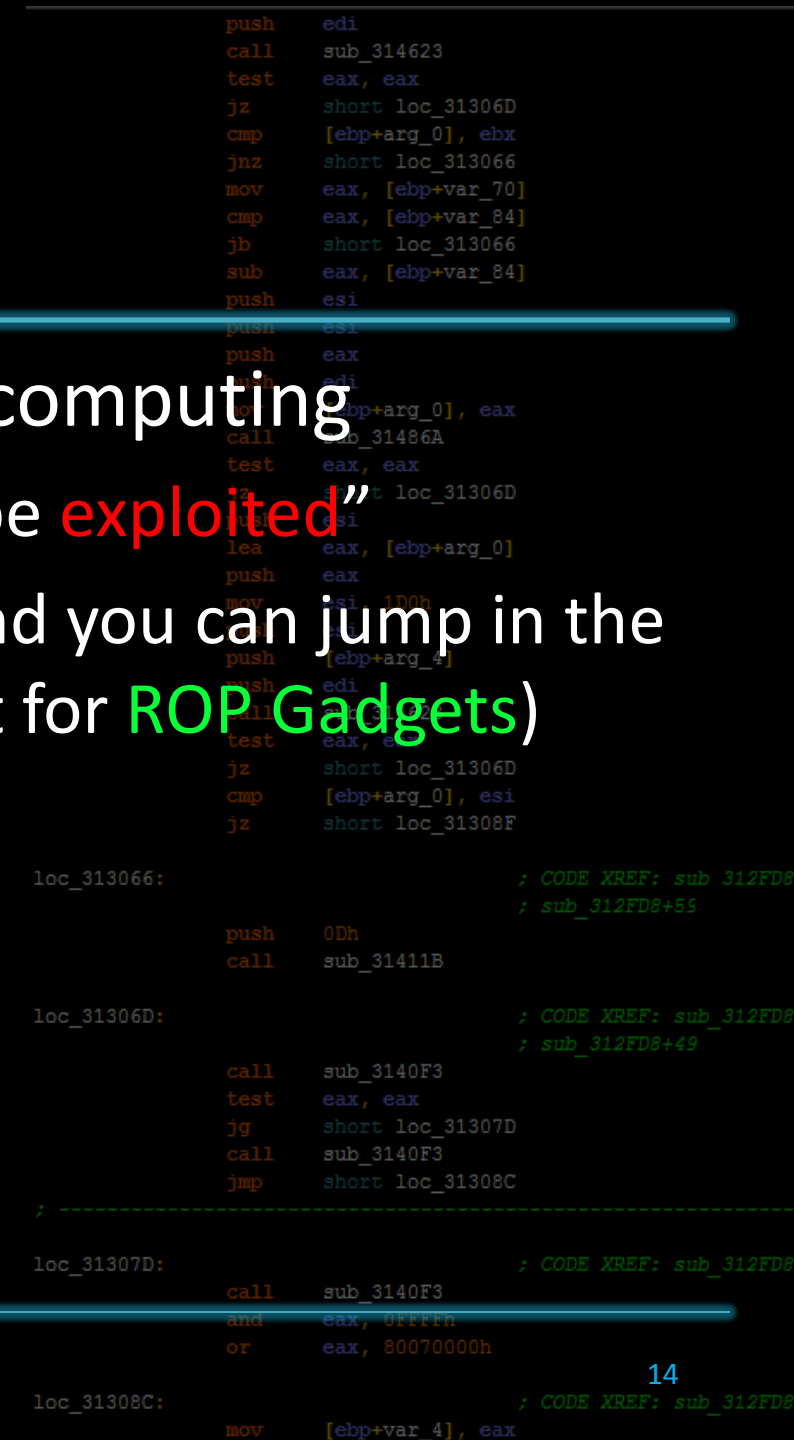
x86 Misc Notes

- x86 is like the wild west in computing
  - "it's like it was designed to be exploited"



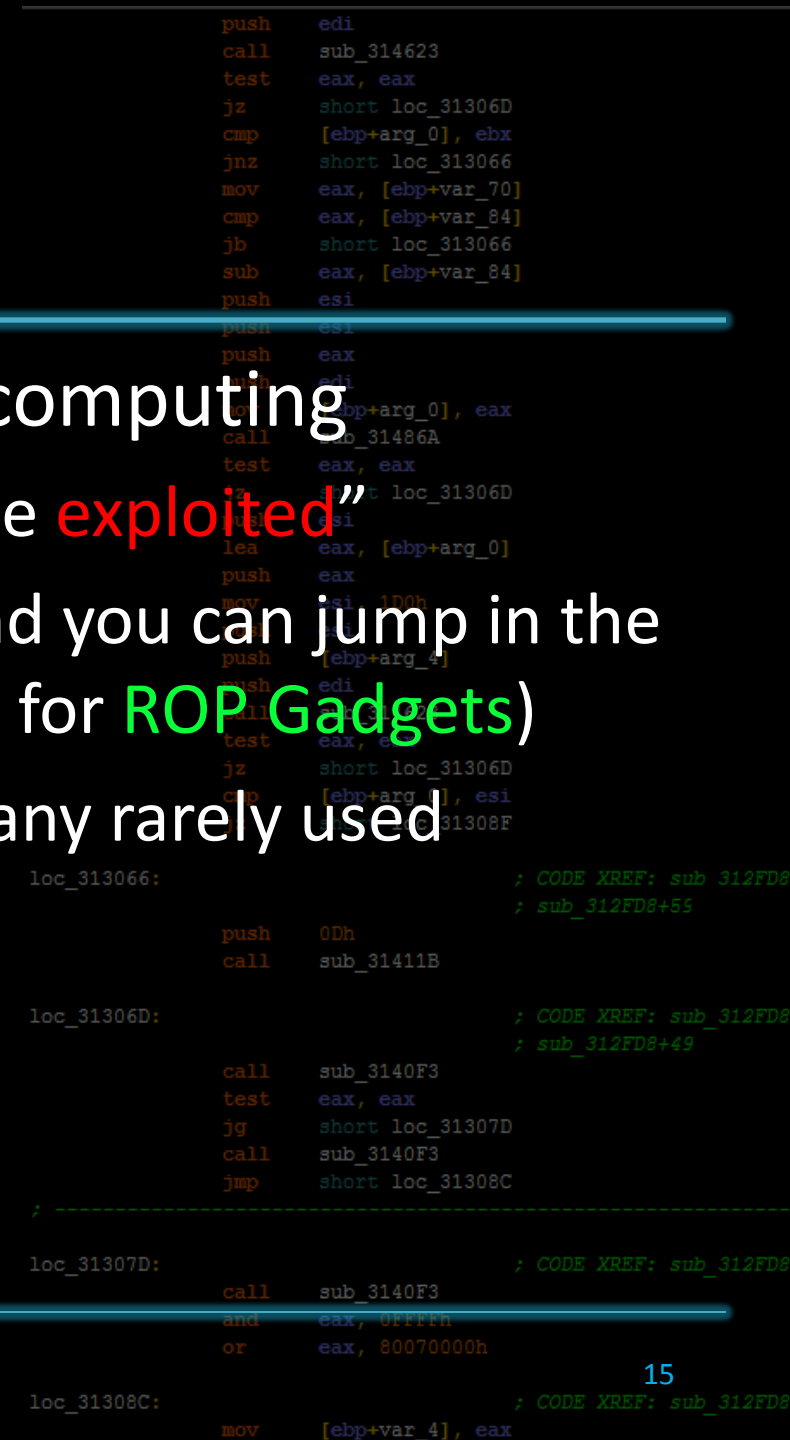
x86 Misc Notes

- x86 is like the wild west in computing
  - "it's like it was designed to be exploited"
  - No instruction alignment, and you can jump in the middle of instructions (great for ROP Gadgets)

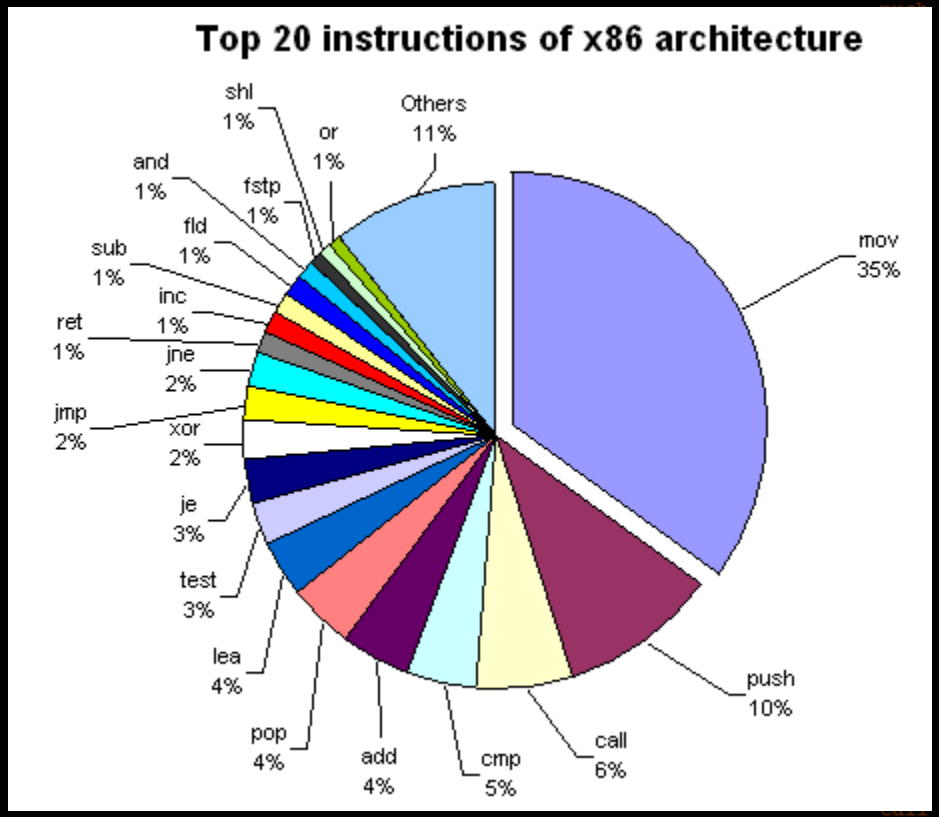


x86 Misc Notes

- x86 is like the wild west in computing
  - "it's like it was designed to be exploited"
  - No instruction alignment, and you can jump in the middle of instructions (great for ROP Gadgets)
  - Hundreds of instructions, many rarely used



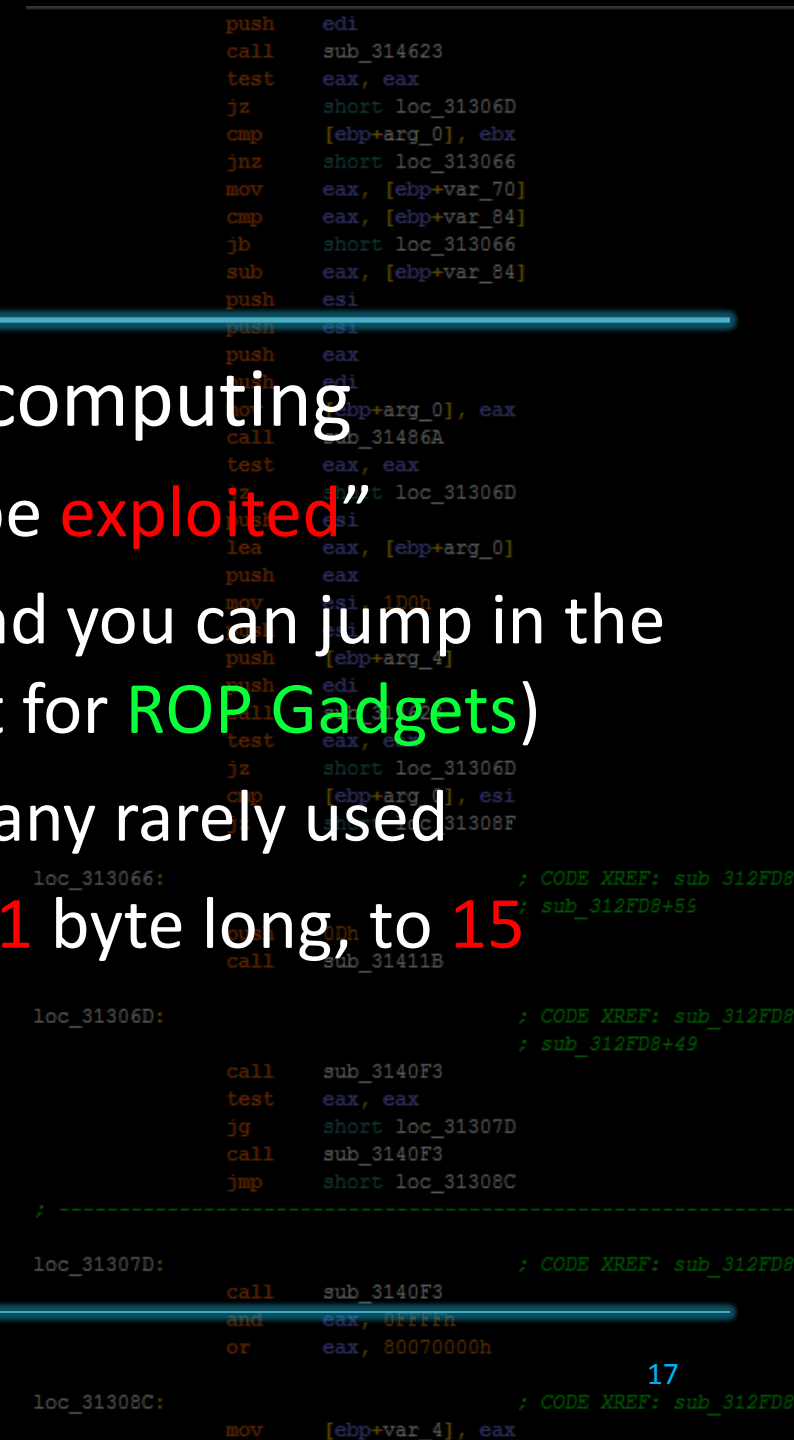
x86 Instruction Stats



http://www.strchr.com/x86_machine_code_statistics

x86 Misc Notes

- x86 is like the wild west in computing
  - "it's like it was designed to be exploited"
  - No instruction alignment, and you can jump in the middle of instructions (great for ROP Gadgets)
  - Hundreds of instructions, many rarely used
  - Instructions can range from 1 byte long, to 15 bytes long!



x86 Long Instructions

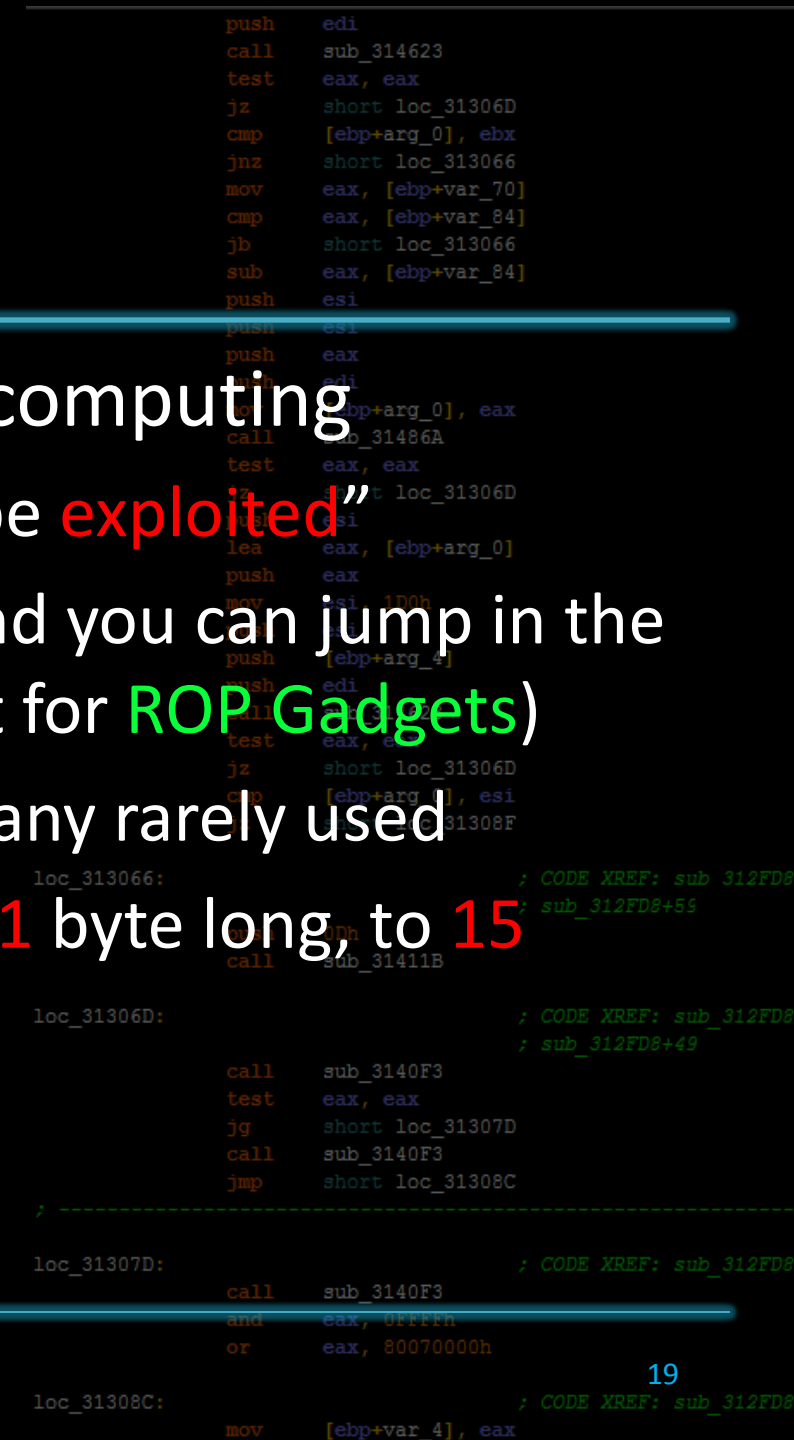
```
lock add DWORD PTR ds:[esi+ecx*4+0x12345678],0xefcdab89
```

```
67 66 f0 3e 81 84 8e 78 56 34 12 89 ab cd ef
```

(from http://blog.onlinedisassembler.com/blog/?p=23)

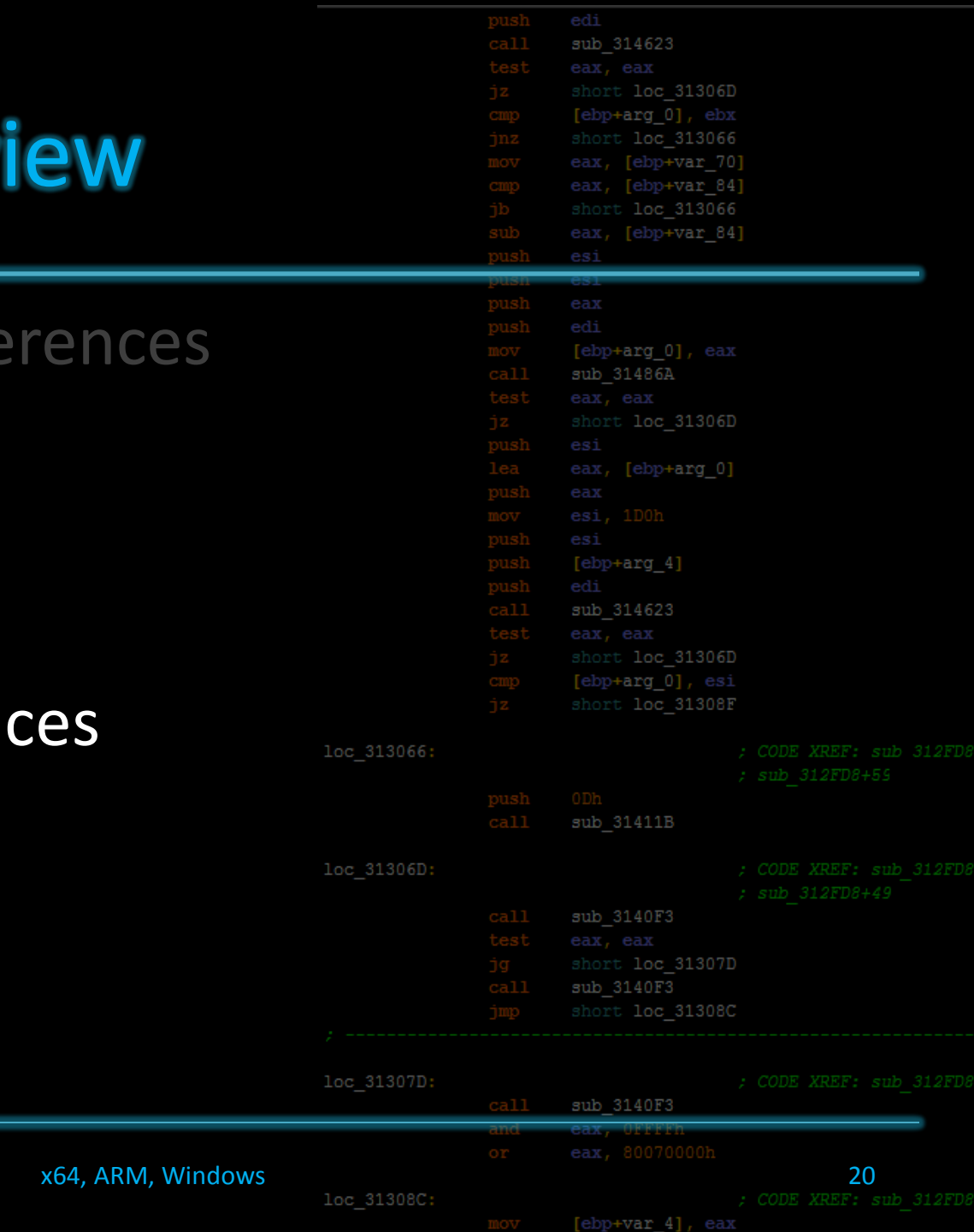
x86 Misc Notes

- x86 is like the wild west in computing
  - "it's like it was designed to be exploited"
  - No instruction alignment, and you can jump in the middle of instructions (great for ROP Gadgets)
  - Hundreds of instructions, many rarely used
  - Instructions can range from 1 byte long, to 15 bytes long!
- It's the devil's playground



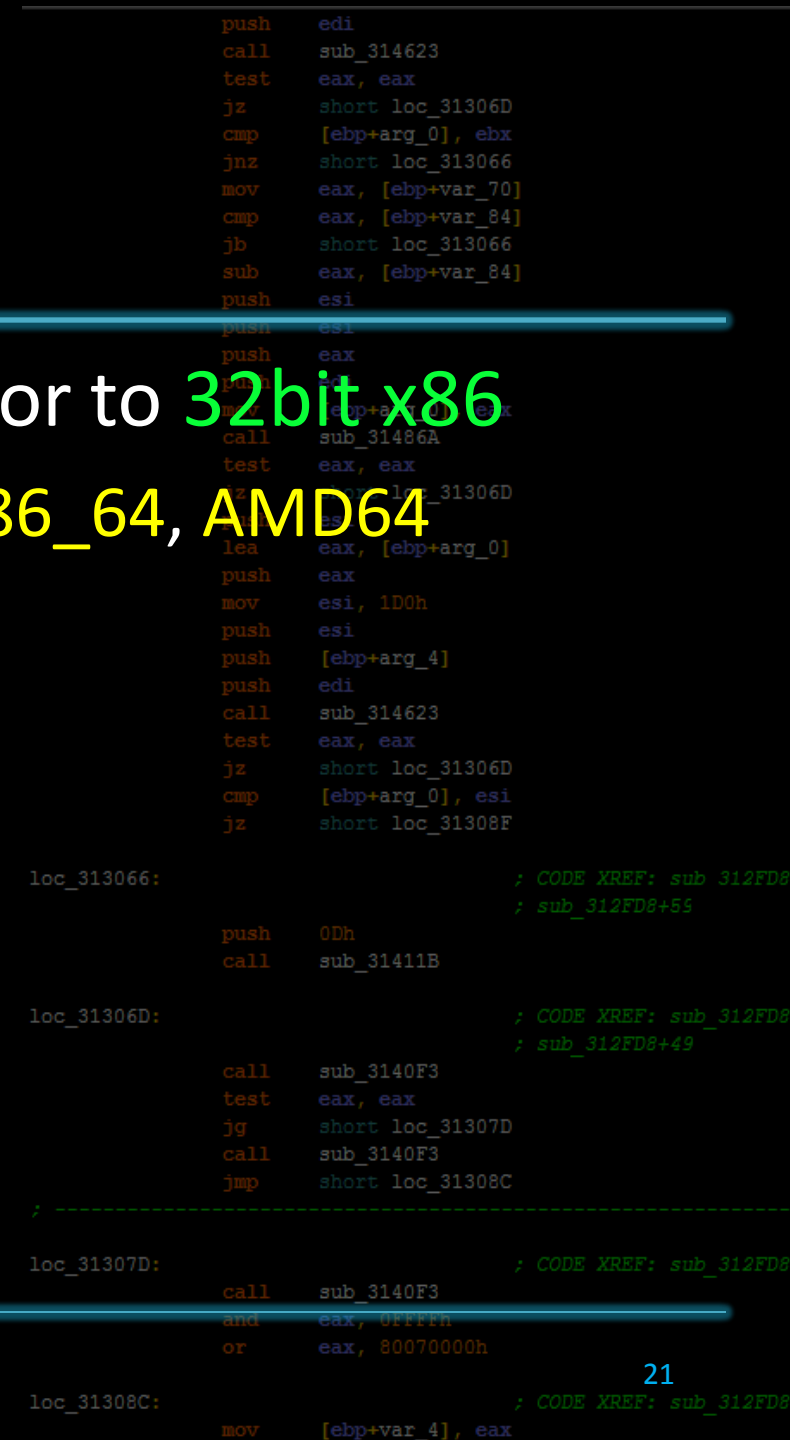
Lecture Overview

- Architecture Differences
  - x86
  - x86_64
  - ARM
- Platform Differences
  - Windows



x86_64 Overview

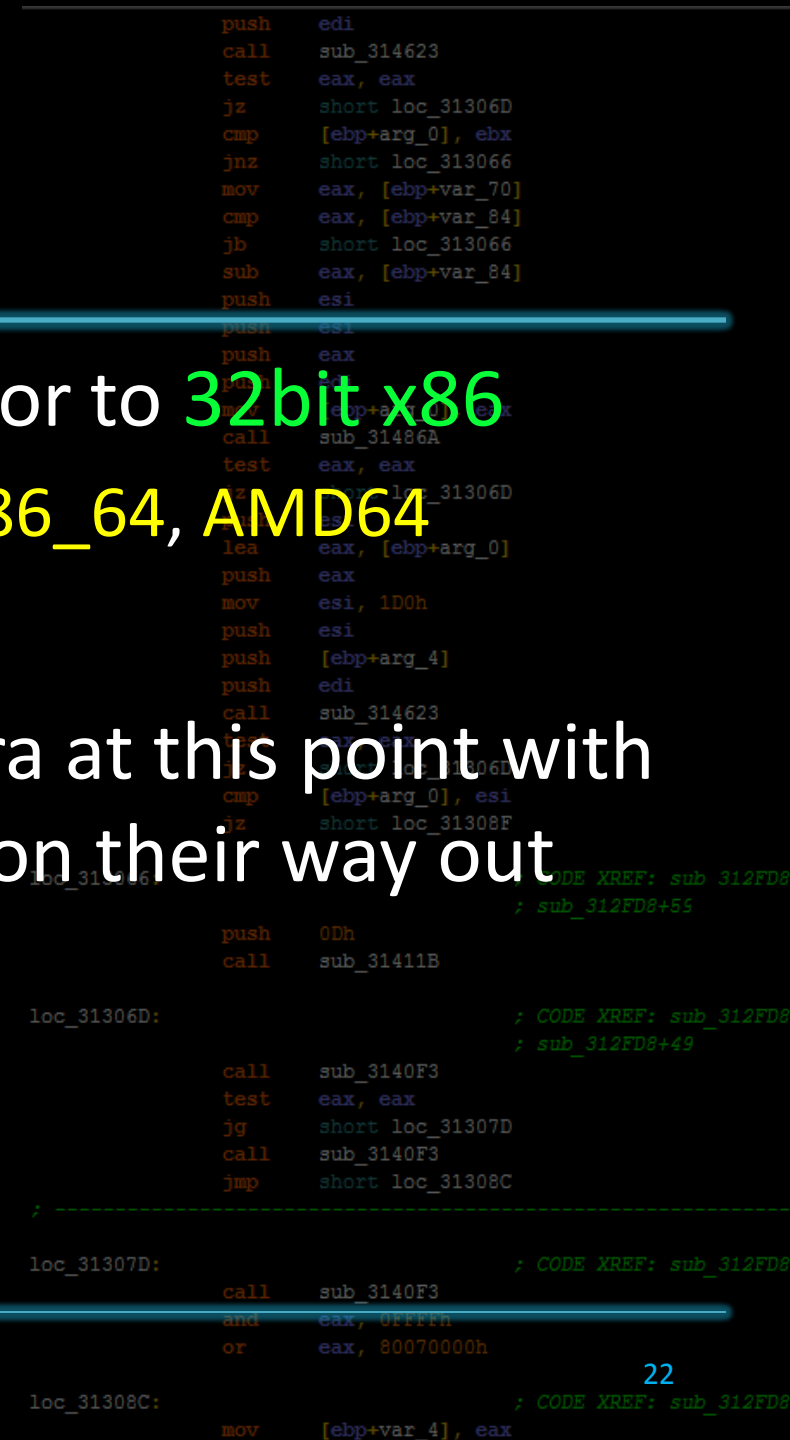
- x86_64 is the 64bit successor to 32bit x86
  - Sometimes known as x64, x86_64, AMD64



x86_64 Overview

- x86_64 is the 64bit successor to 32bit x86
  - Sometimes known as x64, x86_64, AMD64

- We're well into the 64bit era at this point with 32bit x86 machines slowly on their way out



x86_64 Overview

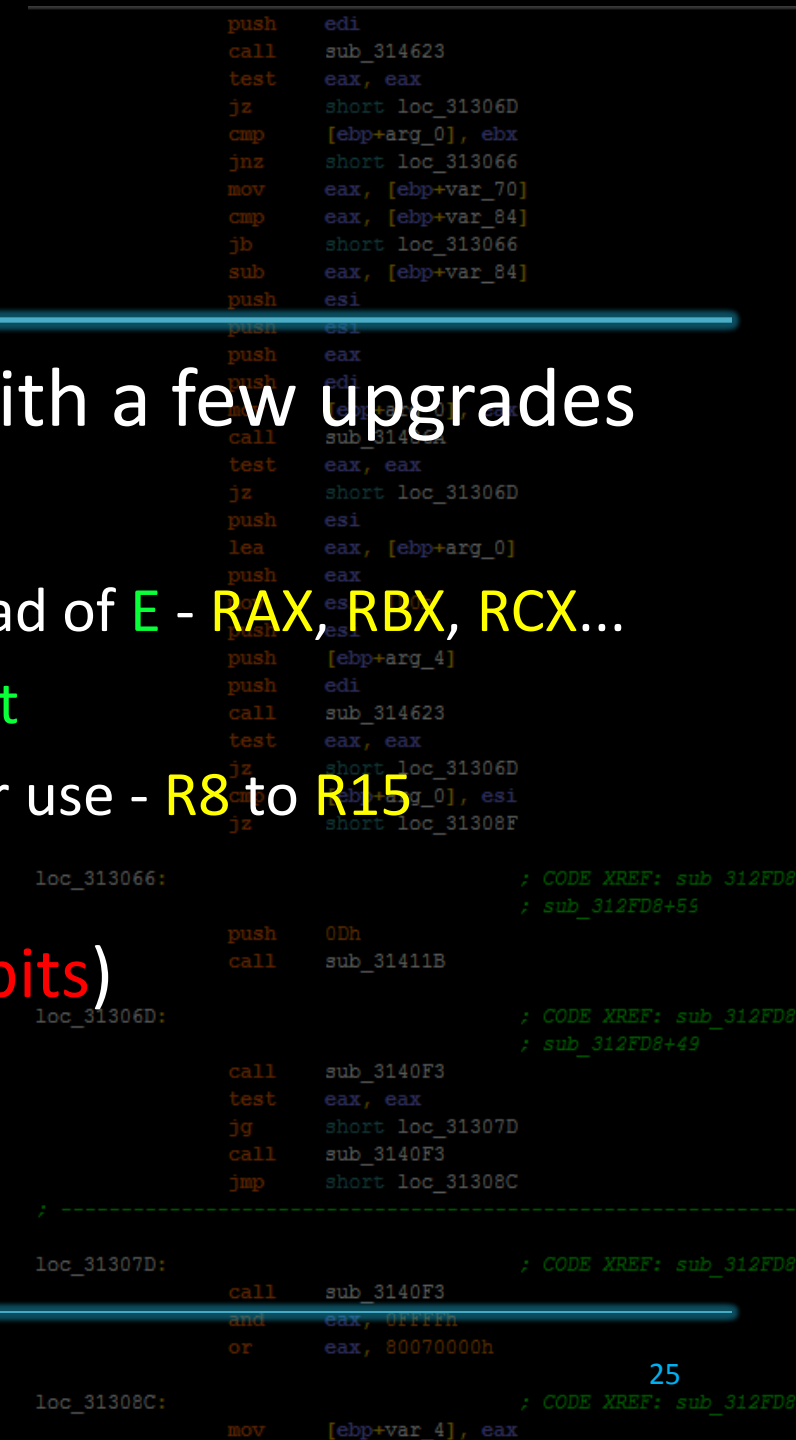
- x86_64 is the 64bit successor to 32bit x86
  - Sometimes known as x64, x86_64, AMD64
- We're well into the 64bit era at this point with 32bit x86 machines slowly on their way out
- x86_64 is Bigger, better, faster… and familiar!

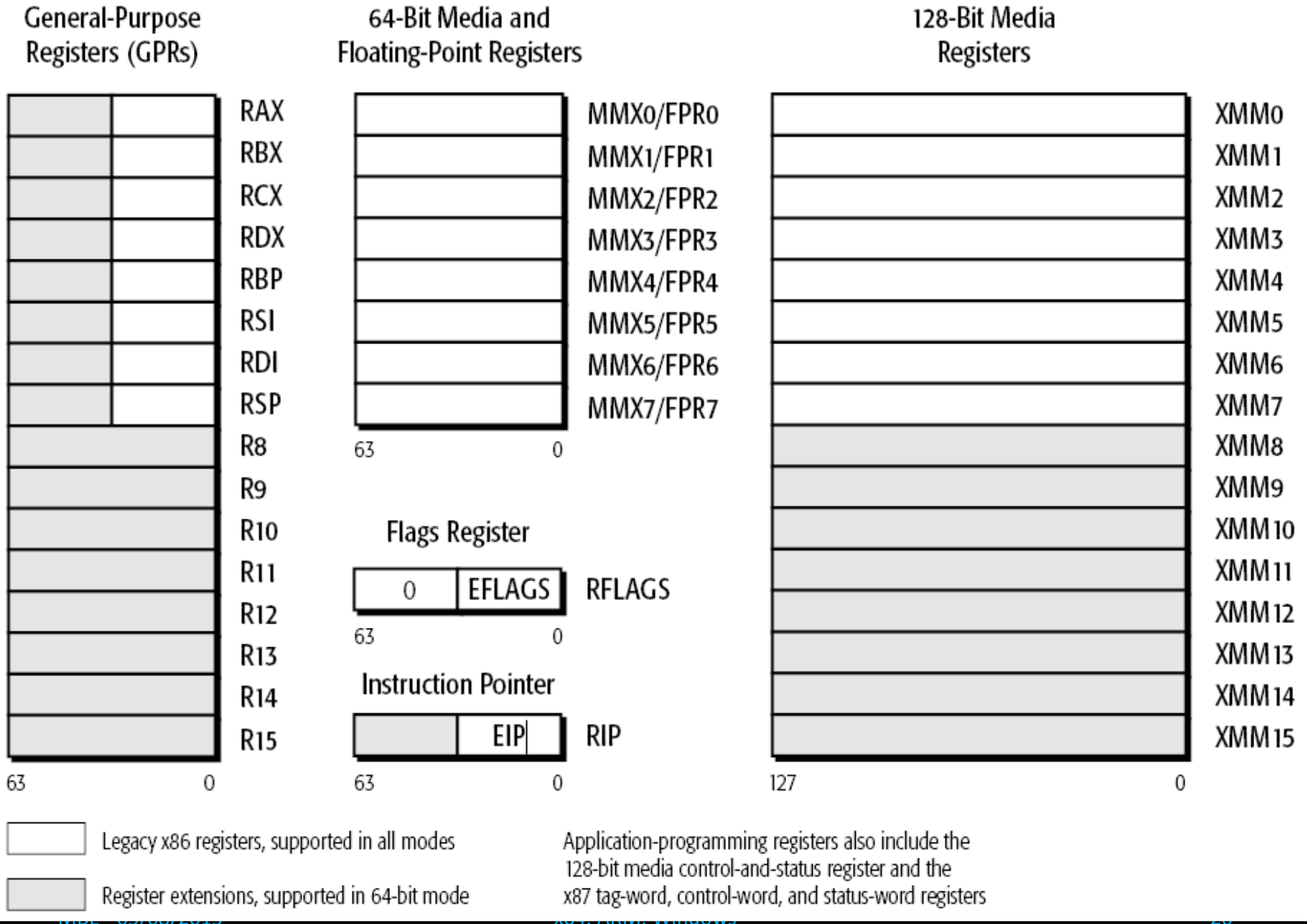
x86_64 CPU


```
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
eax
edi
[ebp+arg_0], eax
sub_31486A
eax, eax
short loc_31306D
esi
eax, [ebp+arg_0]
eax
esi, 1D0h
esi
[ebp+arg_4]
edi
sub_314623
eax, eax
short loc_31306D
[ebp+arg_0], esi
short loc_31308F

                ; CODE XREF: sub_312FD8
                ; sub_312FD8+55
0Dh
sub_31411B

                ; CODE XREF: sub_312FD8
                ; sub_312FD8+49
sub_3140F3
eax, eax
short loc_31307D
sub_3140F3
short loc_31308C

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

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

x86_64 Registers

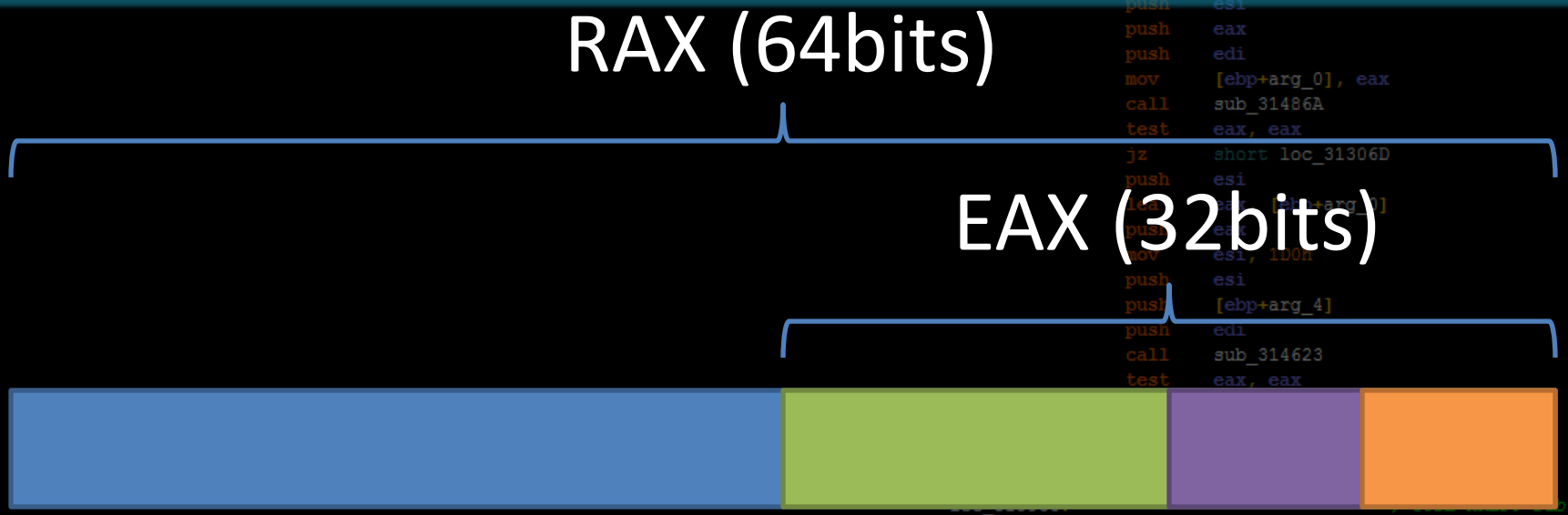
- Pretty similar to x86, but with a few upgrades
  - General Purpose Registers
    - Everything starts with R instead of E - RAX, RBX, RCX...
    - GPR's are now 64bit, not 32bit
    - There is now 8 more GPR's for use - R8 to R15
  - More XMM* registers (128 bits)



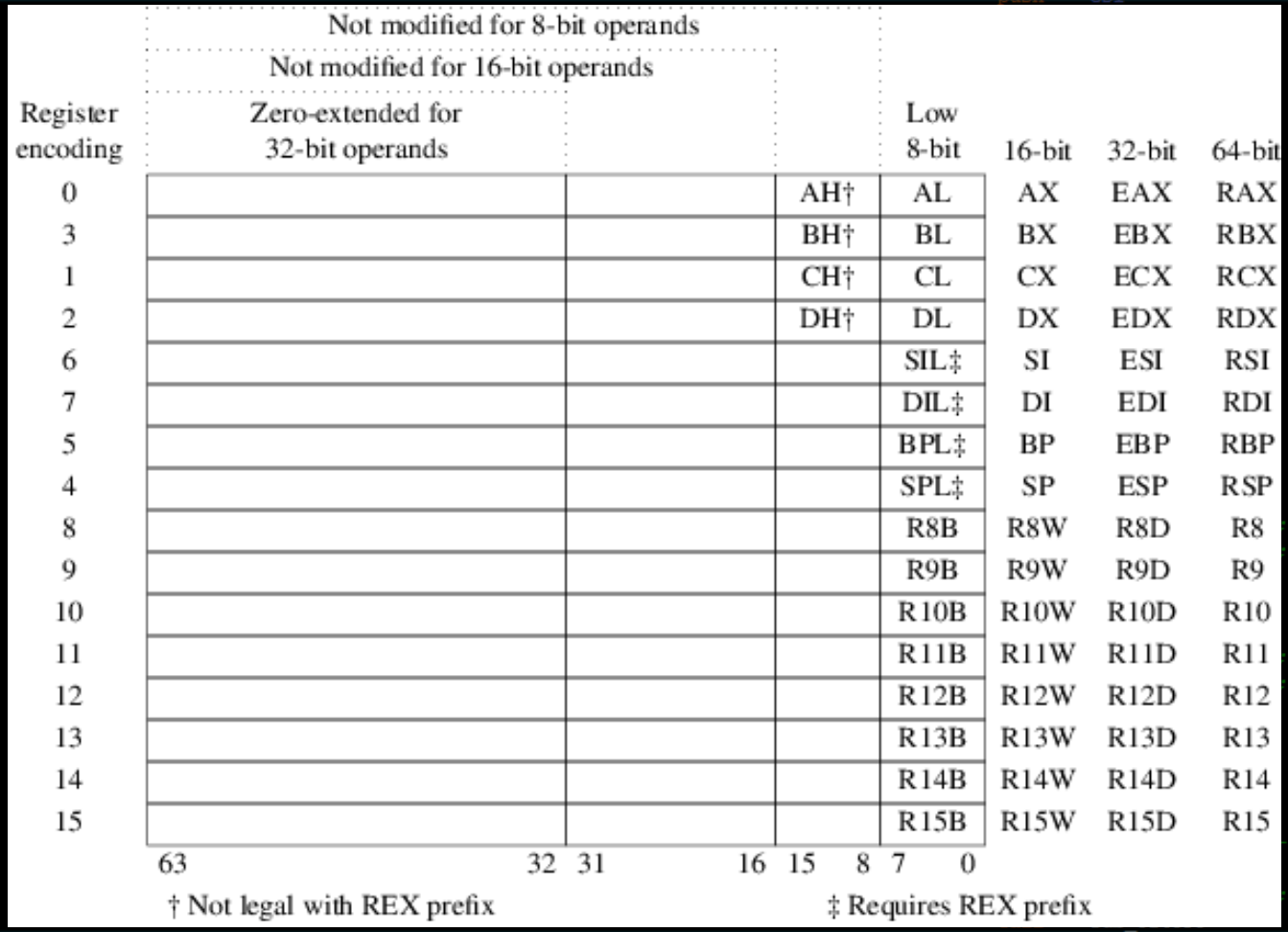

General-Purpose Registers (GPRs)
RAX
RBX
RCX
RDX
RBP
RSI
RDI
RSP
R8
R9
R10
R11
R12
R13
R14
R15
63
0
64-Bit Media and Floating-Point Registers
MMX0/FPR0
MMX1/FPR1
MMX2/FPR2
MMX3/FPR3
MMX4/FPR4
MMX5/FPR5
MMX6/FPR6
MMX7/FPR7
63
0
Flags Register
0
EFLAGS
RFLAGS
63
0
Instruction Pointer
EIP
RIP
63
0
128-Bit Media Registers
XMM0
XMM1
XMM2
XMM3
XMM4
XMM5
XMM6
XMM7
XMM8
XMM9
XMM10
XMM11
XMM12
XMM13
XMM14
XMM15
127
0
Legacy x86 registers, supported in all modes
Register extensions, supported in 64-bit mode
Application-programming registers also include the 128-bit media control-and-status register and the x87 tag-word, control-word, and status-word registers

x86_64 Registers

x86_64 Registers



x86_64 Registers



Not modified for 8-bit operands
Not modified for 16-bit operands
Zero-extended for 32-bit operands

Register encoding	Low 8-bit	16-bit	32-bit	64-bit
0	AL	AX	EAX	RAX
3	BL	BX	EBX	RBX
1	CL	CX	ECX	RCX
2	DL	DX	EDX	RDX
6	SIL‡	SI	ESI	RSI
7	DIL‡	DI	EDI	RDI
5	BPL‡	BP	EBP	RBP
4	SPL‡	SP	ESP	RSP
8	R8B	R8W	R8D	R8
9	R9B	R9W	R9D	R9
10	R10B	R10W	R10D	R10
11	R11B	R11W	R11D	R11
12	R12B	R12W	R12D	R12
13	R13B	R13W	R13D	R13
14	R14B	R14W	R14D	R14
15	R15B	R15W	R15D	R15
	AH†			
	BH†			
	CH†			
	DH†			

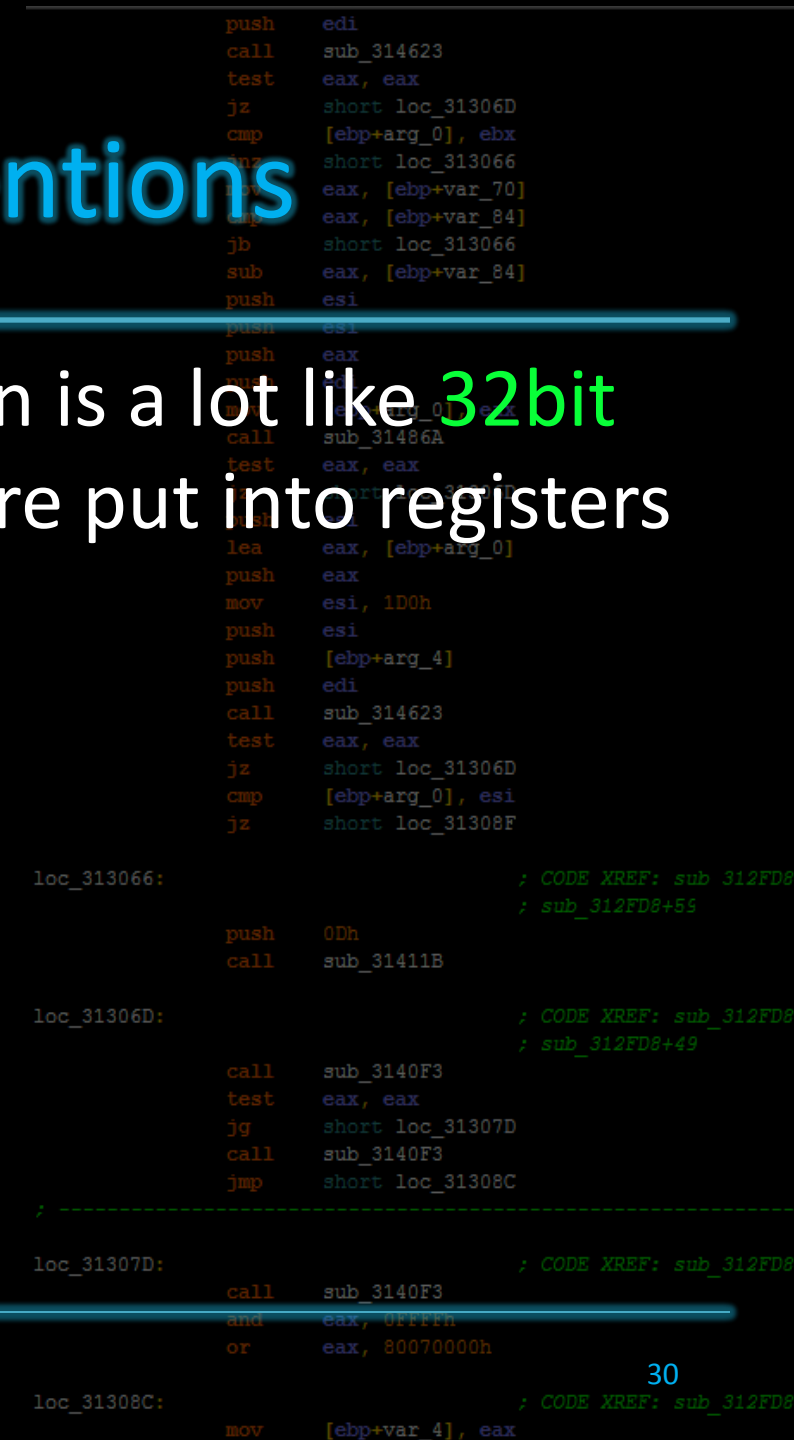
63 32 31 16 15 8 7 0

† Not legal with REX prefix

‡ Requires REX prefix

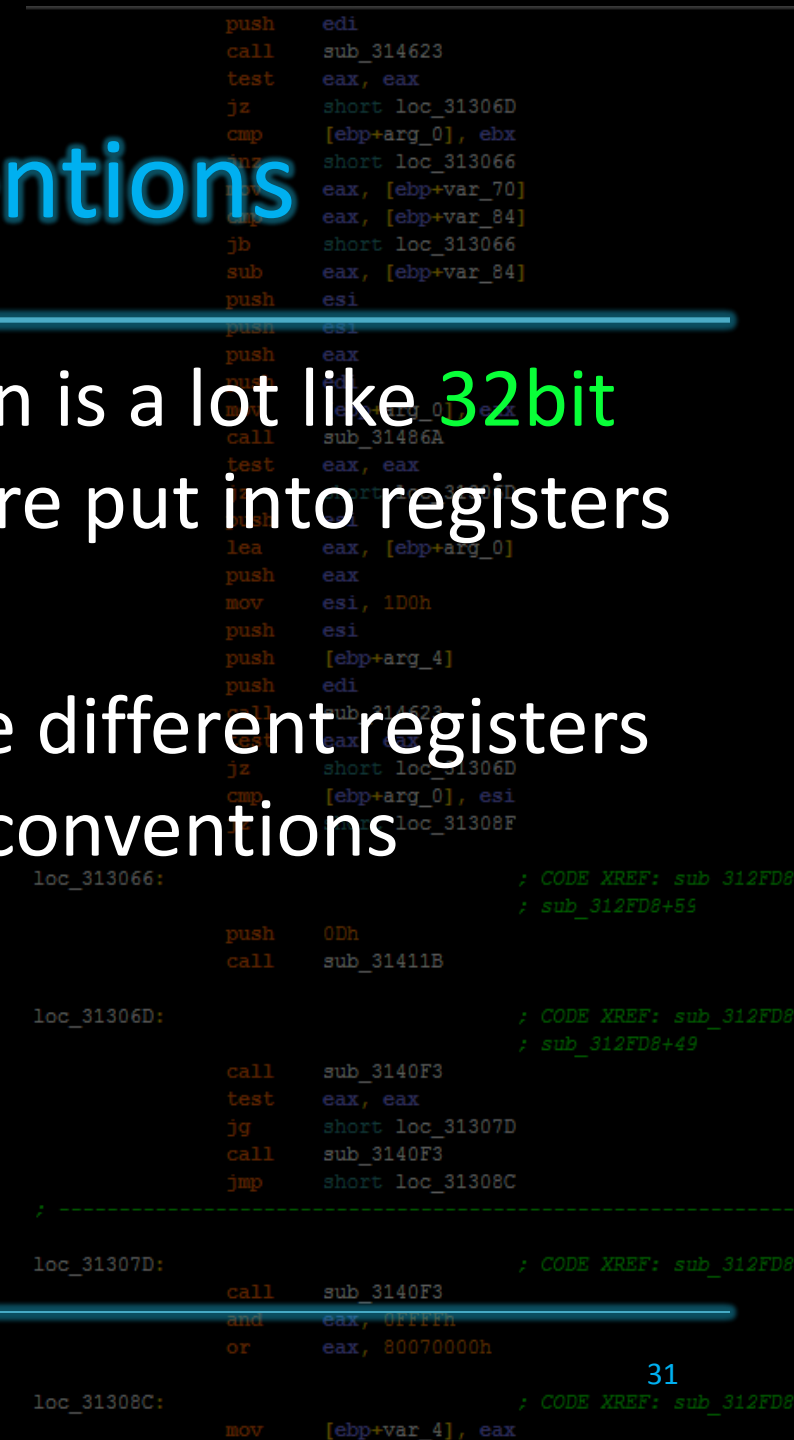
x86_64 Calling Conventions

- The 64bit calling convention is a lot like 32bit fastcall where arguments are put into registers



x86_64 Calling Conventions

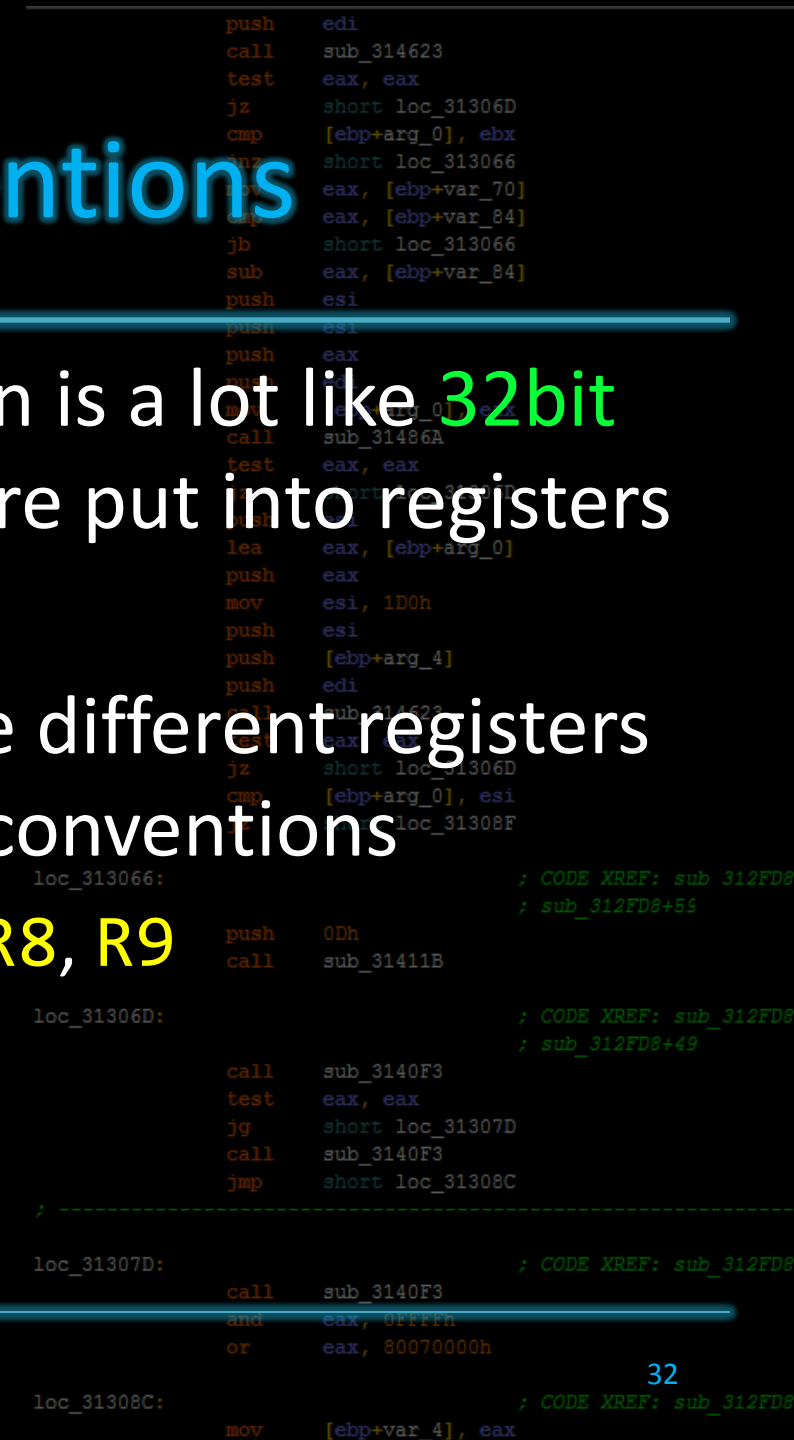
- The 64bit calling convention is a lot like 32bit fastcall where arguments are put into registers
- But Linux and Windows use different registers for their respective calling conventions



x86_64 Calling Conventions

- The 64bit calling convention is a lot like 32bit fastcall where arguments are put into registers

- But Linux and Windows use different registers for their respective calling conventions
  - Linux: RDI, RSI, RDX, RCX, R8, R9
  - Windows: RCX, RDX, R8, R9



x86_64 Calling Conventions

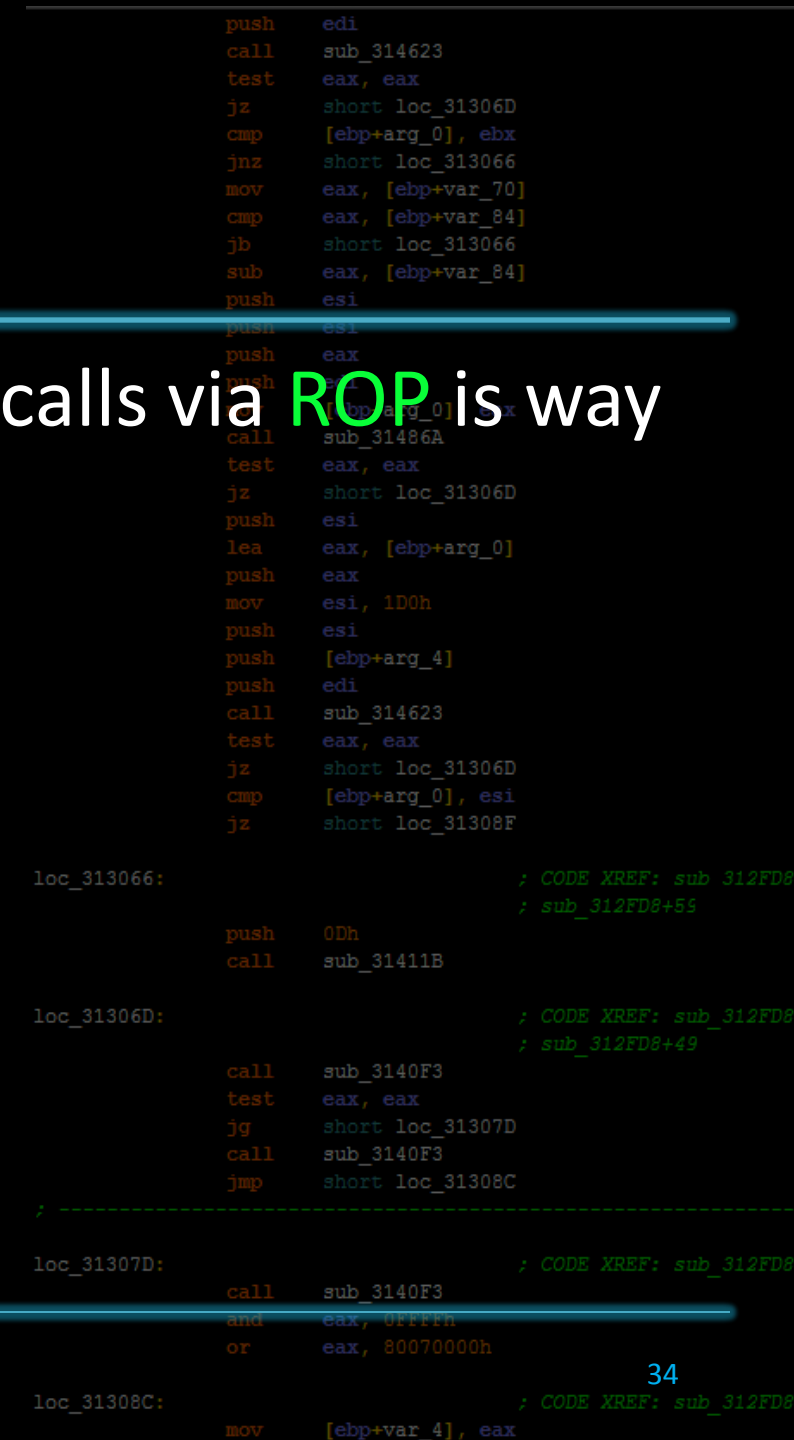
- The 64bit calling convention is a lot like 32bit fastcall where arguments are put into registers

- But Linux and Windows use different registers for their respective calling conventions
  - Linux: RDI, RSI, RDX, RCX, R8, R9
  - Windows: RCX, RDX, R8, R9

(any other arguments are pushed onto the stack)

x86_64 ROP

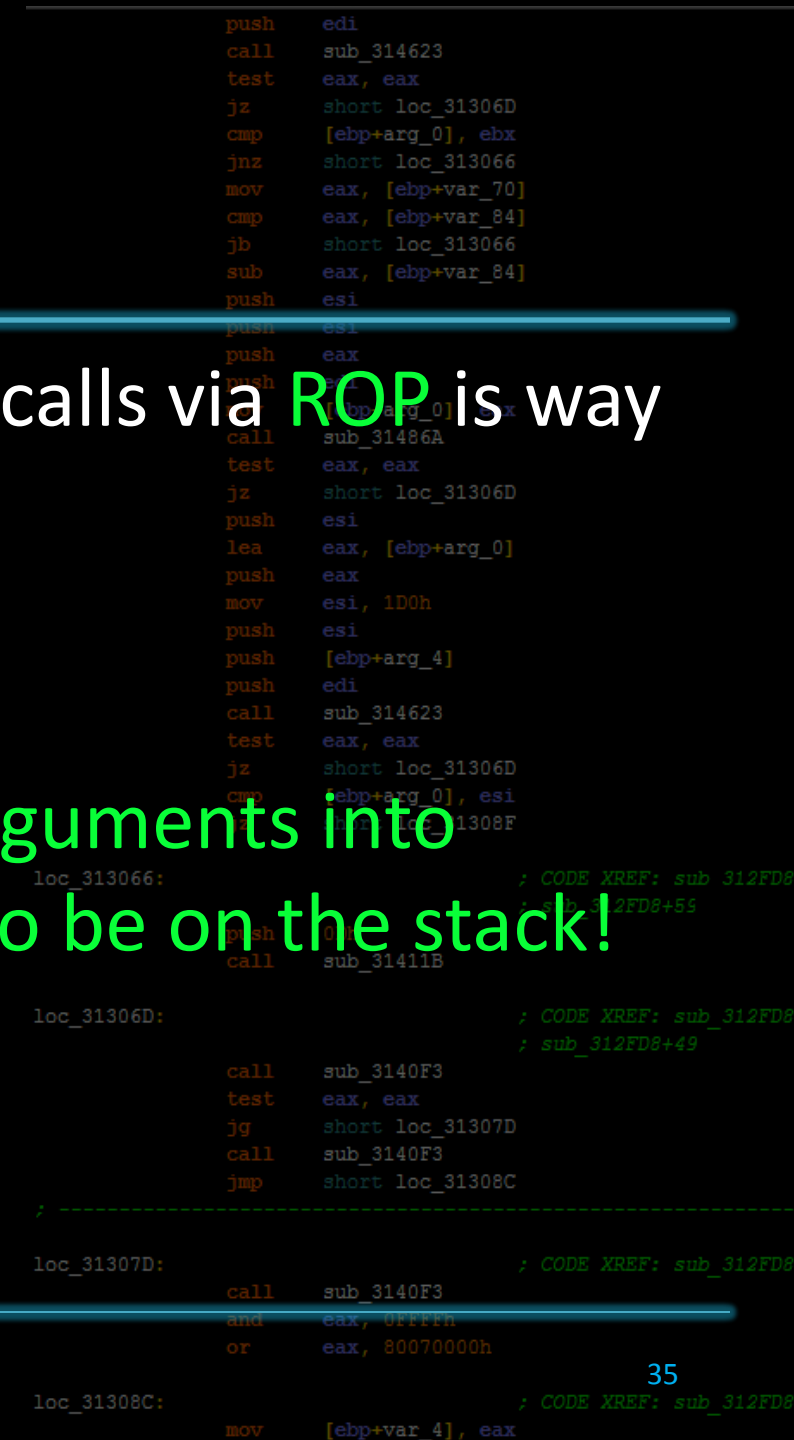
- Chaining multiple function calls via ROP is way easier on 64bit
  - Why?



x86_64 ROP

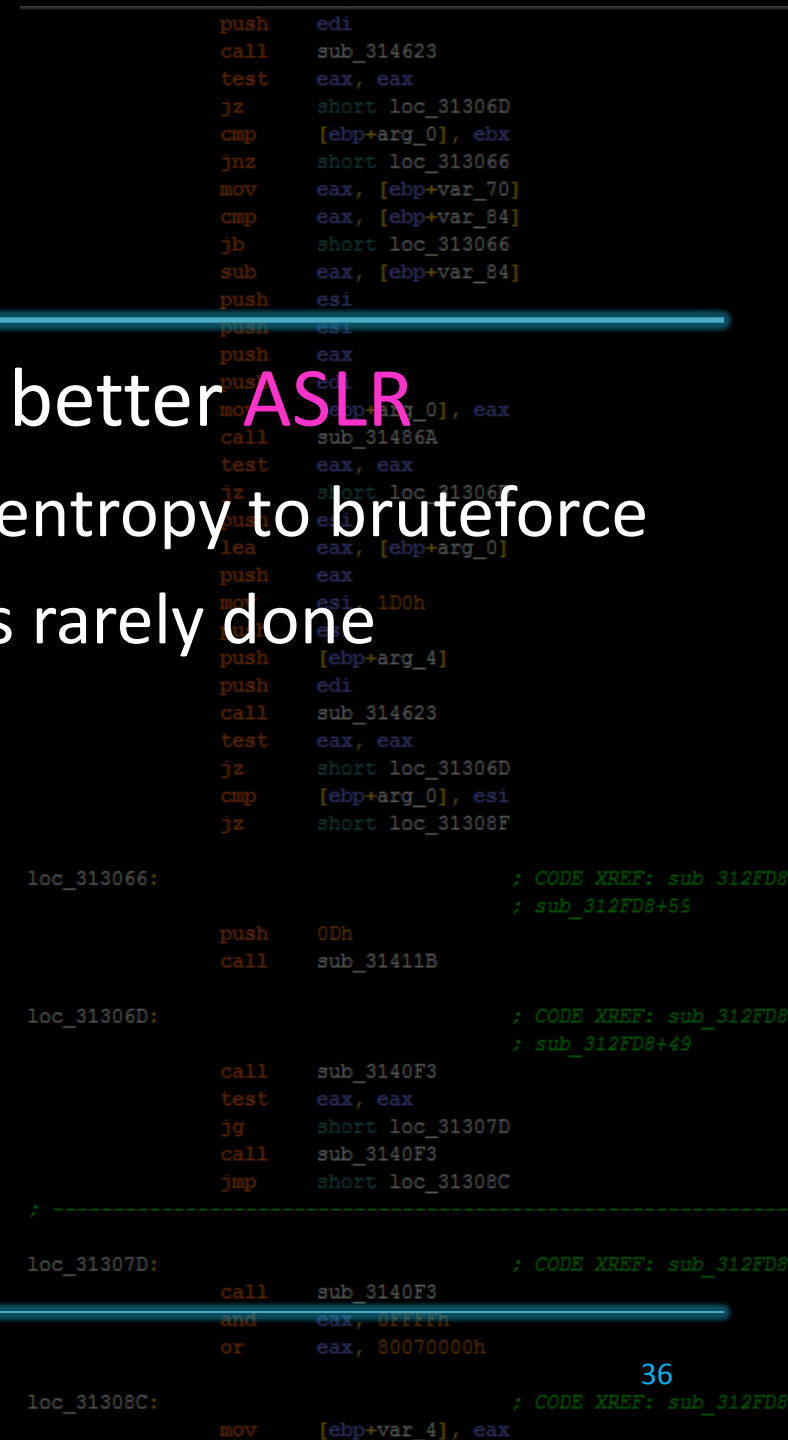
- Chaining multiple function calls via ROP is way easier on 64bit
  - Why?

- You simply load function arguments into registers, they don't need to be on the stack!



x86_64 ASLR

- 64bit address space means better ASLR
  - 'better' simply means more entropy to bruteforce
  - Bruteforcing ASLR on 64bit is rarely done



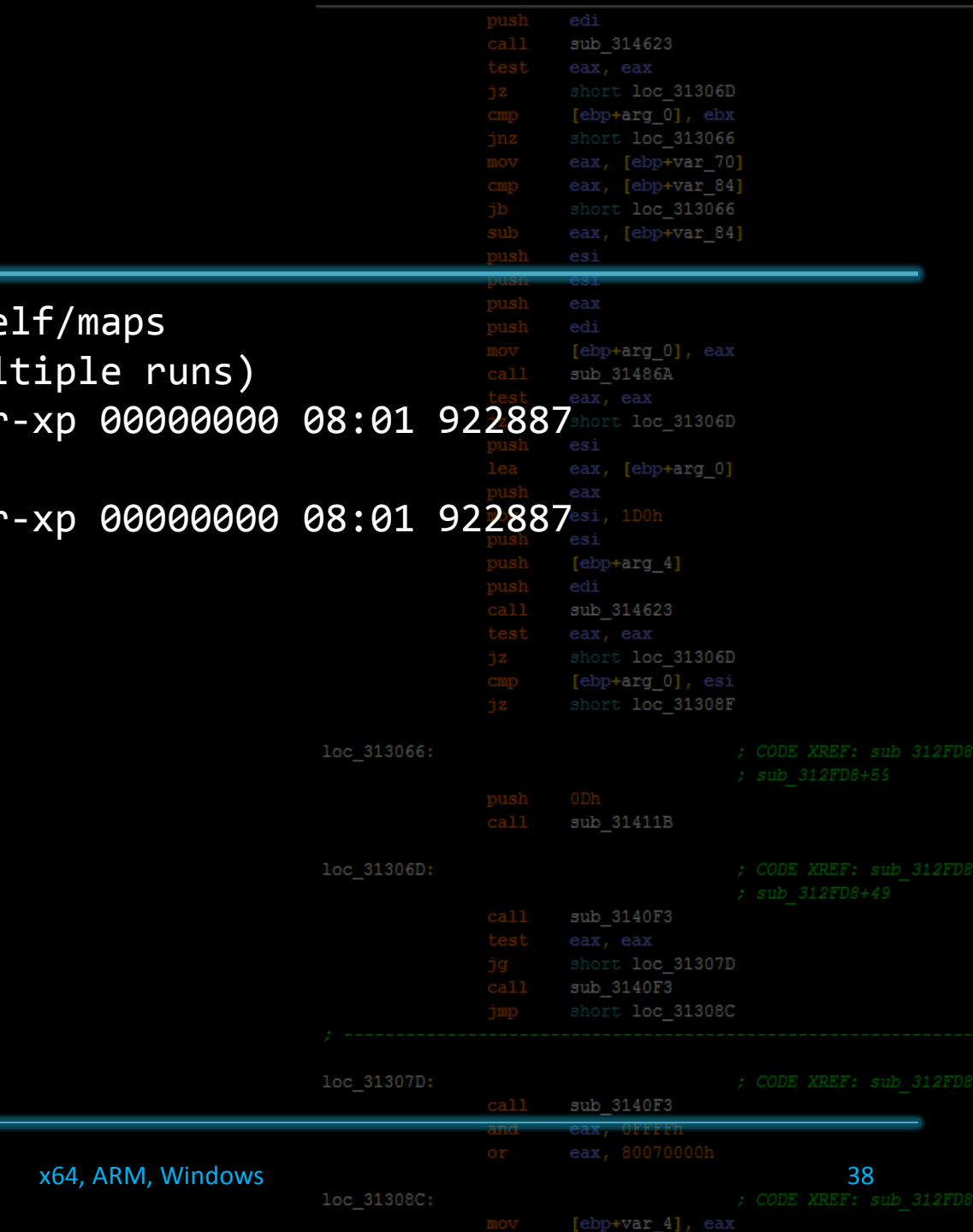
x86_64 ASLR

```
doom@upwn64:~$ cat /proc/self/maps
(the same segment after multiple runs)
7f638218c000-7f6382347000 r-xp 00000000 08:01 922887
...
```

```
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, 0FFFFFh
or      eax, 80070000h

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

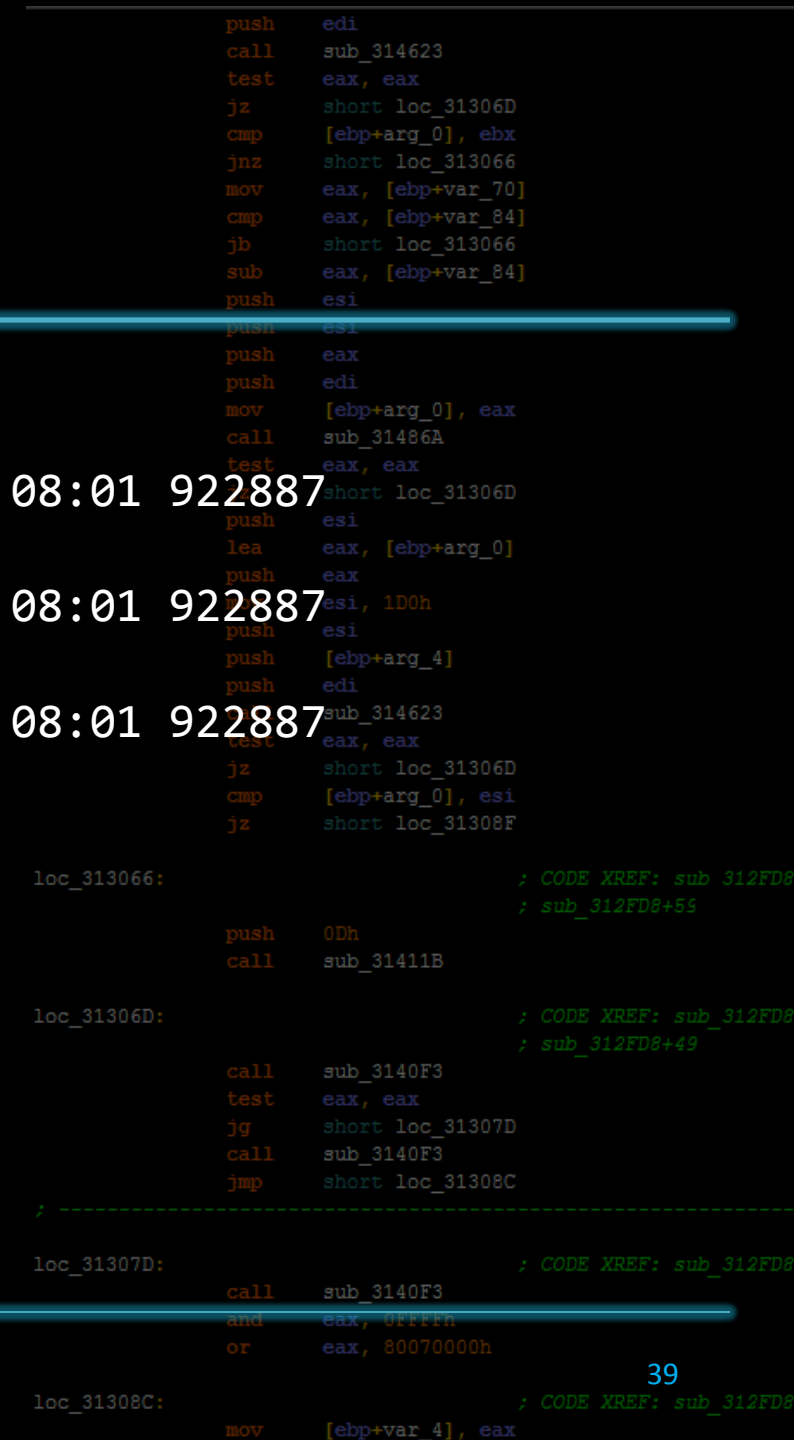
x86_64 ASLR

```
doom@upwn64:~$ cat /proc/self/maps
(the same segment after multiple runs)
7f638218c000-7f6382347000 r-xp 00000000 08:01 922887
...
7f6fa368e000-7f6fa3849000 r-xp 00000000 08:01 922887
...
```



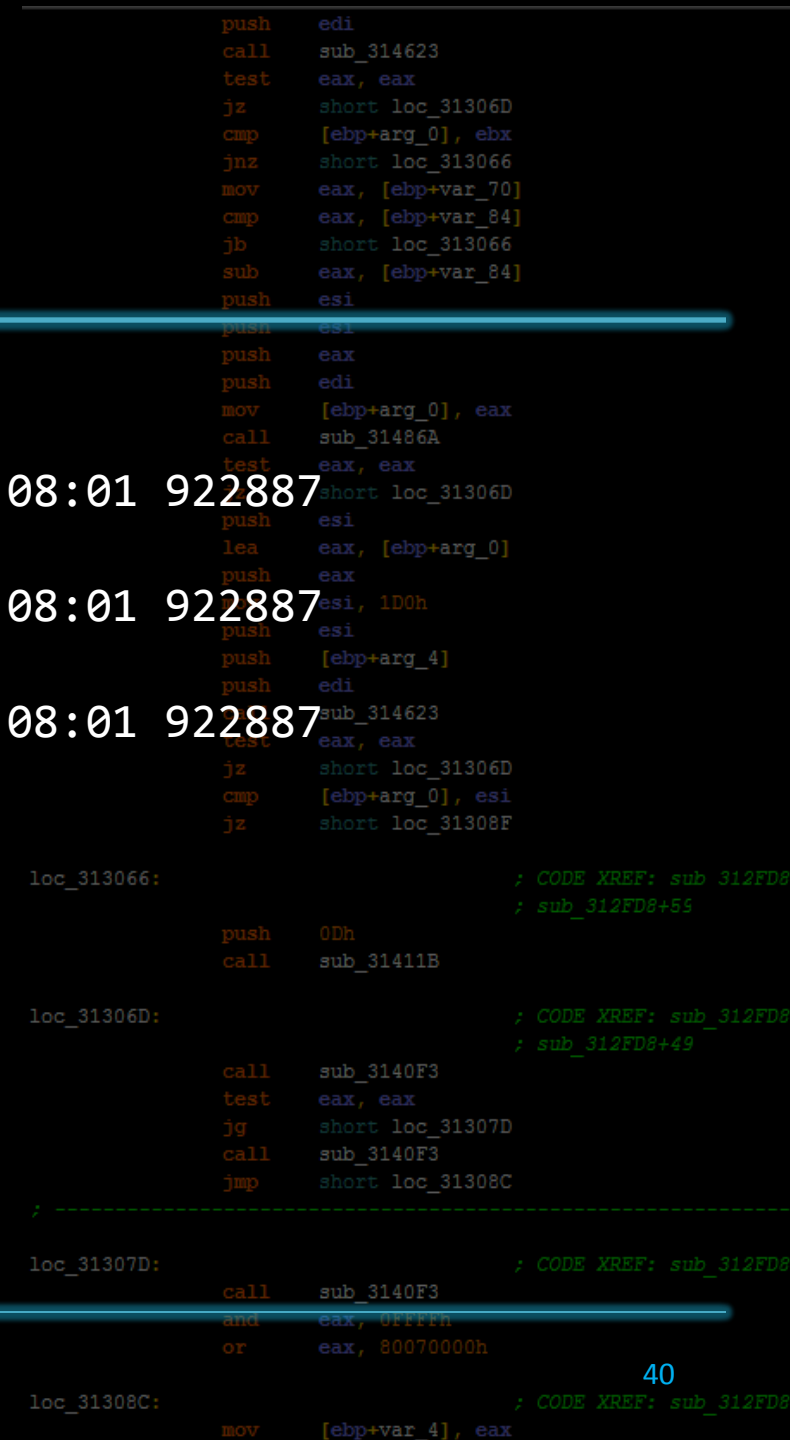
x86_64 ASLR

```
doom@upwn64:~$ cat /proc/self/maps
(the same segment after multiple runs)
7f638218c000-7f6382347000 r-xp 00000000 08:01 922887
...
7f6fa368e000-7f6fa3849000 r-xp 00000000 08:01 922887
...
7f974db38000-7f974dcf3000 r-xp 00000000 08:01 922887
...
```



x86_64 ASLR

```
doom@upwn64:~$ cat /proc/self/maps
(the same segment after multiple runs)
7f638218c000-7f6382347000 r-xp 00000000 08:01 922887
...
7f6fa368e000-7f6fa3849000 r-xp 00000000 08:01 922887
...
7f974db38000-7f974dcf3000 r-xp 00000000 08:01 922887
...
```



x86_64 ASLR

```
doom@upwn64:~$ cat /proc/self/maps
(the same segment after multiple runs)
7f638218c000-7f6382347000 r-xp 00000000 08:01 922887
...
7f6fa368e000-7f6fa3849000 r-xp 00000000 08:01 922887
...
7f974db38000-7f974dcf3000 r-xp 00000000 08:01 922887
...
```

At least 7 nibbles of libc is changing per run on Ubuntu 14.04 x64

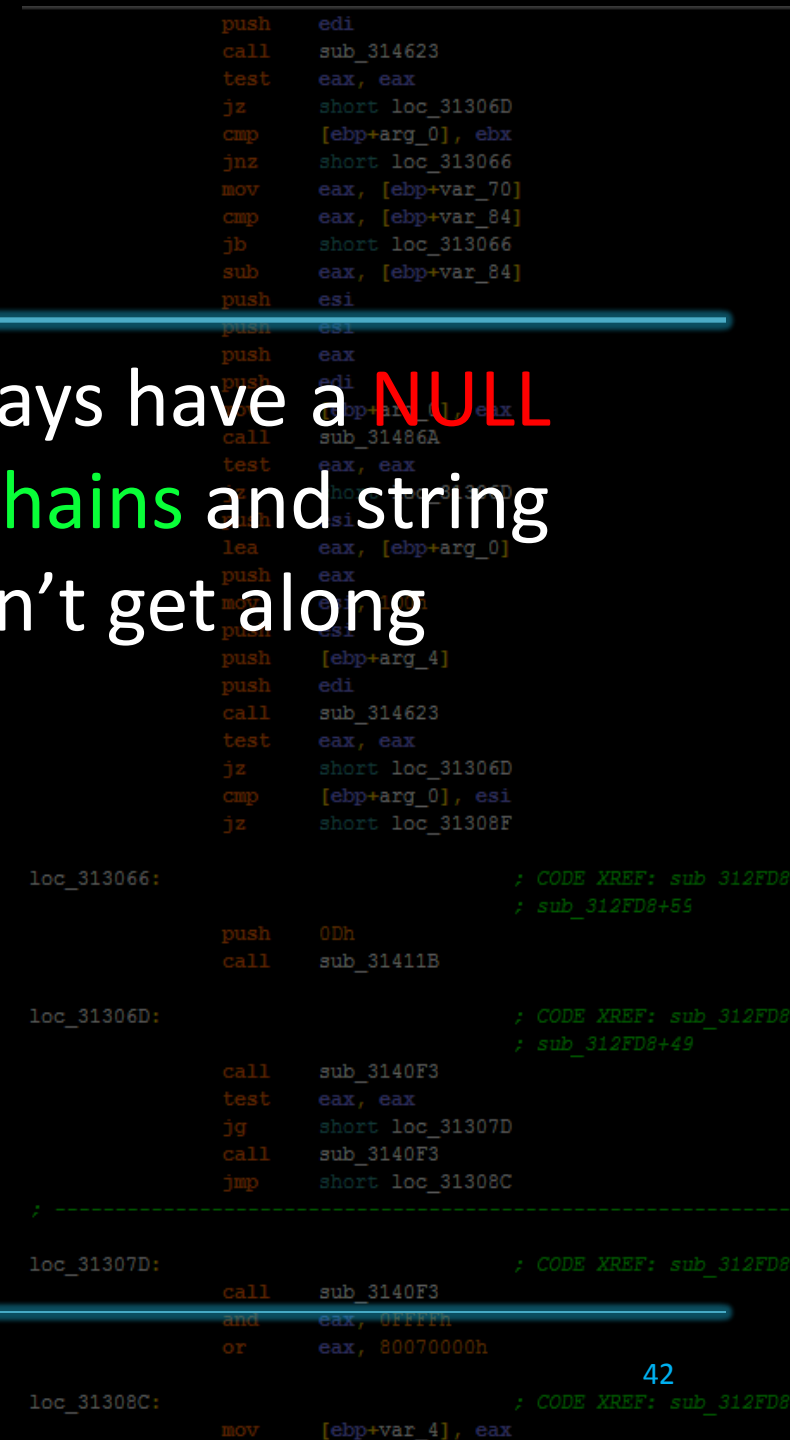
7 (nibbles) * 4 (bits) = 28

2^{28} bruteforce

0.0000000037% exploit reliability!

x86_64 Addresses

- 64bit addresses almost always have a NULL upper byte, meaning ROP chains and string functions (eg `strncpy`) don't get along



x86_64 Addresses

```
doom@upwn64:~$ cat /proc/self/maps
00400000-0040b000 r-xp 00000000 08:01 790596    /bin/cat
0060a000-0060b000 r--p 0000a000 08:01 790596    /bin/cat
0060b000-0060c000 rw-p 0000b000 08:01 790596    /bin/cat
...
7fc6a4788000-7fc6a4943000 r-xp 00000000 08:01 922887  libc-2.19.so
7fc6a4943000-7fc6a4b42000 ---p 001bb000 08:01 922887  libc-2.19.so
7fc6a4b42000-7fc6a4b46000 r--p 001ba000 08:01 922887  libc-2.19.so
7fc6a4b46000-7fc6a4b48000 rw-p 001be000 08:01 922887  libc-2.19.so
...
```

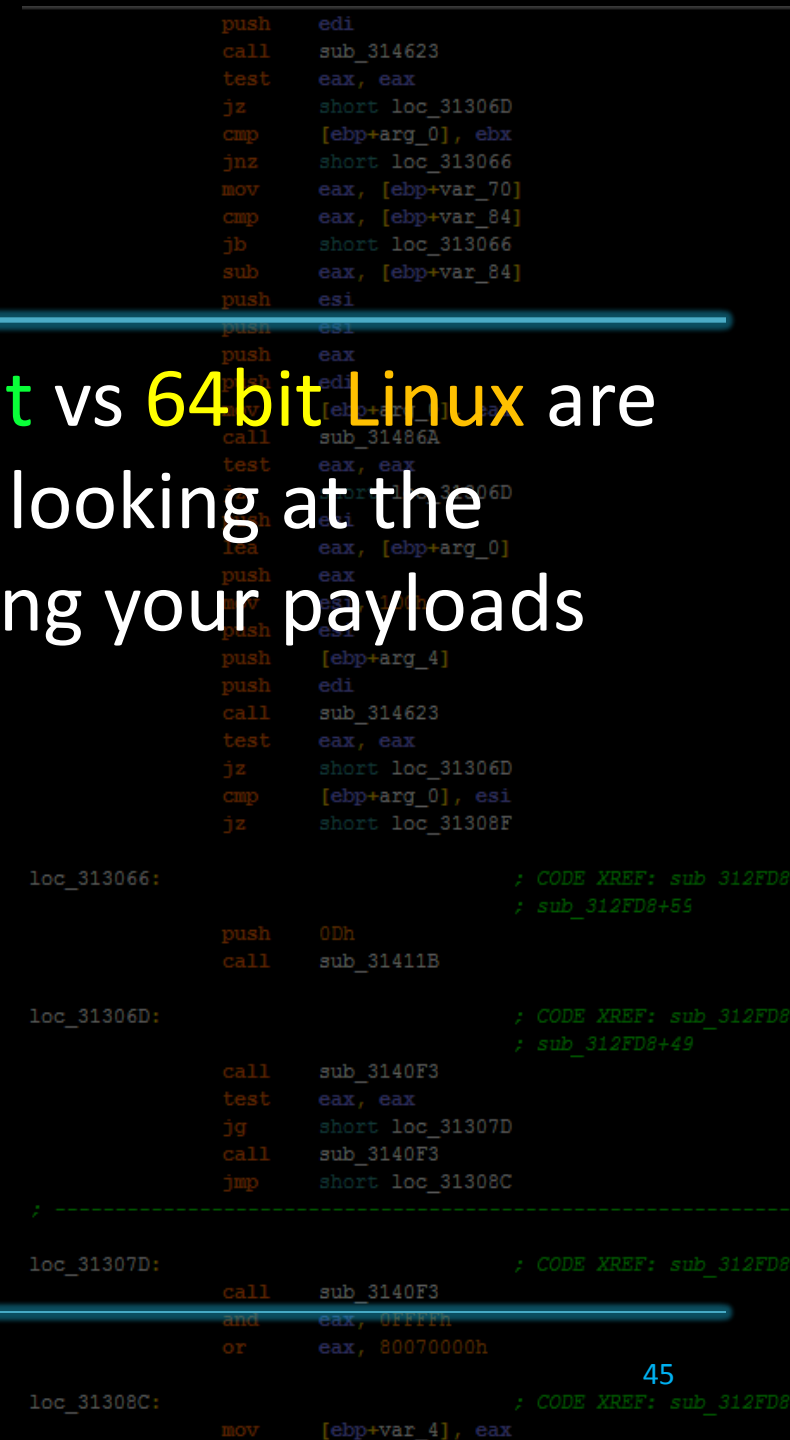
x86_64 Addresses

```
doom@upwn64:~$ cat /proc/self/maps
00400000-0040b000 r-xp 00000000 08:01 790596    /bin/cat
0060a000-0060b000 r--p 0000a000 08:01 790596    /bin/cat
0060b000-0060c000 rw-p 0000b000 08:01 790596    /bin/cat
...
7fc6a4788000-7fc6a4943000 r-xp 00000000 08:01 922887  libc-2.19.so
7fc6a4943000-7fc6a4b42000 ---p 001bb000 08:01 922887  libc-2.19.so
7fc6a4b42000-7fc6a4b46000 r--p 001ba000 08:01 922887  libc-2.19.so
7fc6a4b46000-7fc6a4b48000 rw-p 001be000 08:01 922887  libc-2.19.so
...


0x0000000000400000 - 0x000000000040b000
0x00007fc6a4788000 - 0x00007fc6a4943000
```

These are 64bit addresses, so yes there's plenty of space for nulls

x86_64 Syscalls

- The syscall numbers in 32bit vs 64bit Linux are different, so be sure you're looking at the respective table when writing your payloads



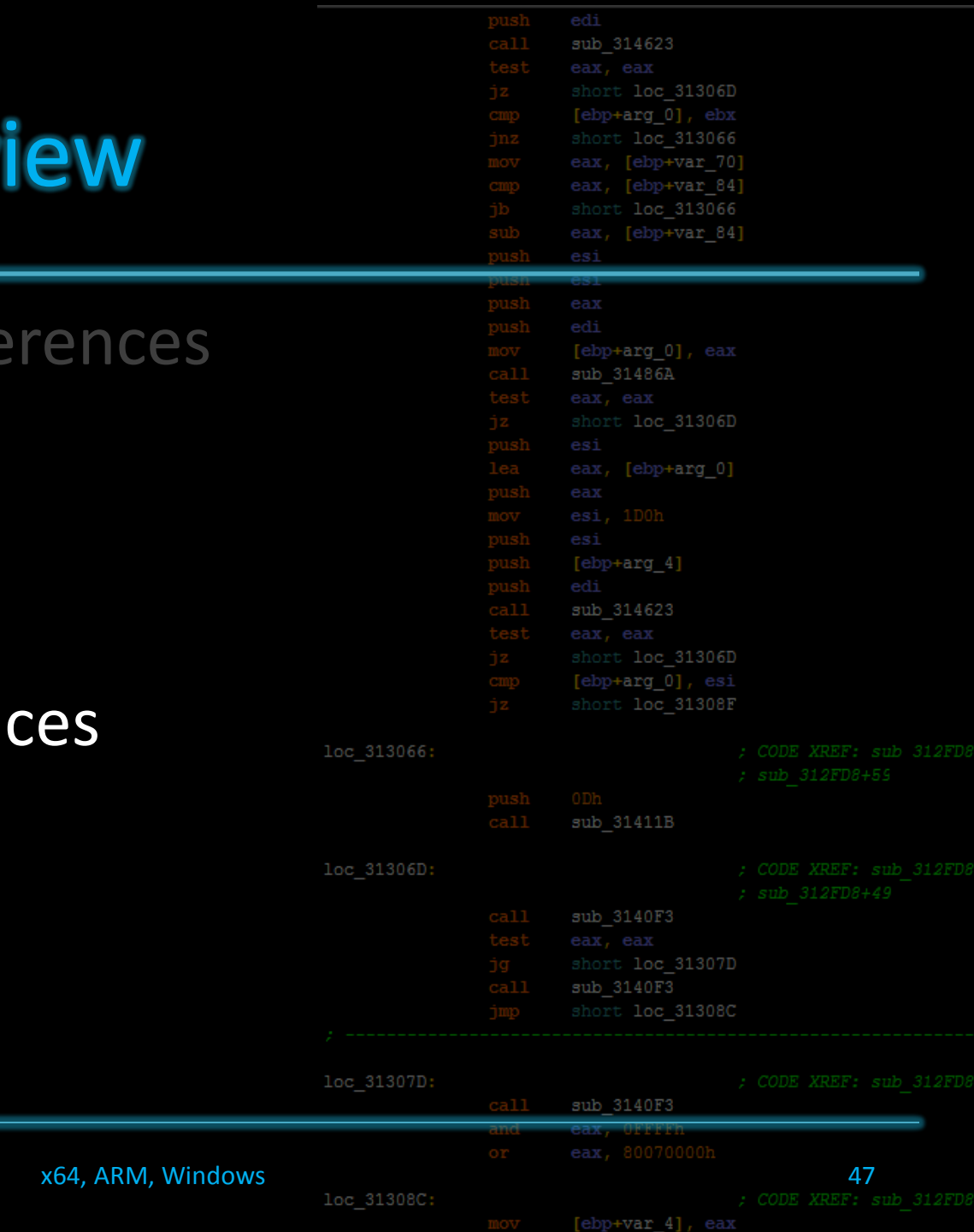
x86_64 Syscalls

- The syscall numbers in 32bit vs 64bit Linux are different, so be sure you're looking at the respective table when writing your payloads

```
exec syscall on 32bit: 0x0b
exec syscall on 64bit: 0x3b
```

Lecture Overview

- Architecture Differences
  - x86
  - x86_64
  - ARM
- Platform Differences
  - Windows



ARM Overview

- ARM is a 32bit RISC instruction set built for low power devices
  - Has a ’16bit’ THUMB mode

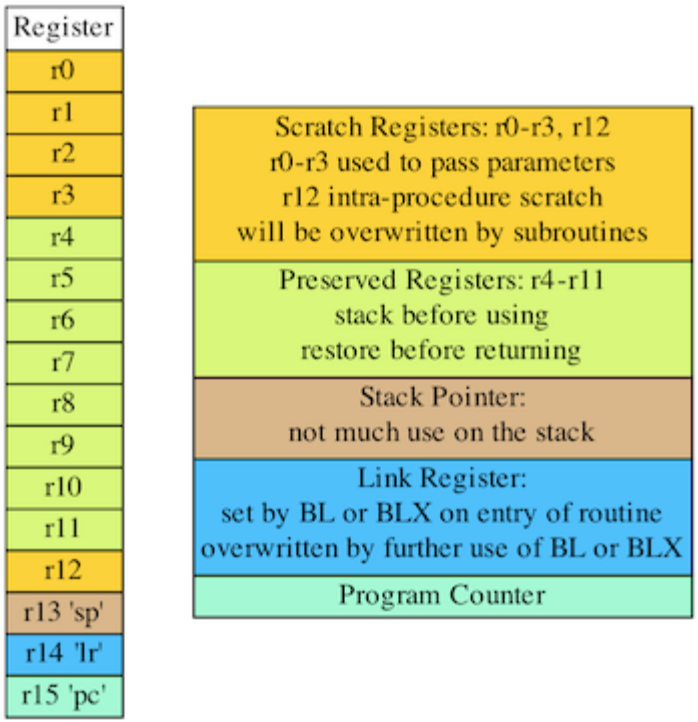

```
push    edi
call    sub_314623
test    eax, eax
jz      short loc_31306D
cmp     [ebp+arg_0], ebx
jnz     short loc_313066
mov     eax, [ebp+var_70]
cmp     eax, [ebp+var_84]
jb      short loc_313066
sub     eax, [ebp+var_84]
push    esi
push    esi
push    eax
push    edi
mov     esi, [ebp+arg_0]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

ARM Overview

- ARM is a 32bit RISC instruction set built for low power devices
  - Has a ’16bit’ THUMB mode

- Used on your phone, tablet, raspberry pi, other small or mobile devices
  - ‘low power’

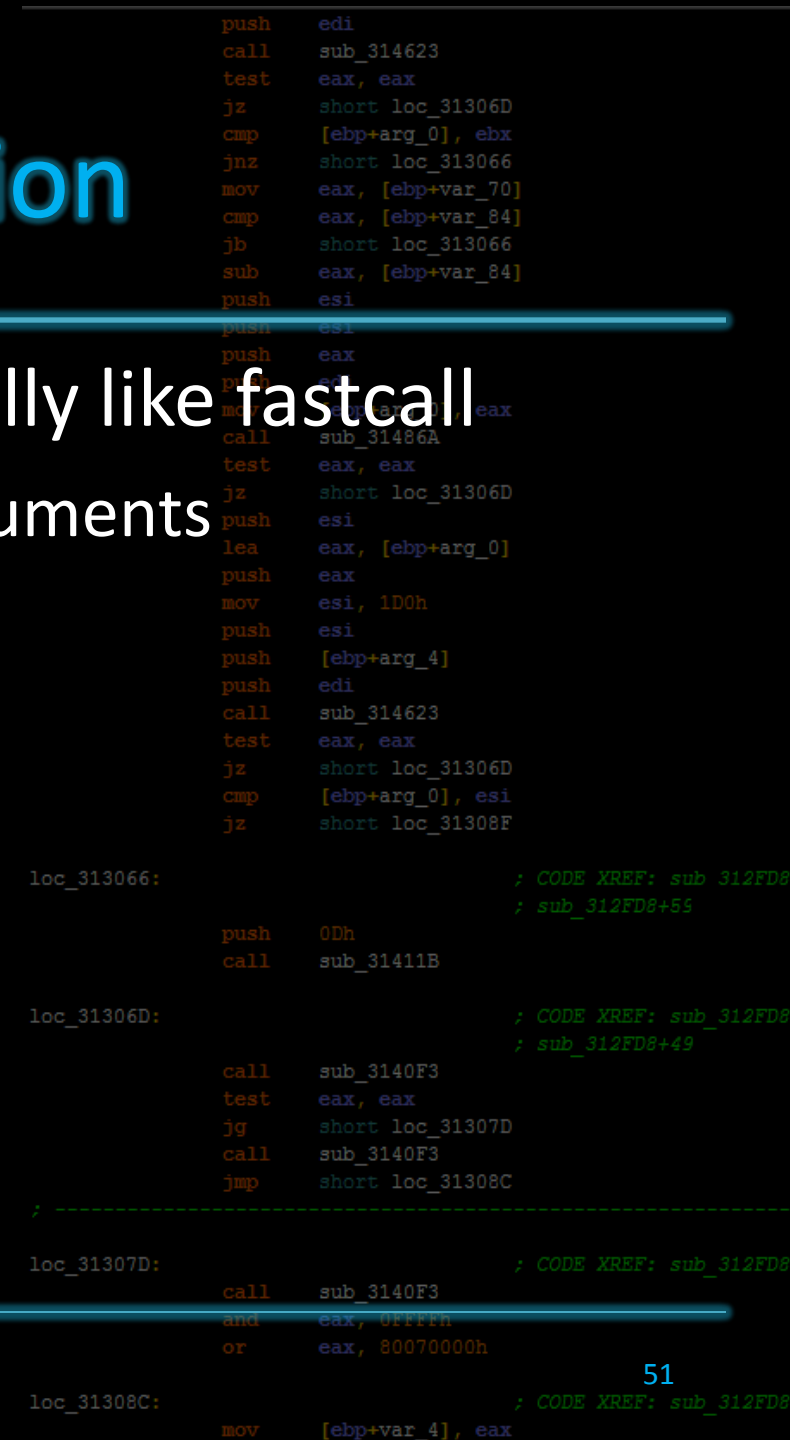
ARM Registers



Register Use in the ARM Procedure Call Standard

ARM Calling Convention

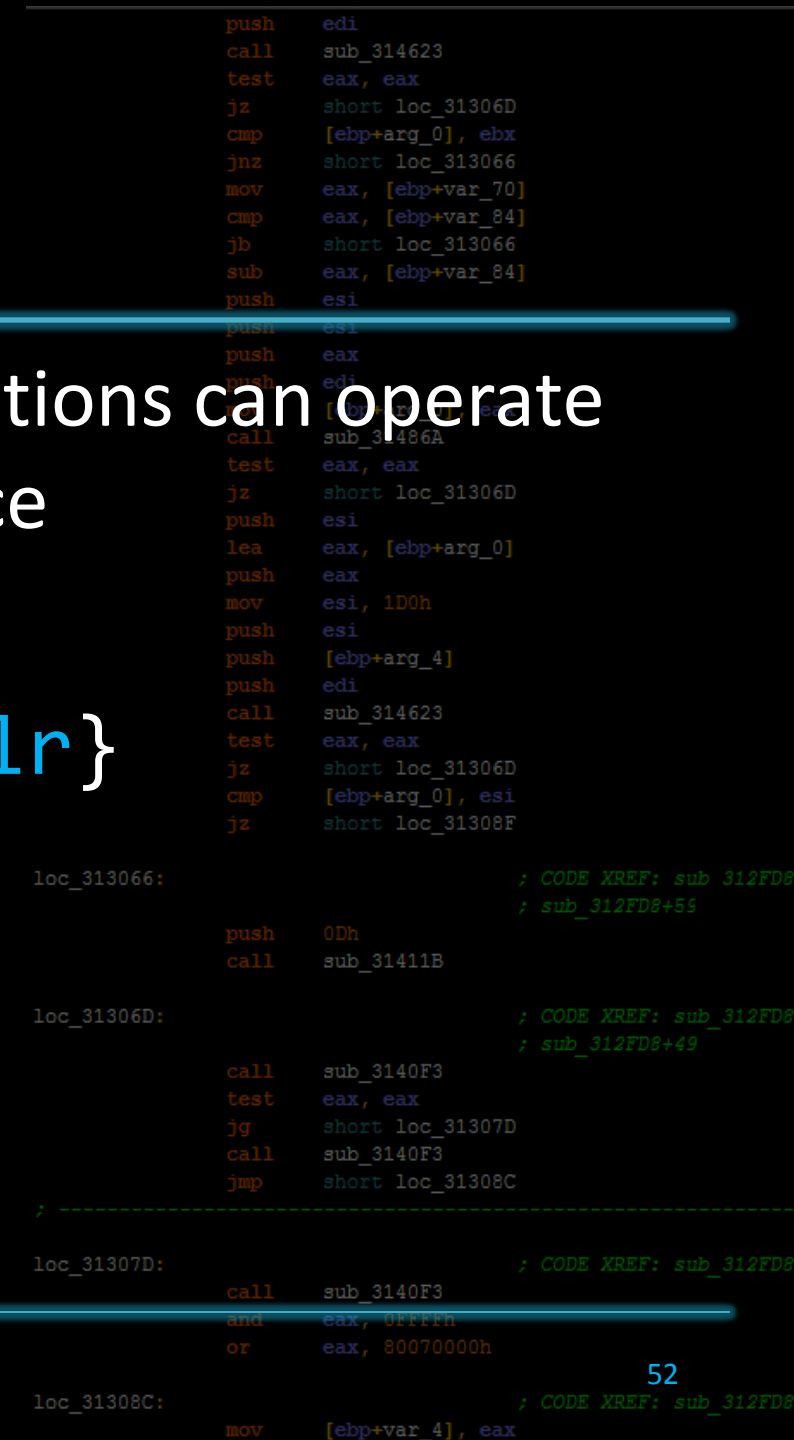
- Calling convention is basically like fastcall
  - r0-r3 hold your function arguments



ARM Assembly

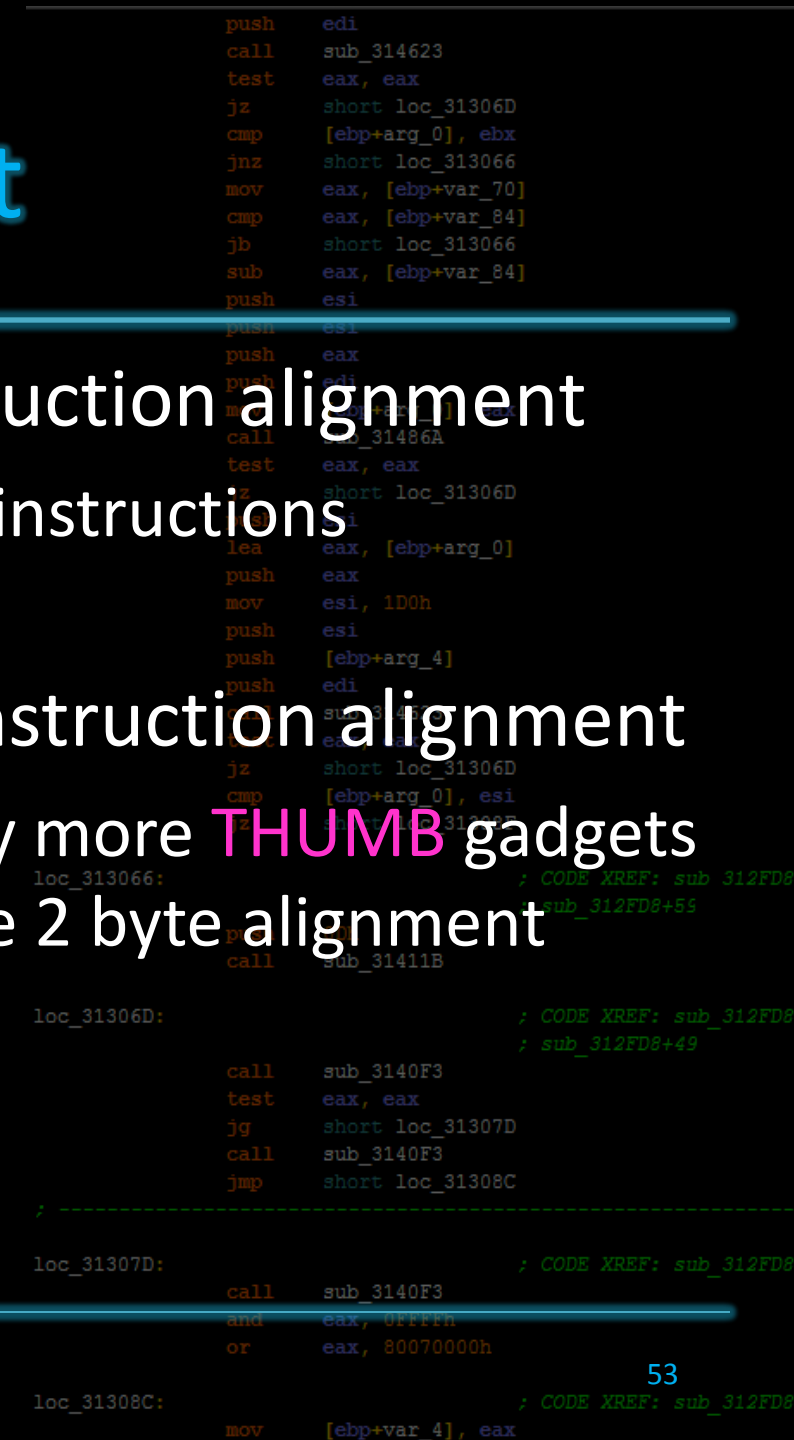
- Some ARM/THUMB instructions can operate on multiple registers at once

```
pop {r4, r5, r6, lr}
...
```



Instruction Alignment

- ARM mode has 4 byte instruction alignment
  - Can't jump in the middle of instructions
- THUMB mode has 2 byte instruction alignment
  - When ROPing there's usually more THUMB gadgets that will be of use due to the 2 byte alignment

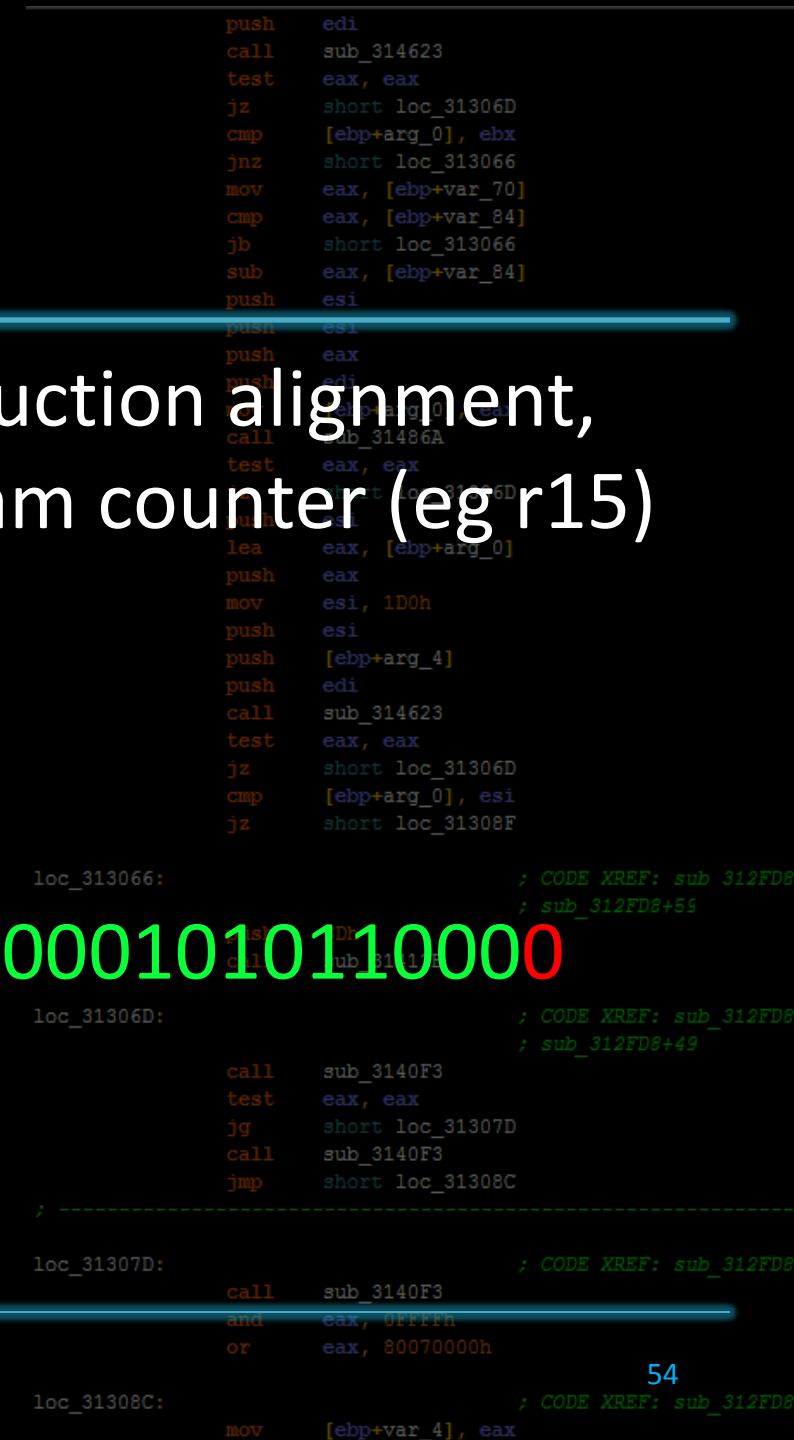


An Interesting Bit

- Because of 2 & 4 byte instruction alignment, the lowest bit of the program counter (eg r15) will never be set

0x080462B0

00001000000001000110001010110000



An Interesting Bit

- Because of 2 & 4 byte instruction alignment, the lowest bit of the program counter (eg r15) will never be set

0x080462B0

00001000000001000110001010110000

This bit is re-purposed to tell the processor if we are in THUMB mode or ARM mode

An Interesting Bit

r15 = 0x080462B0

= 00001000000001000110001010110000

Interpret bytes at 0x080462B0 as ARM

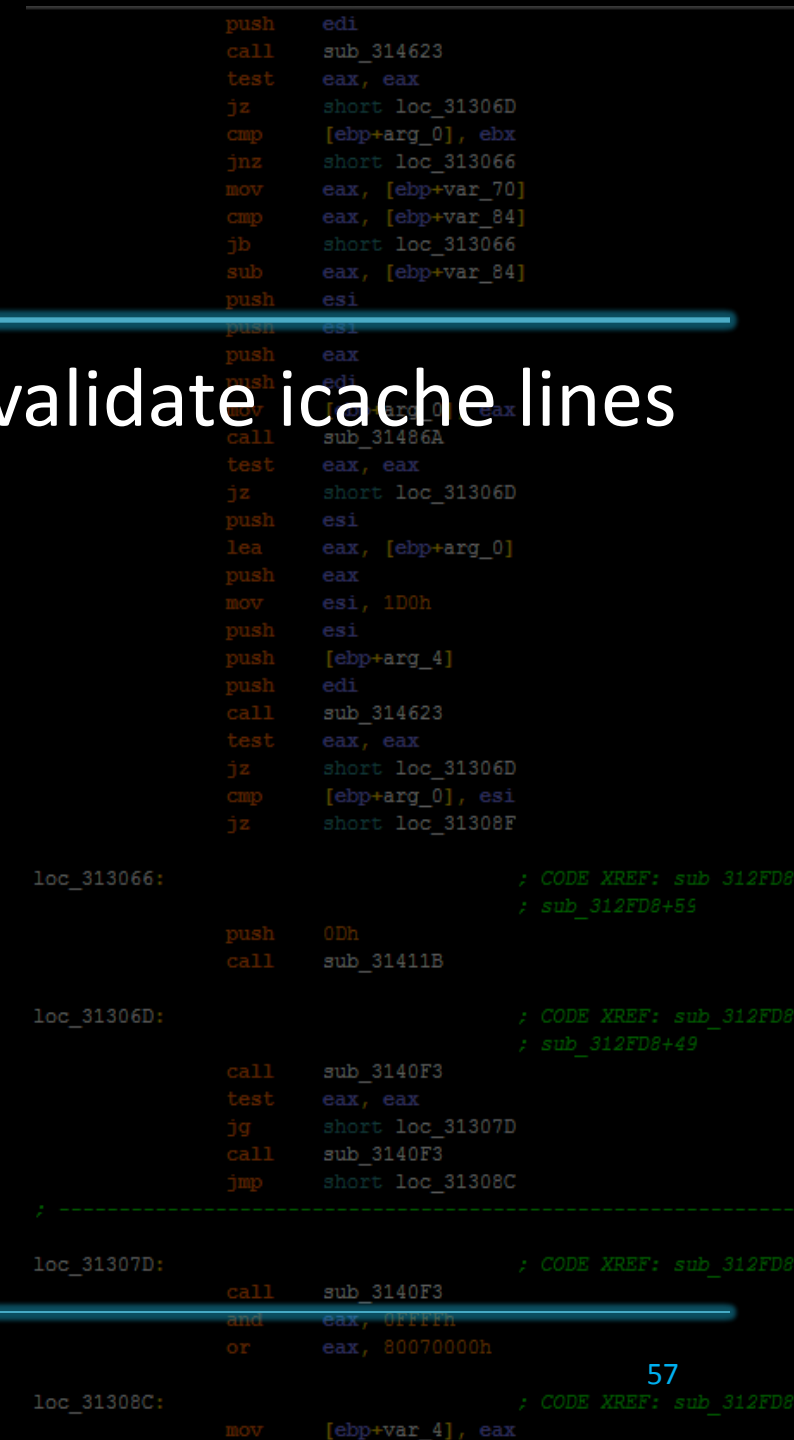
r15 = 0x080462B1

= 00001000000001000110001010110001

Interpret bytes at 0x080462B0 as THUMB

Caching

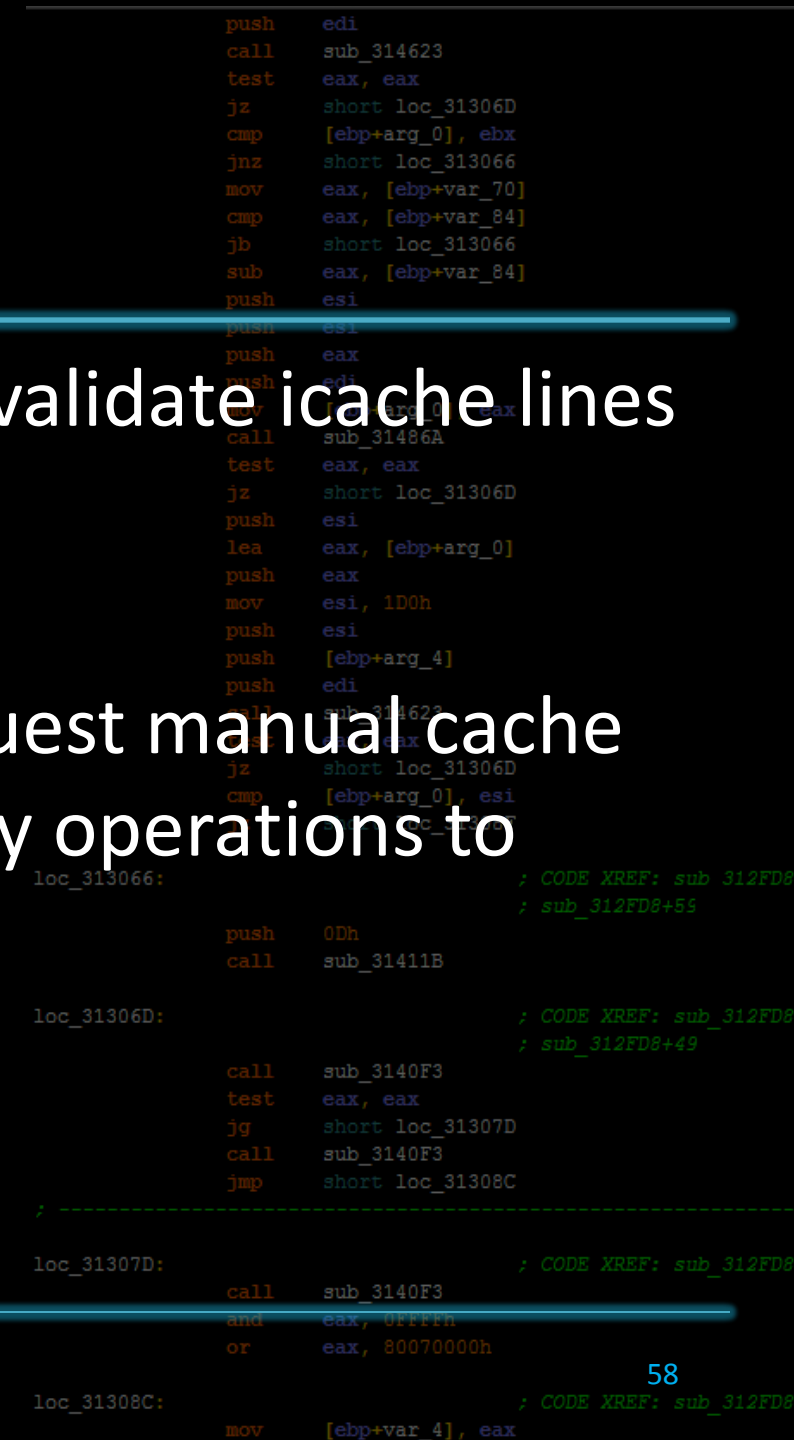
- In x86 the processor will invalidate icache lines if the line is written to



Caching

- In x86 the processor will invalidate icache lines if the line is written to

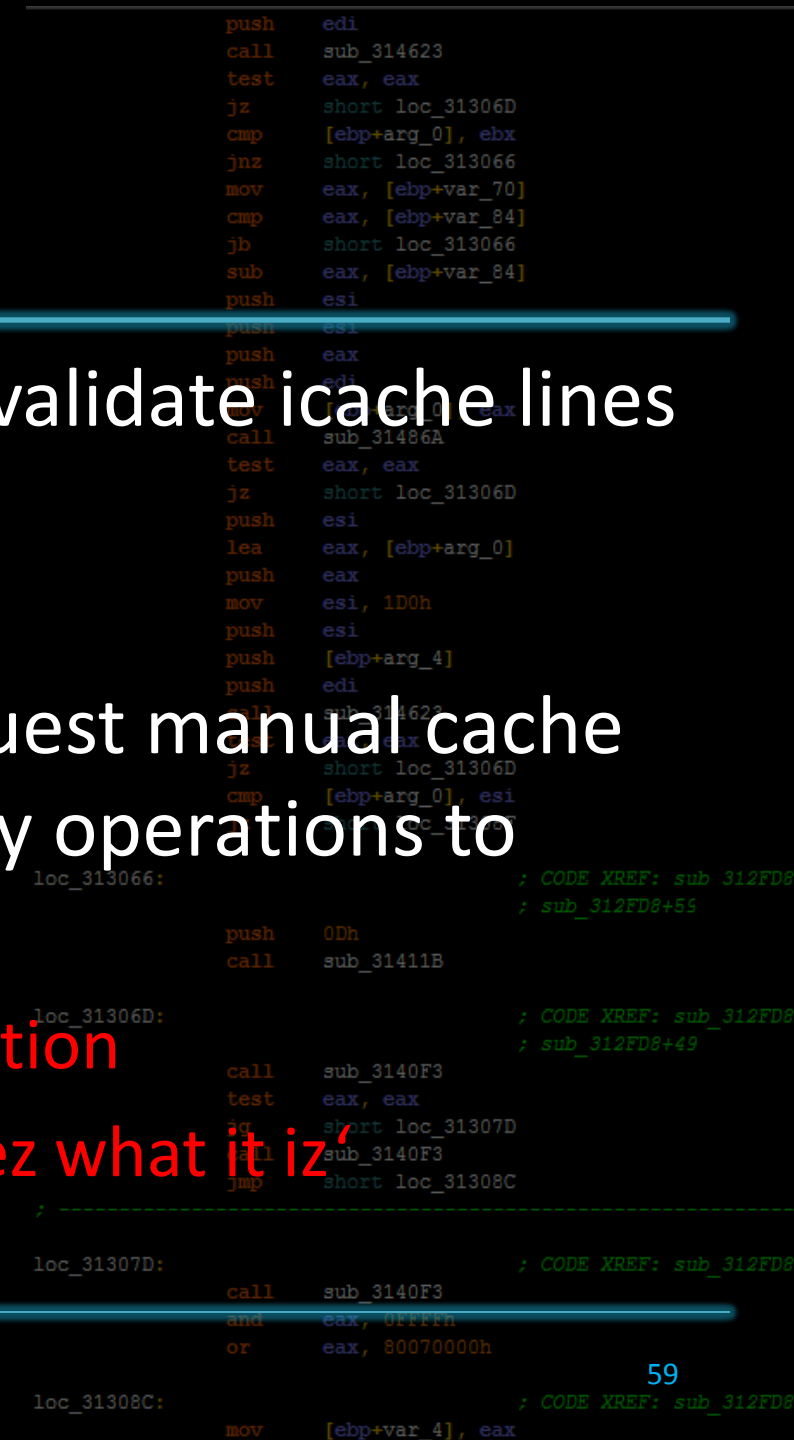
- With ARM you have to request manual cache flushes, or do large memory operations to flush the cache naturally



Caching

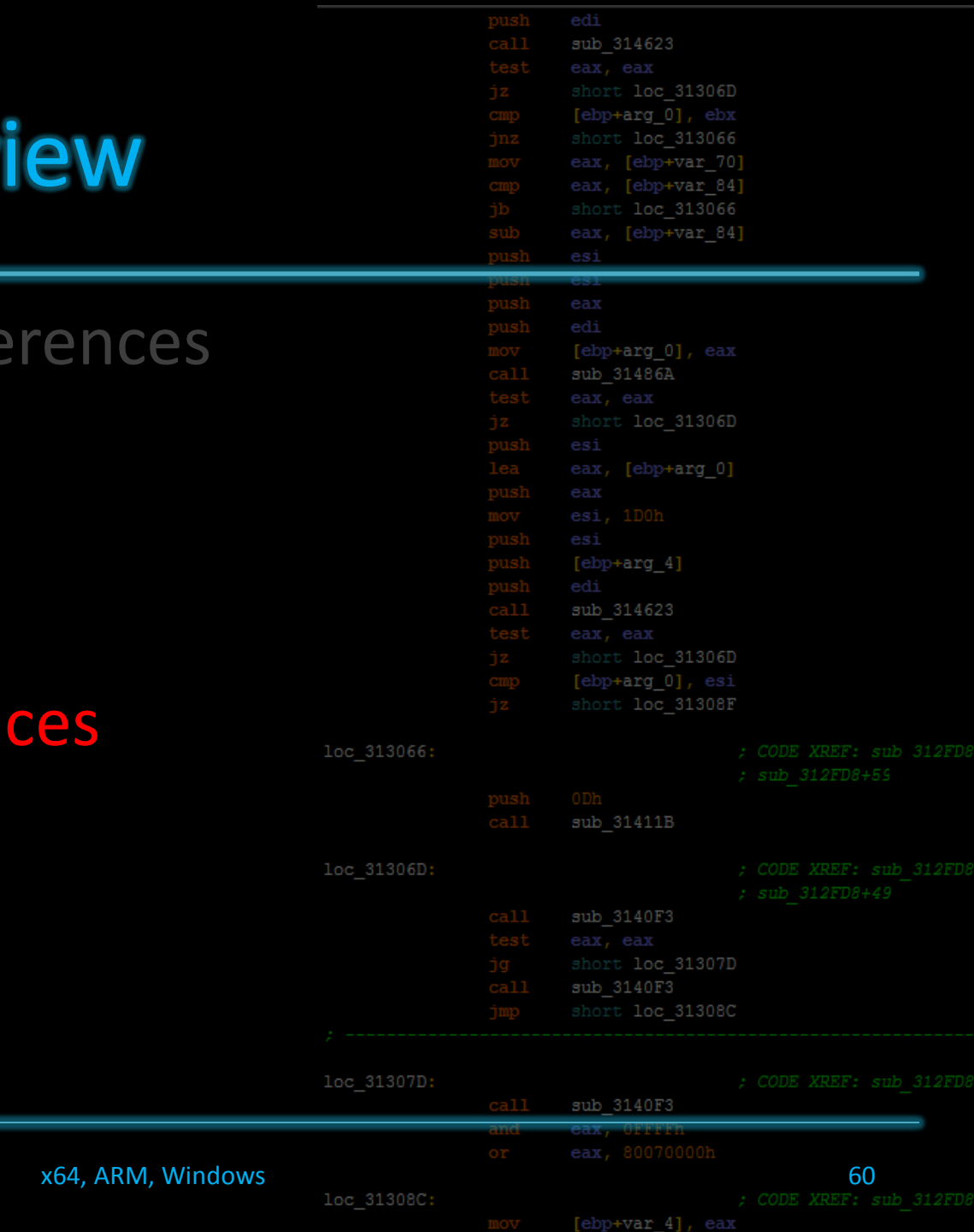
- In x86 the processor will invalidate icache lines if the line is written to

- With ARM you have to request manual cache flushes, or do large memory operations to flush the cache naturally
  - Can get annoying in exploitation
  - 'what you seez, may not beez what it iz'



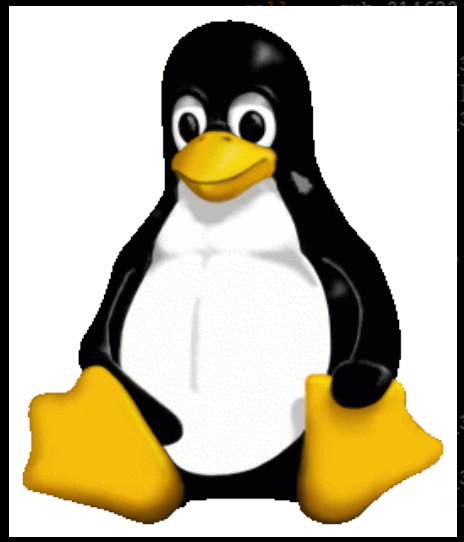
Lecture Overview

- Architecture Differences
  - x86
  - x86_64
  - ARM
- Platform Differences
  - Windows



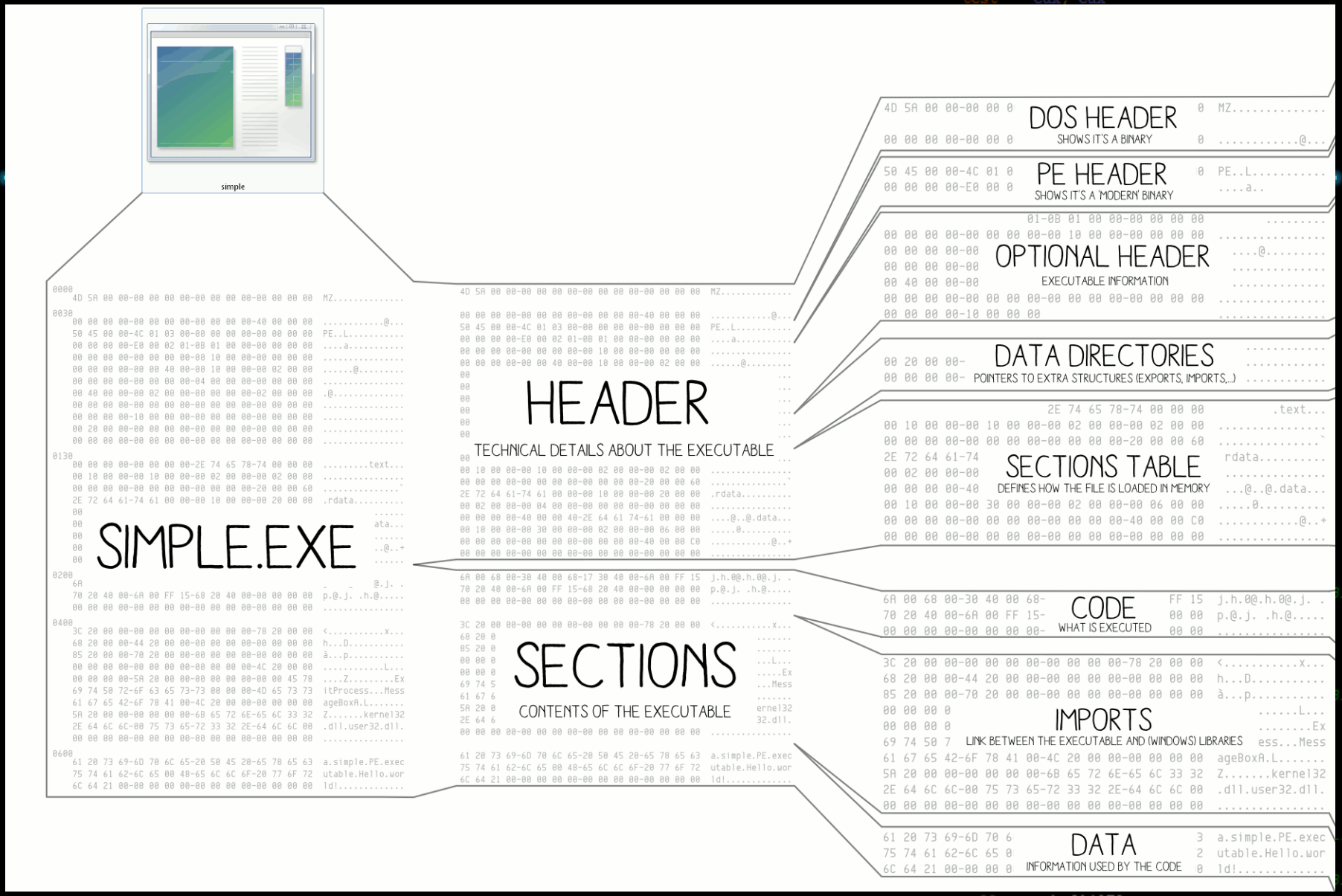
Windows vs Linux

- Almost all the vulnerability classes and exploitation techniques you have learned in this course will apply directly to Windows



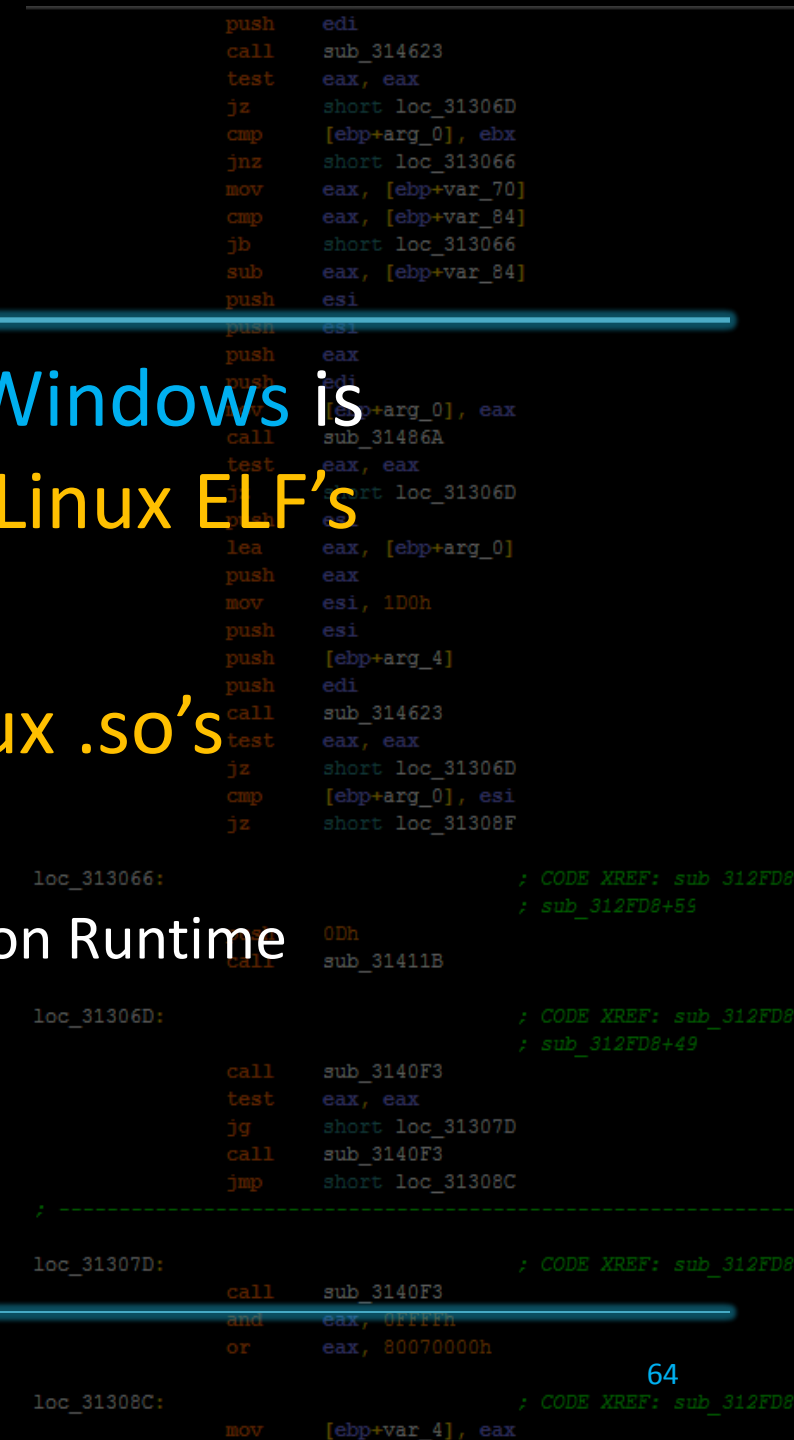
Windows Basics

- The executable format on Windows is obviously .EXE's instead of Linux ELF's


simple
SIMPLE.EXE
HEADER
TECHNICAL DETAILS ABOUT THE EXECUTABLE
SECTIONS
CONTENTS OF THE EXECUTABLE
DOS HEADER
SHOWS IT'S A BINARY
PE HEADER
SHOWS IT'S A 'MODERN' BINARY
OPTIONAL HEADER
EXECUTABLE INFORMATION
DATA DIRECTORIES
POINTERS TO EXTRA STRUCTURES (EXPORTS, IMPORTS,...)
SECTIONS TABLE
DEFINES HOW THE FILE IS LOADED IN MEMORY
CODE
WHAT IS EXECUTED
IMPORTS
LINK BETWEEN THE EXECUTABLE AND (WINDOWS) LIBRARIES
DATA
INFORMATION USED BY THE CODE

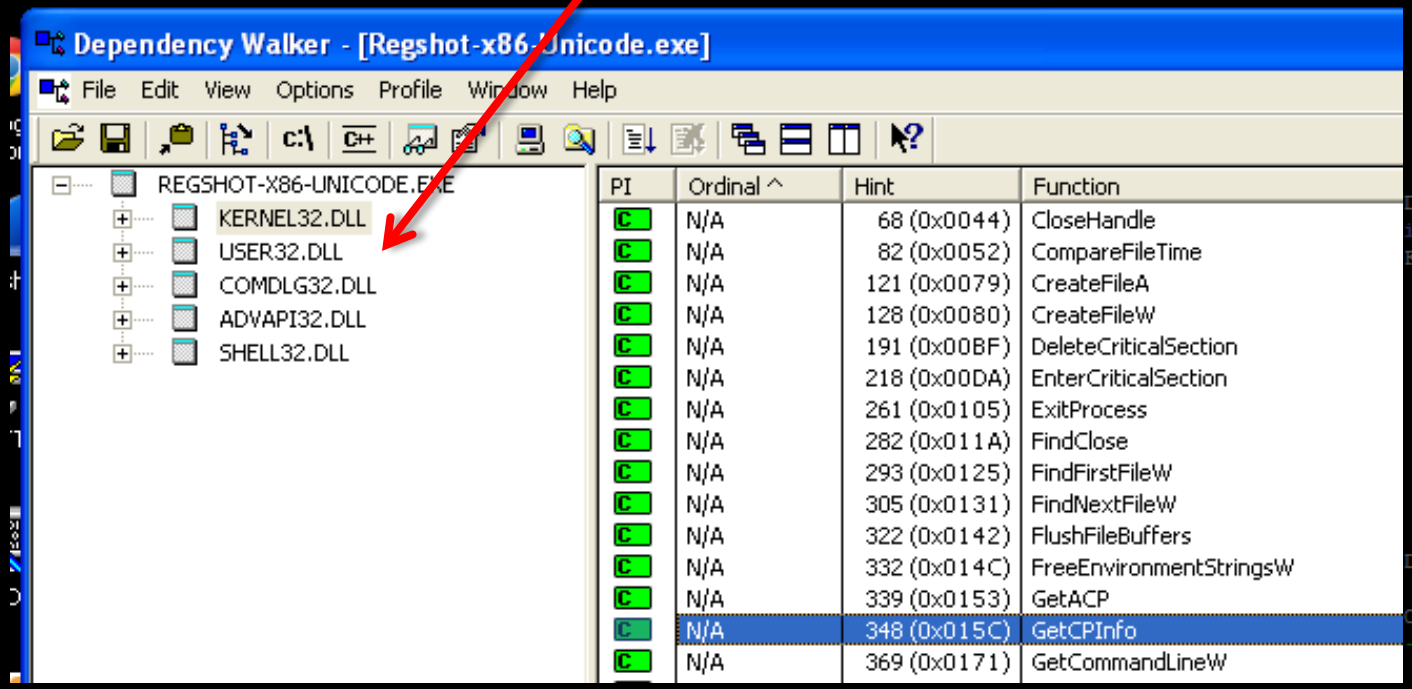
Windows Basics

- The executable format on Windows is obviously .EXE's instead of Linux ELF's
- Libraries are .DLL's, like Linux .so's
  - eg: MSVCRT.dll is like libc
    - Microsoft Visual C(++) Common Runtime



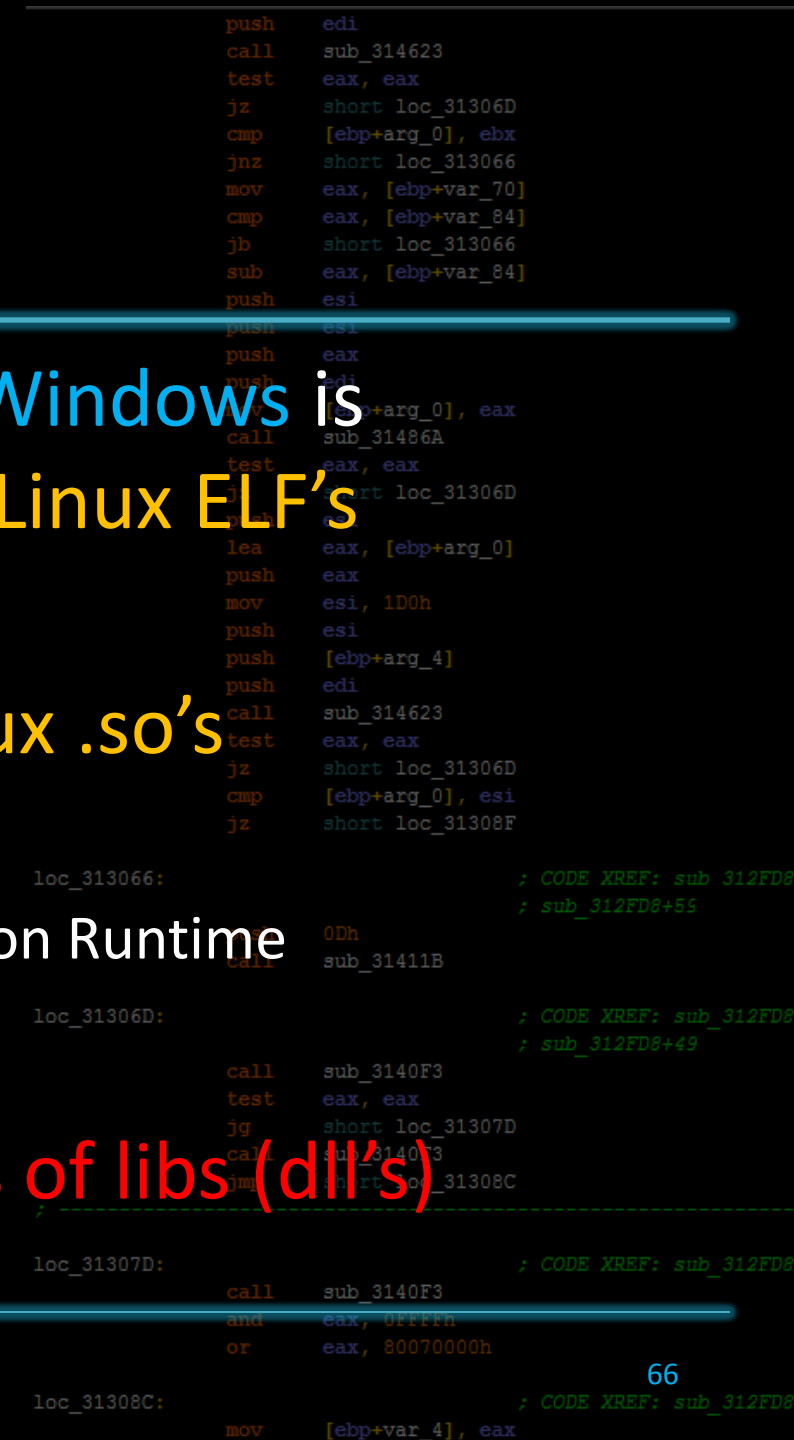
Windows Basics

Loaded DLL's



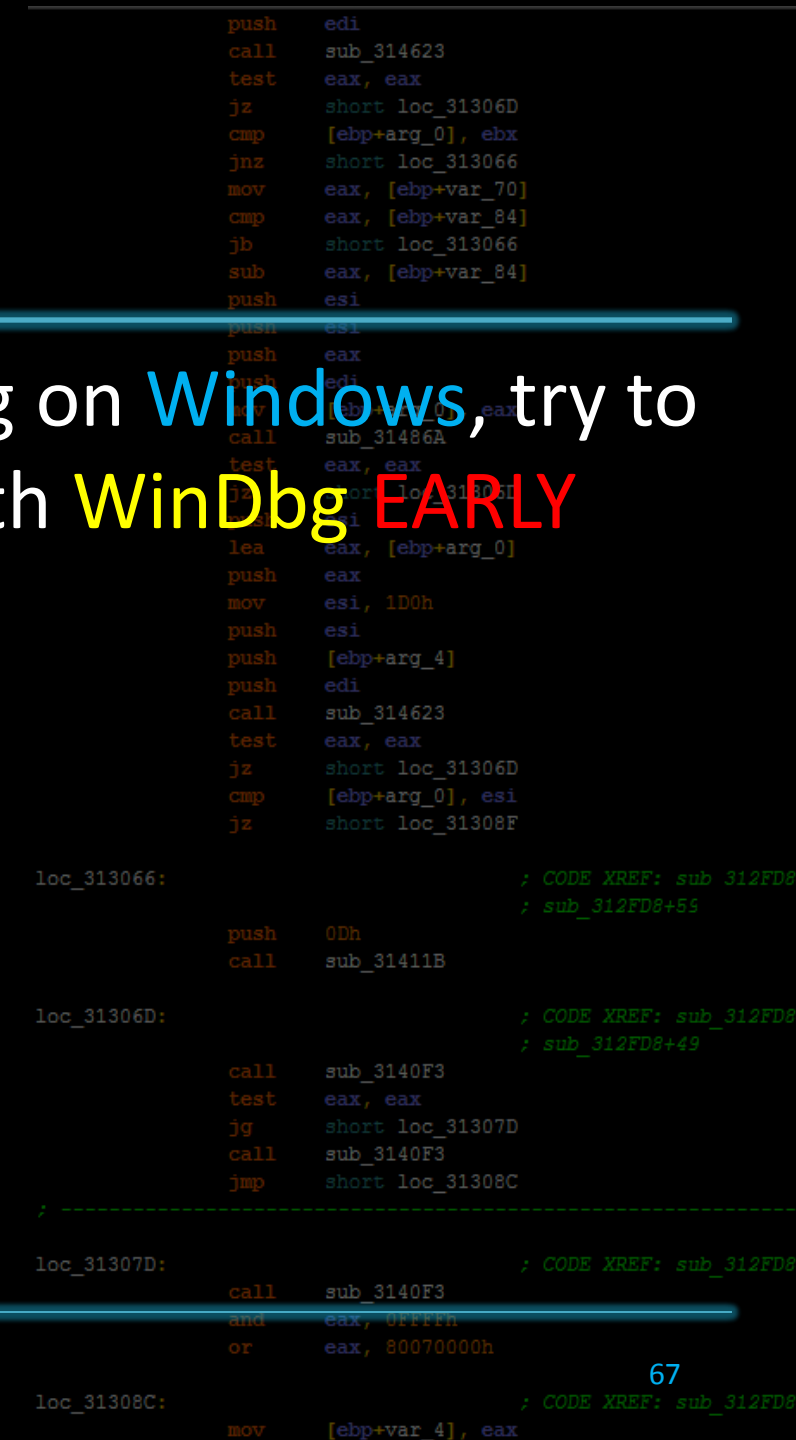
Windows Basics

- The executable format on Windows is obviously .EXE's instead of Linux ELF's
- Libraries are .DLL's, like Linux .so's
  - eg: MSVCRT.dll is like libc
    - Microsoft Visual C(++) Common Runtime
- A process usually loads lots of libs (dll's)



Windows Debuggers

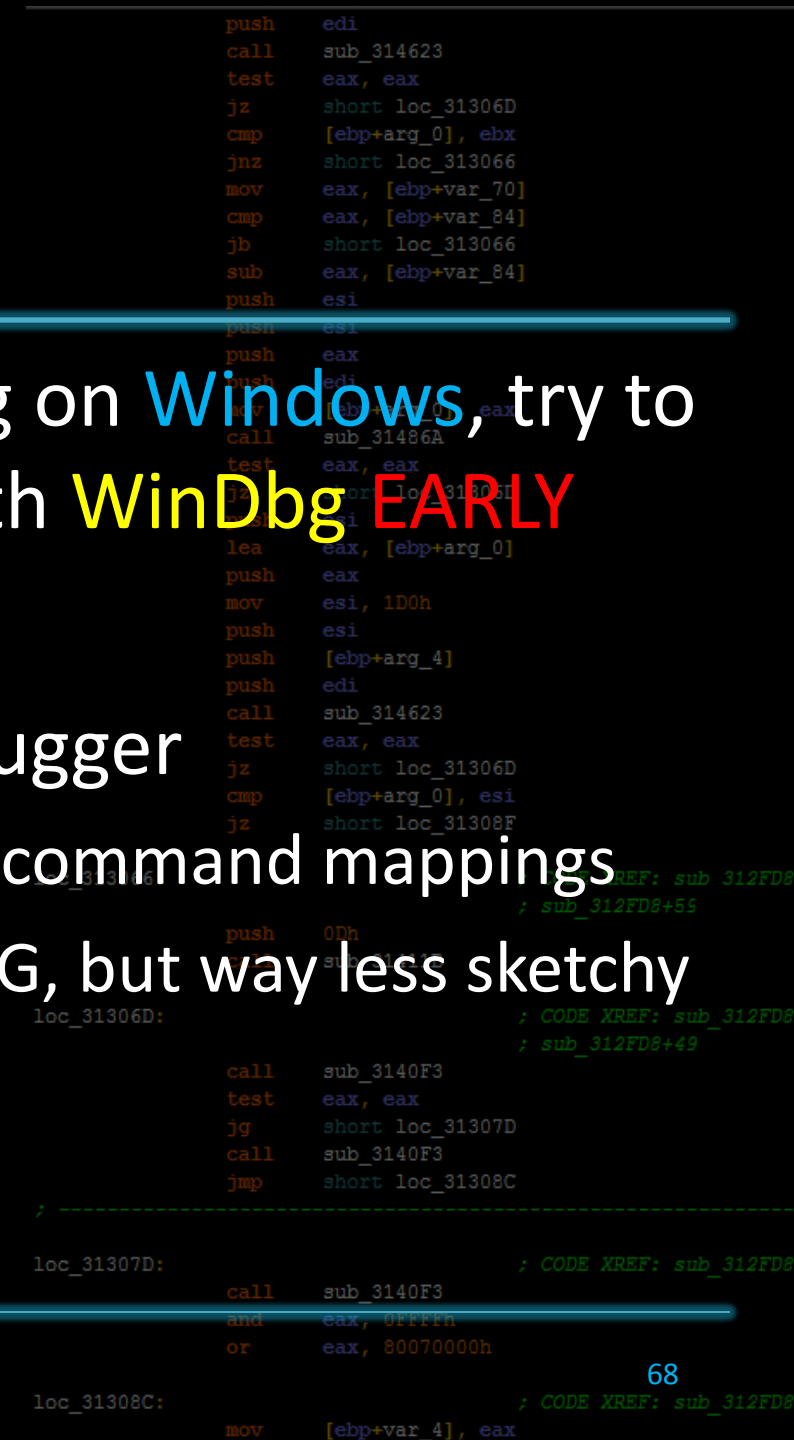
- If you're going to get rolling on Windows, try to pick up skills debugging with WinDbg EARLY



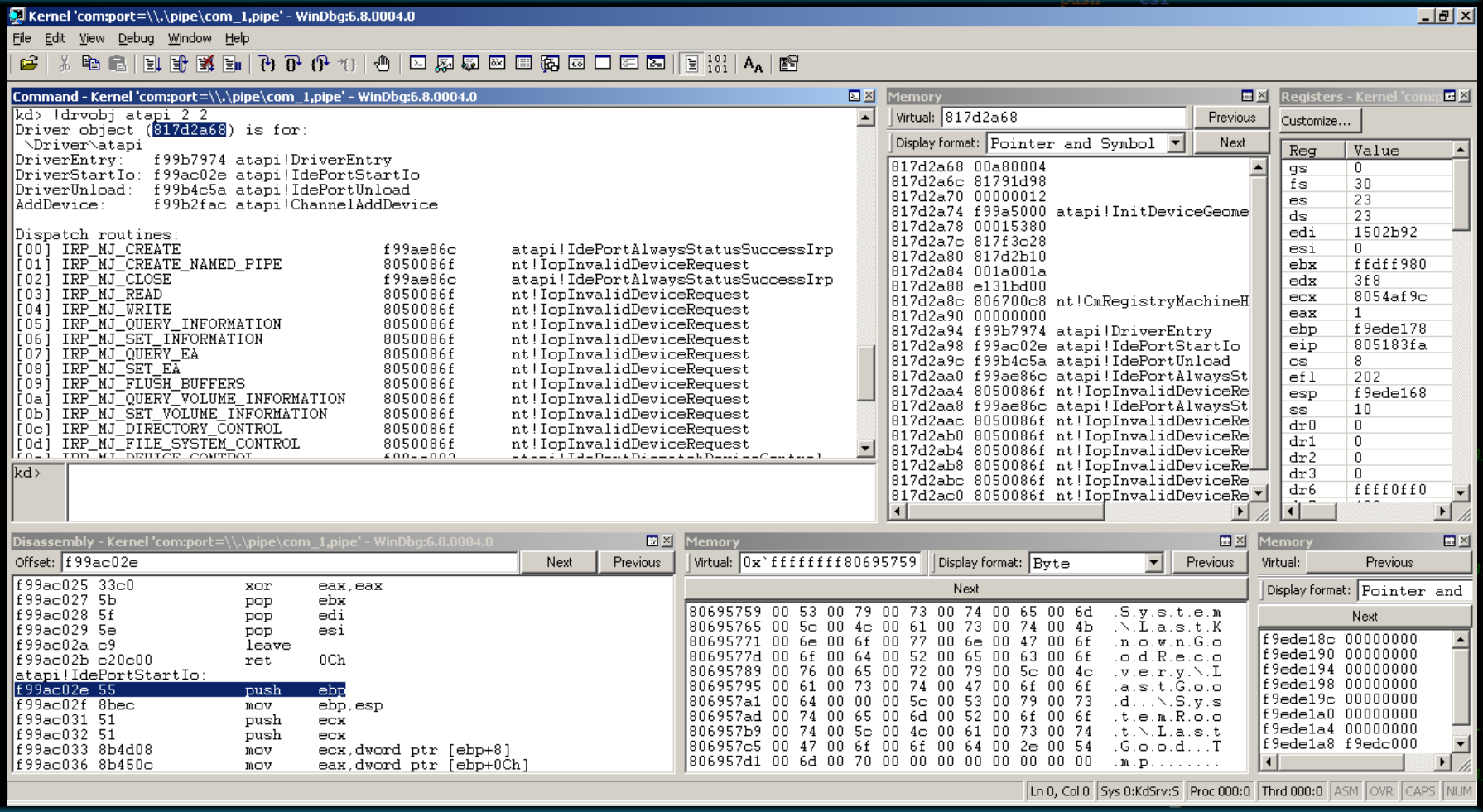
Windows Debuggers

- If you're going to get rolling on Windows, try to pick up skills debugging with WinDbg EARLY

- WinDBG is Microsoft's debugger
  - Basically GDB with different command mappings
  - Not as convenient as OllyDBG, but way less sketchy
  - Best 64bit debugger

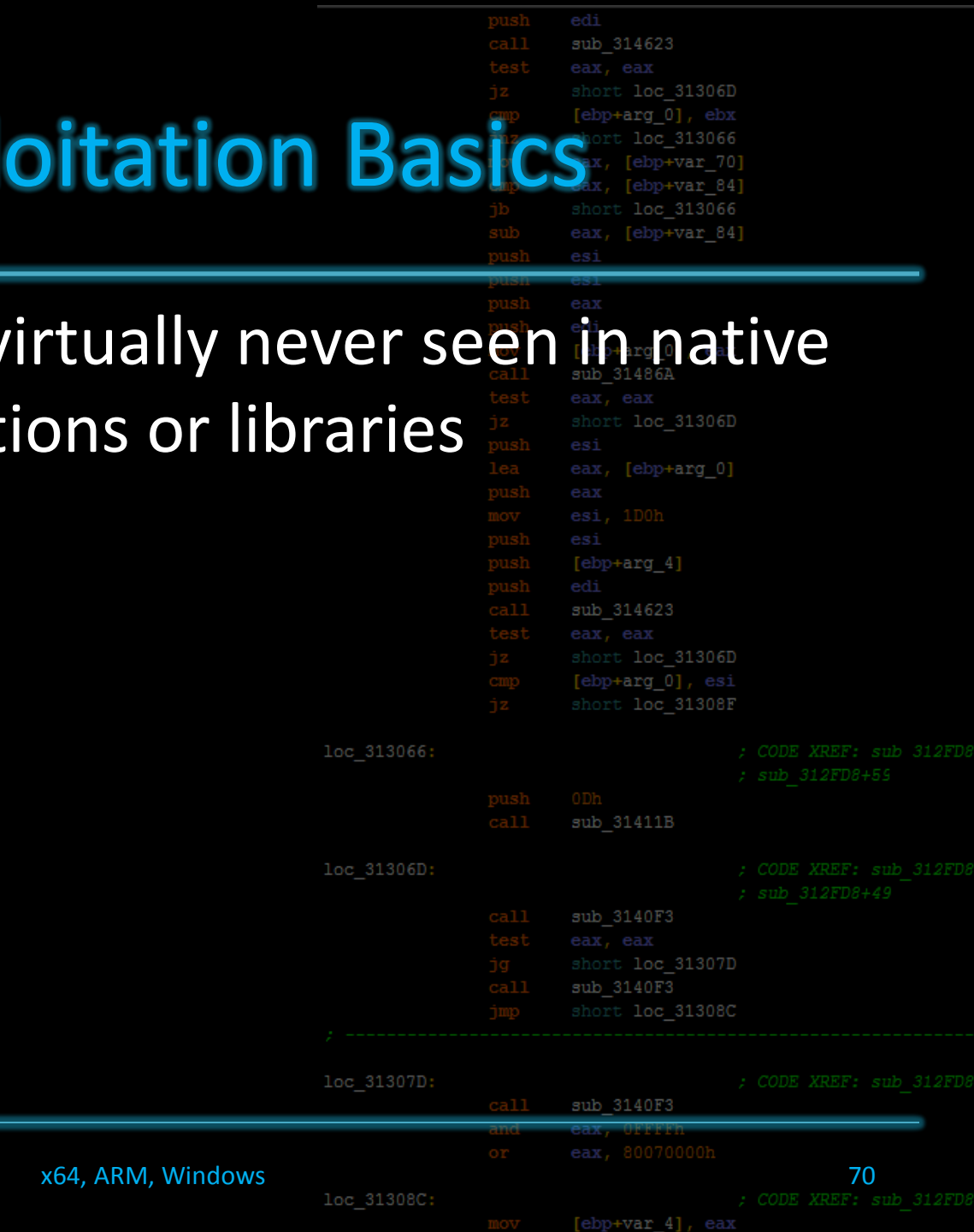


WinDbg


Kernel 'com:port=\\.\pipe\com_1,pipe' - WinDbg:6.8.0004.0
File Edit View Debug Window Help
Command - Kernel 'com:port=\\.\pipe\com_1,pipe' - WinDbg:6.8.0004.0
kd> !drvobj atapi 2 2
Driver object (817d2a68) is for:
\Driver\atapi
DriverEntry: f99b7974 atapi!DriverEntry
DriverStartIo: f99ac02e atapi!IdePortStartIo
DriverUnload: f99b4c5a atapi!IdePortUnload
AddDevice: f99b2fac atapi!ChannelAddDevice
Dispatch routines:
[00] IRP_MJ_CREATE f99ae86c atapi!IdePortAlwaysStatusSuccessIrp
[01] IRP_MJ_CREATE_NAMED_PIPE 8050086f nt!IopInvalidDeviceRequest
[02] IRP_MJ_CLOSE f99ae86c atapi!IdePortAlwaysStatusSuccessIrp
[03] IRP_MJ_READ 8050086f nt!IopInvalidDeviceRequest
[04] IRP_MJ_WRITE 8050086f nt!IopInvalidDeviceRequest
[05] IRP_MJ_QUERY_INFORMATION 8050086f nt!IopInvalidDeviceRequest
[06] IRP_MJ_SET_INFORMATION 8050086f nt!IopInvalidDeviceRequest
[07] IRP_MJ_QUERY_EA 8050086f nt!IopInvalidDeviceRequest
[08] IRP_MJ_SET_EA 8050086f nt!IopInvalidDeviceRequest
[09] IRP_MJ_FLUSH_BUFFERS 8050086f nt!IopInvalidDeviceRequest
[0a] IRP_MJ_QUERY_VOLUME_INFORMATION 8050086f nt!IopInvalidDeviceRequest
[0b] IRP_MJ_SET_VOLUME_INFORMATION 8050086f nt!IopInvalidDeviceRequest
[0c] IRP_MJ_DIRECTORY_CONTROL 8050086f nt!IopInvalidDeviceRequest
[0d] IRP_MJ_FILE_SYSTEM_CONTROL 8050086f nt!IopInvalidDeviceRequest
kd>
Memory
Virtual: 817d2a68
Previous
Display format: Pointer and Symbol
Next
817d2a68 00a80004
817d2a6c 81791d98
817d2a70 00000012
817d2a74 f99a5000 atapi!InitDeviceGeome
817d2a78 00015380
817d2a7c 817f3c28
817d2a80 817d2b10
817d2a84 001a001a
817d2a88 e131bd00
817d2a8c 806700c8 nt!CmRegistryMachineH
817d2a90 00000000
817d2a94 f99b7974 atapi!DriverEntry
817d2a98 f99ac02e atapi!IdePortStartIo
817d2a9c f99b4c5a atapi!IdePortUnload
817d2aa0 f99ae86c atapi!IdePortAlwaysSt
817d2aa4 8050086f nt!IopInvalidDeviceRe
817d2aa8 f99ae86c atapi!IdePortAlwaysSt
817d2aac 8050086f nt!IopInvalidDeviceRe
817d2ab0 8050086f nt!IopInvalidDeviceRe
817d2ab4 8050086f nt!IopInvalidDeviceRe
817d2ab8 8050086f nt!IopInvalidDeviceRe
817d2abc 8050086f nt!IopInvalidDeviceRe
817d2ac0 8050086f nt!IopInvalidDeviceRe
Registers - Kernel 'com:
Customize...
Reg Value
gs 0
fs 30
es 23
ds 23
edi 1502b92
esi 0
ebx ffdff980
edx 3f8
ecx 8054af9c
eax 1
ebp f9ede178
eip 805183fa
cs 8
efl 202
esp f9ede168
ss 10
dr0 0
dr1 0
dr2 0
dr3 0
dr6 ffff0ff0
Disassembly - Kernel 'com:port=\\.\pipe\com_1,pipe' - WinDbg:6.8.0004.0
Offset: f99ac02e
Next
Previous
f99ac025 33c0 xor eax,eax
f99ac027 5b pop ebx
f99ac028 5f pop edi
f99ac029 5e pop esi
f99ac02a c9 leave
f99ac02b c20c00 ret 0Ch
atapi!IdePortStartIo:
f99ac02e 55 push ebp
f99ac02f 8bec mov ebp,esp
f99ac031 51 push ecx
f99ac032 51 push ecx
f99ac033 8b4d08 mov ecx,dword ptr [ebp+8]
f99ac036 8b450c mov eax,dword ptr [ebp+0Ch]
Memory
Virtual: 0x`ffffffff80695759
Display format: Byte
Previous
Next
80695759 00 53 00 79 00 73 00 74 00 65 00 6d .S.y.s.t.e.m
80695765 00 5c 00 4c 00 61 00 73 00 74 00 4b .\.L.a.s.t.K
80695771 00 6e 00 6f 00 77 00 6e 00 47 00 6f .n.o.w.n.G.o
8069577d 00 6f 00 64 00 52 00 65 00 63 00 6f .o.d.R.e.c.o
80695789 00 76 00 65 00 72 00 79 00 5c 00 4c .v.e.r.y.\.L
80695795 00 61 00 73 00 74 00 47 00 6f 00 6f .a.s.t.G.o.o
806957a1 00 64 00 00 00 5c 00 53 00 79 00 73 .d...\.S.y.s
806957ad 00 74 00 65 00 6d 00 52 00 6f 00 6f .t.e.m.R.o.o
806957b9 00 74 00 5c 00 4c 00 61 00 73 00 74 .t.\.L.a.s.t
806957c5 00 47 00 6f 00 6f 00 64 00 2e 00 54 .G.o.o.d...T
806957d1 00 6d 00 70 00 00 00 00 00 00 00 00 .m.p........
Memory
Virtual:
Previous
Display format: Pointer and
Next
f9ede18c 00000000
f9ede190 00000000
f9ede194 00000000
f9ede198 00000000
f9ede19c 00000000
f9ede1a0 00000000
f9ede1a4 00000000
f9ede1a8 f9edc000
Ln 0, Col 0
Sys 0:KdSrv:S
Proc 000:0
Thrd 000:0
ASM
OVR
CAPS
NUM

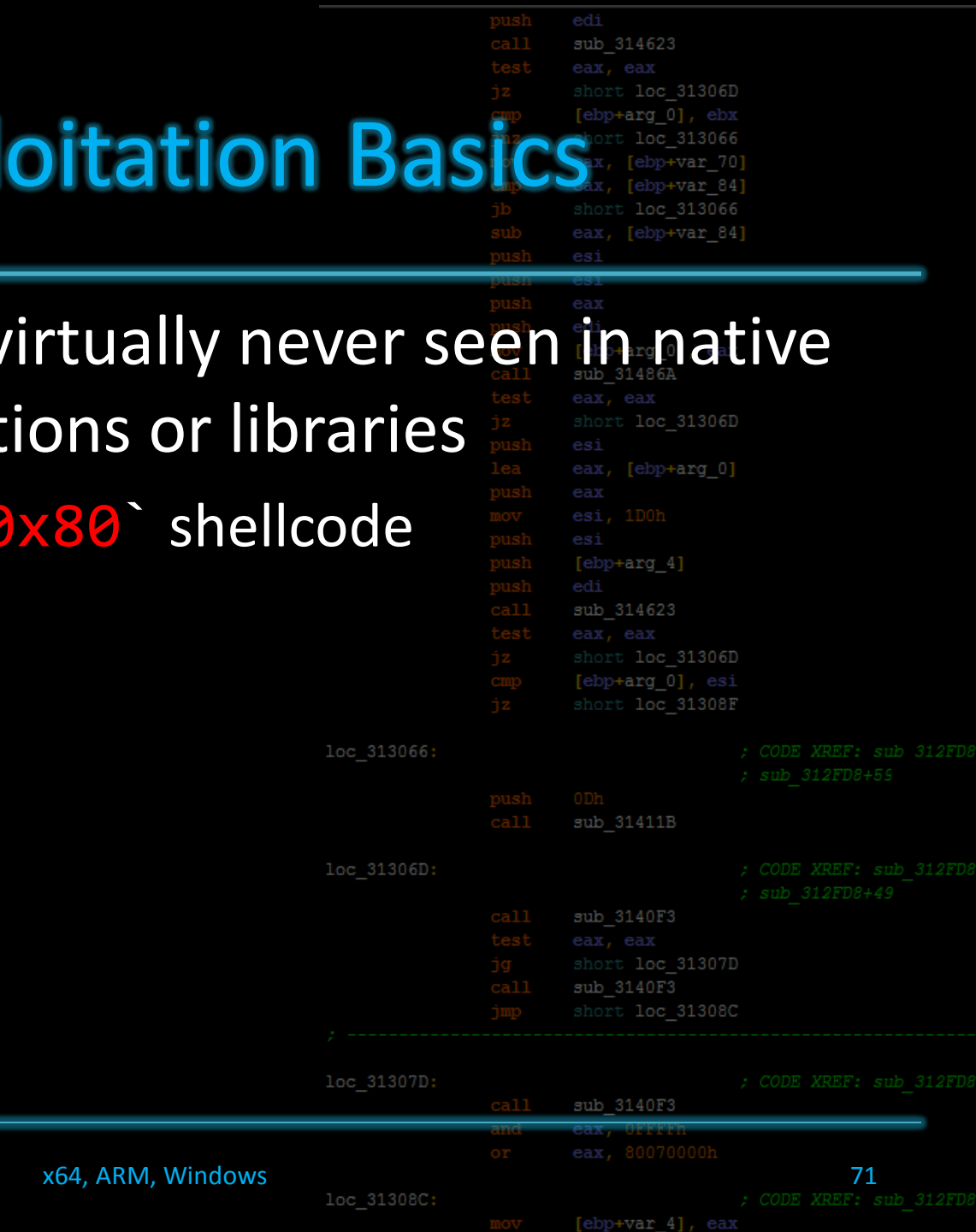
Windows Exploitation Basics

- Raw syscalls are virtually never seen in native windows applications or libraries



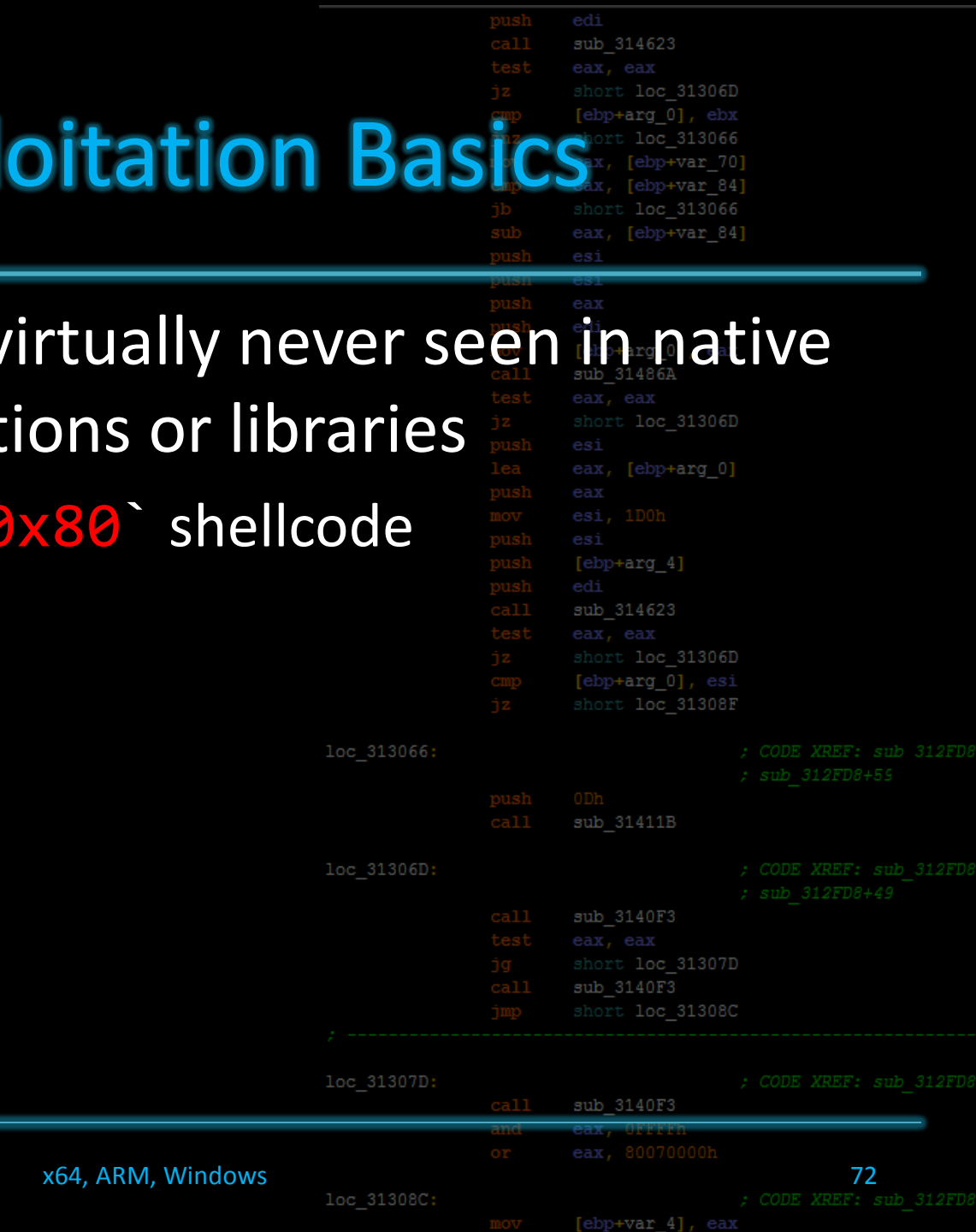
Windows Exploitation Basics

- Raw syscalls are virtually never seen in native windows applications or libraries
  - No more `int 0x80` shellcode



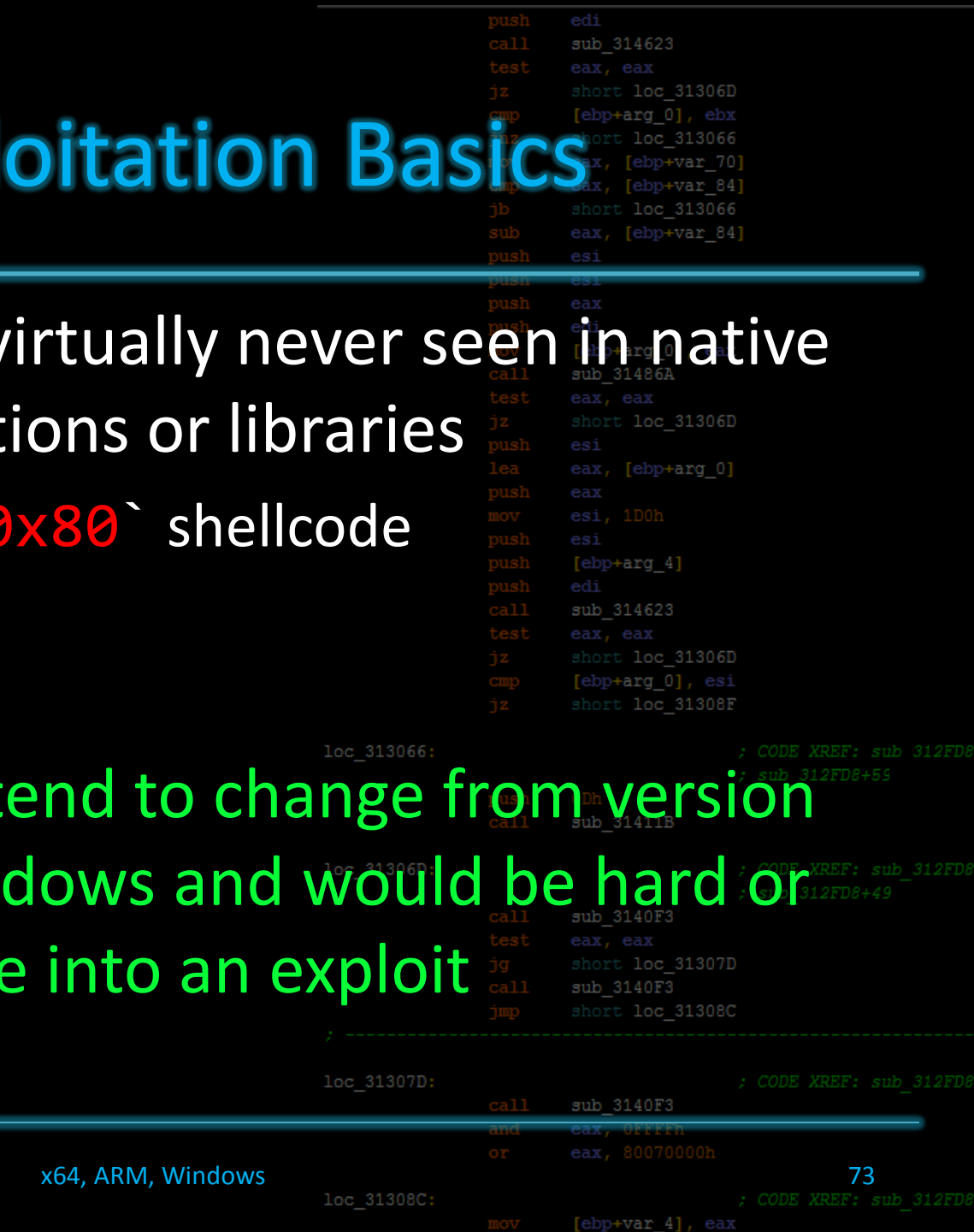
Windows Exploitation Basics

- Raw syscalls are virtually never seen in native windows applications or libraries
  - No more `int 0x80` shellcode
  - Why?



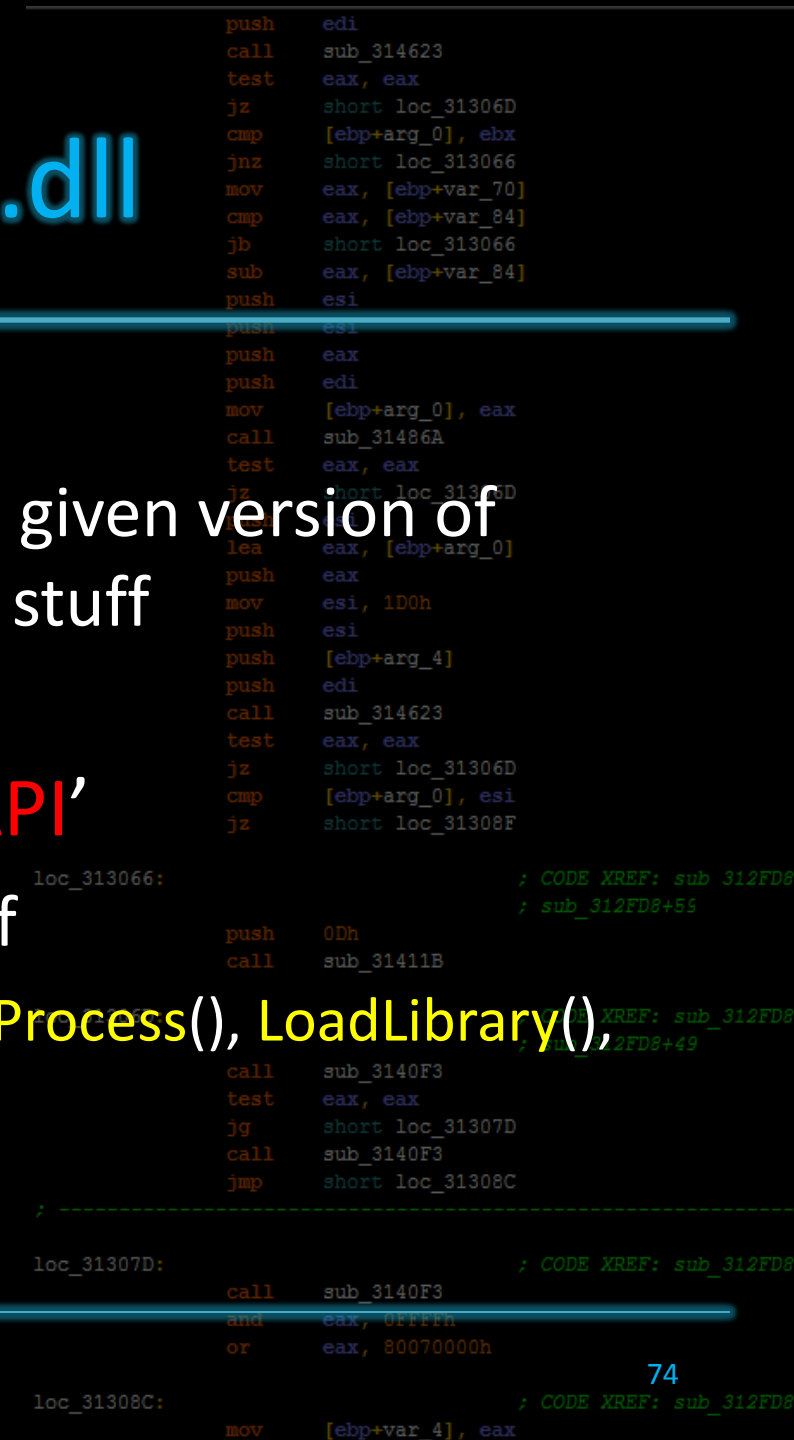
Windows Exploitation Basics

- Raw syscalls are virtually never seen in native windows applications or libraries
  - No more `` `int 0x80` `` shellcode
  - Why?
- Syscall numbers tend to change from version to version of Windows and would be hard or unreliable to code into an exploit



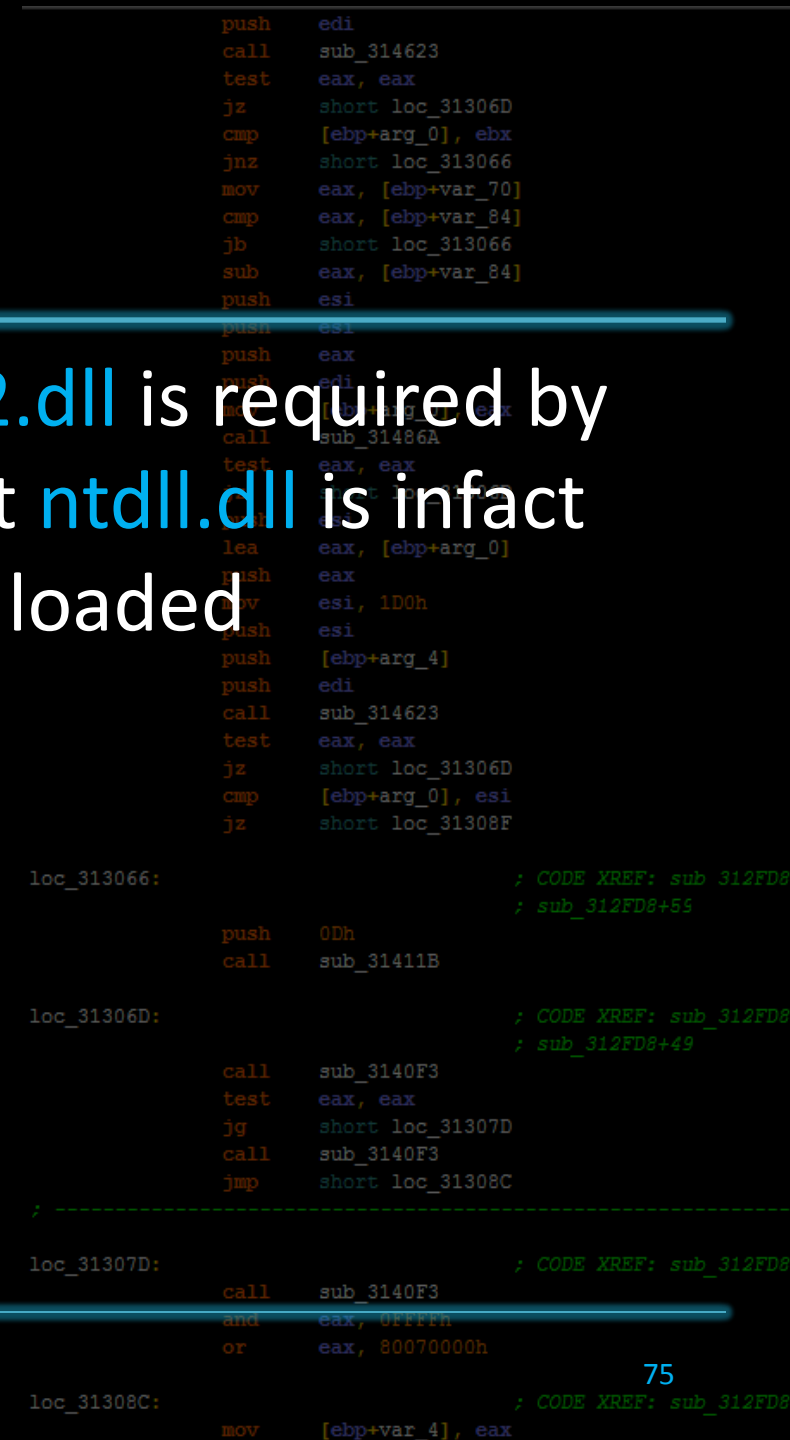
ntdll.dll and kernel32.dll

- ntdll.dll – the 'Native API'
  - Wraps all the syscalls for the given version of Windows, is pretty low level stuff
- kernel32.dll – the 'Win32 API'
  - More familiar high level stuff
    - OpenFile(), ReadFile(), CreateProcess(), LoadLibrary(), GetProcAddress(),



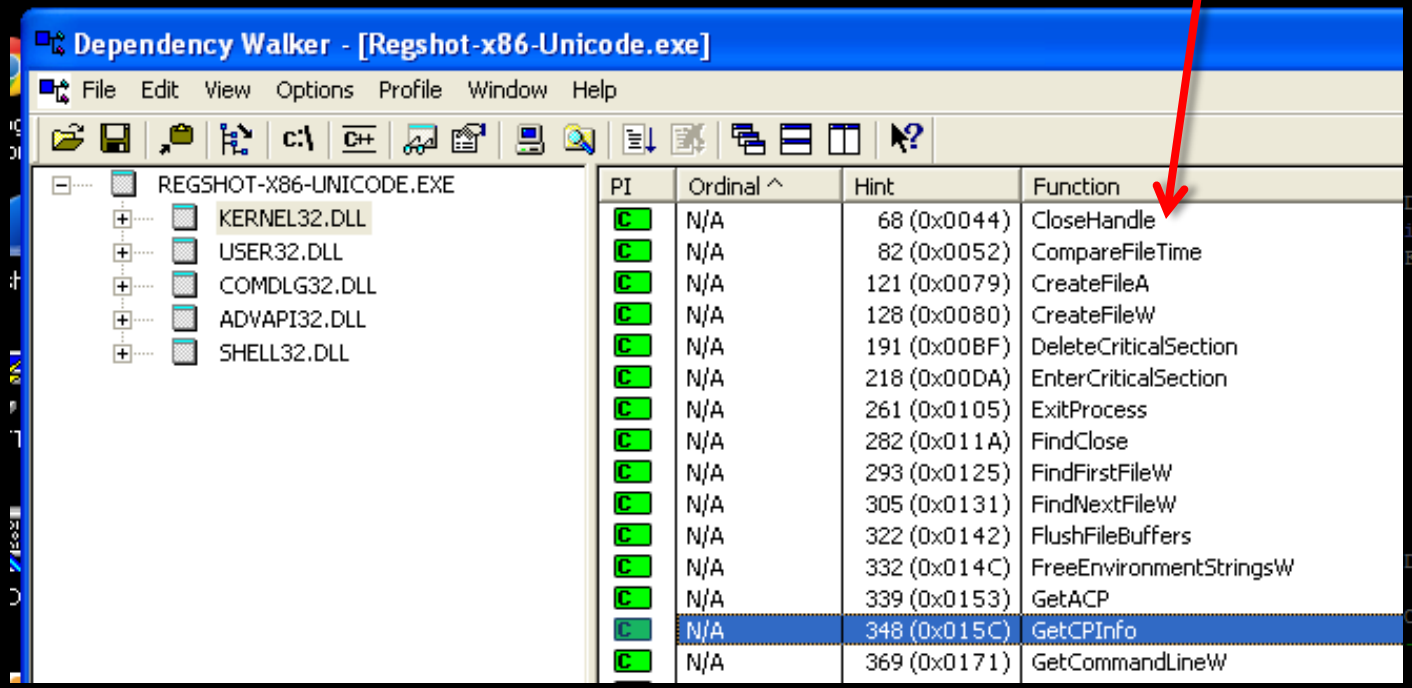
Windows Fun Facts

- Most people think kernel32.dll is required by every windows process, but ntdll.dll is infact the only one that MUST be loaded



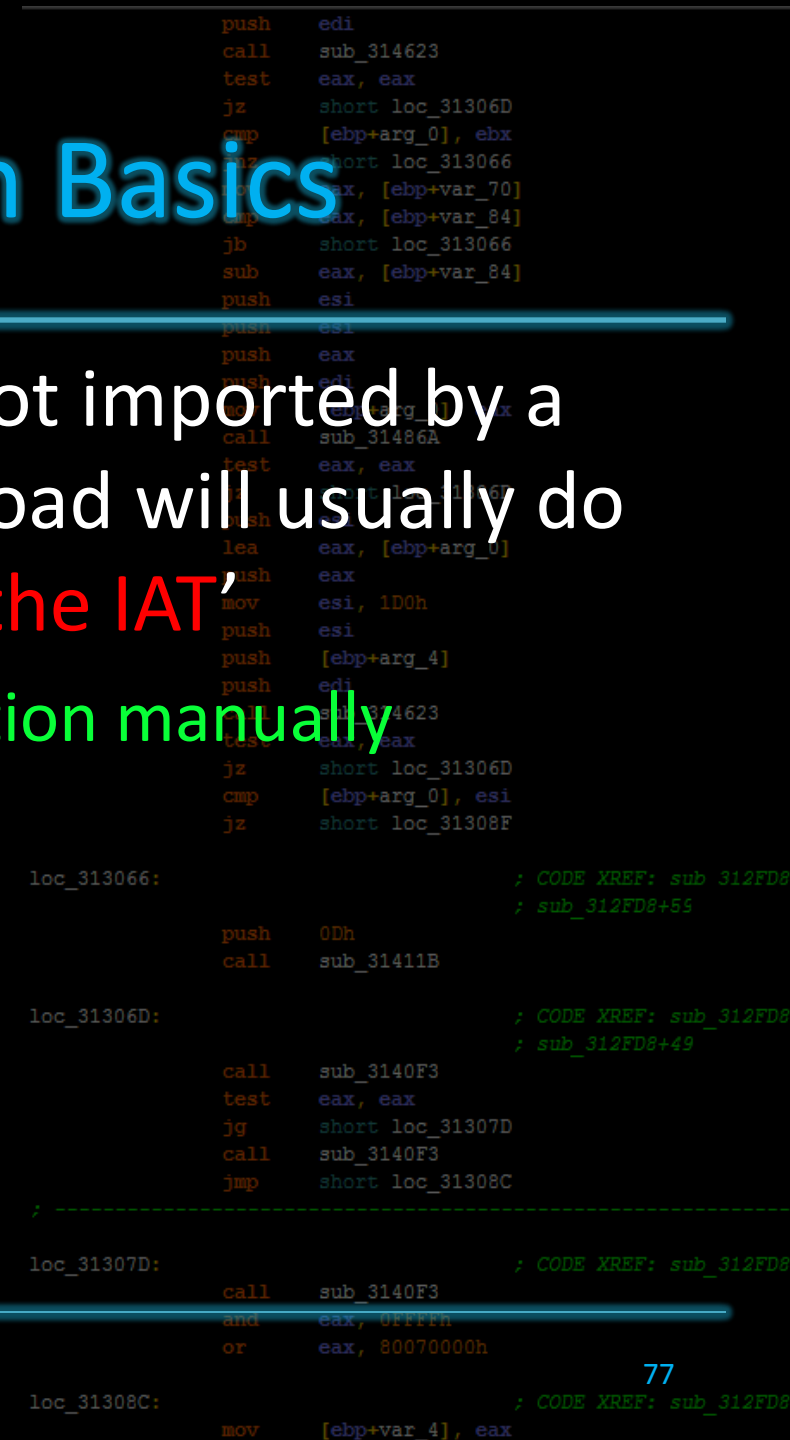
Windows Exploitation Basics

- So instead of using syscalls, an exploit will almost always use existing imported functions



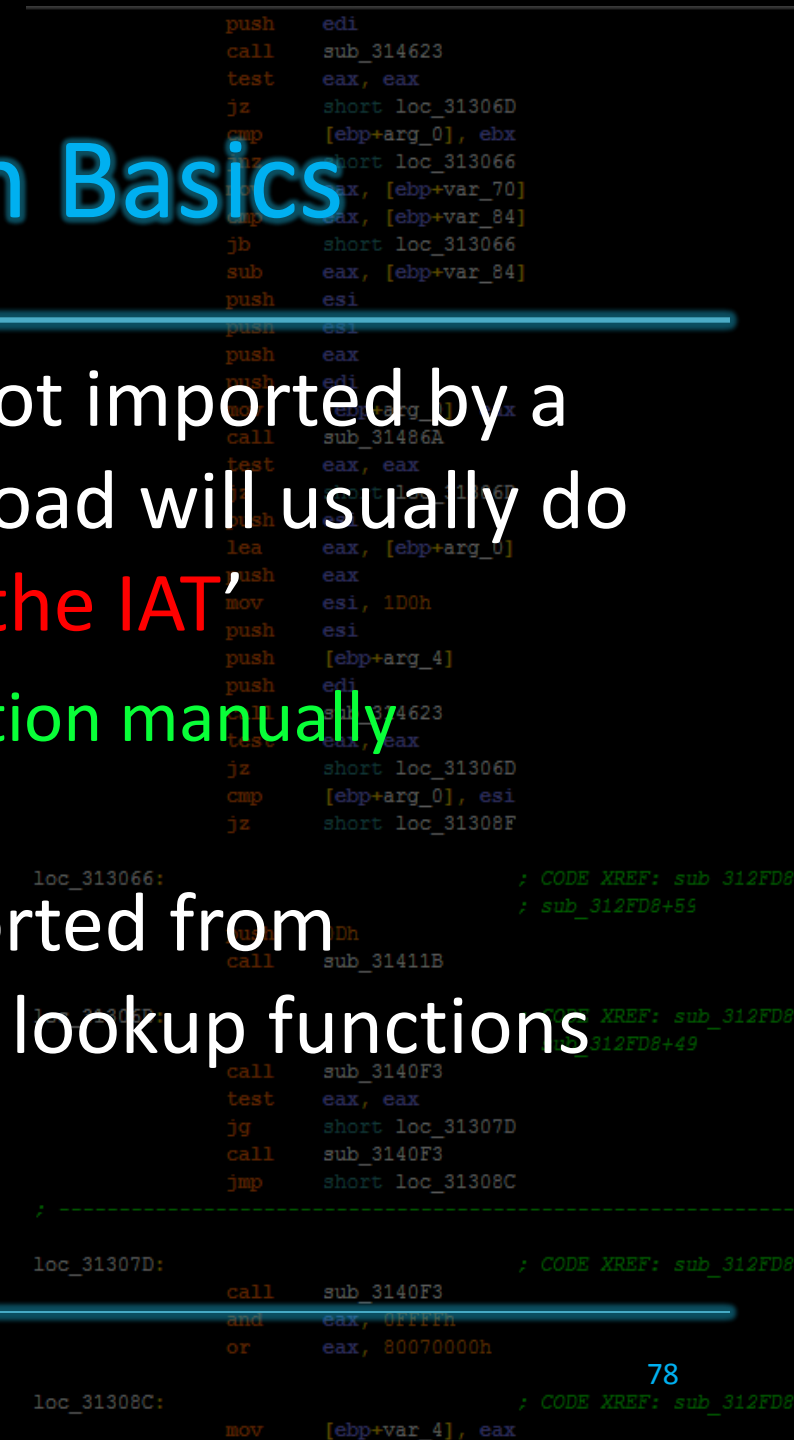
Windows Exploitation Basics

- If a function of interest is not imported by a loaded DLL, an exploit payload will usually do what is known as 'walking the IAT'
  - It resolves the function location manually



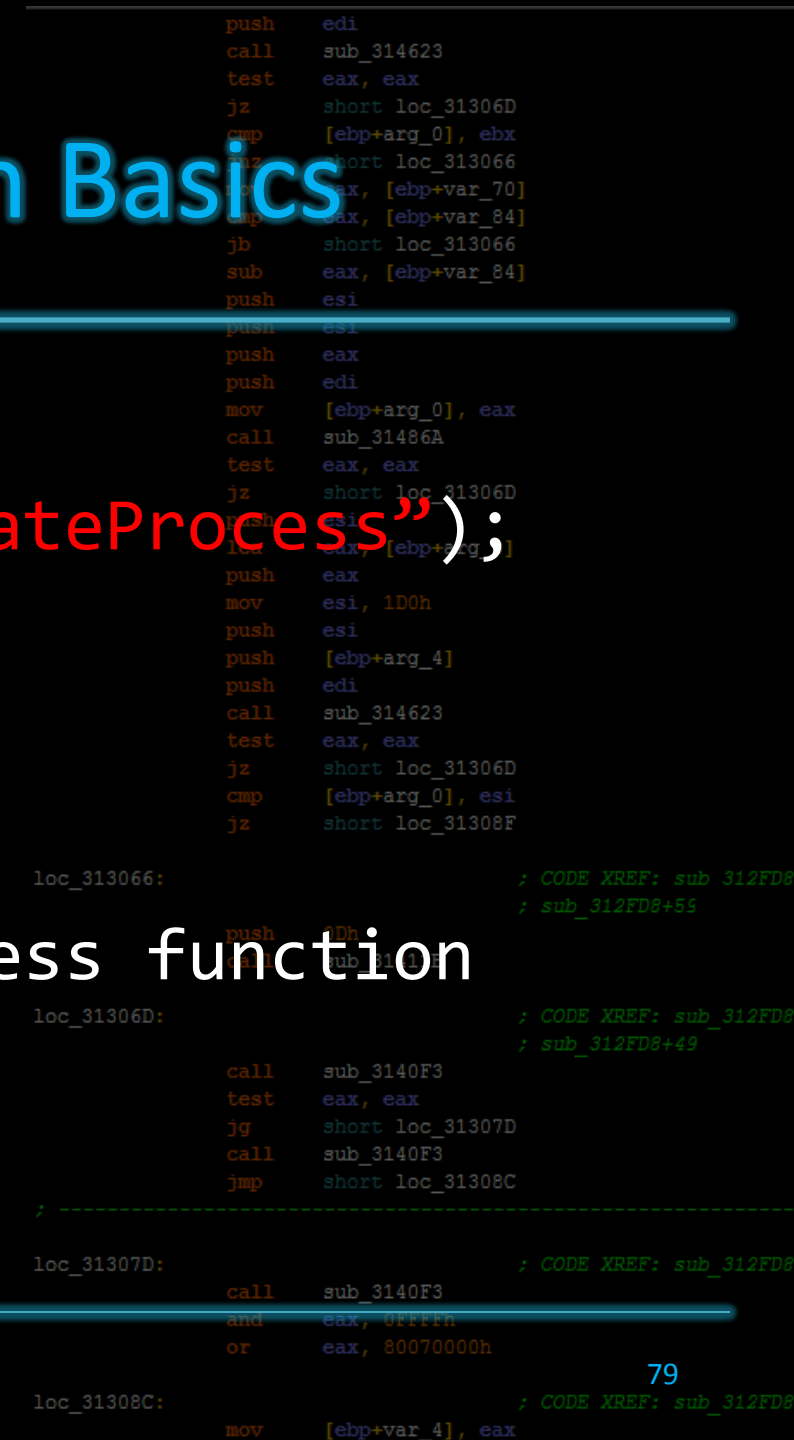
Windows Exploitation Basics

- If a function of interest is not imported by a loaded DLL, an exploit payload will usually do what is known as 'walking the IAT'
  - It resolves the function location manually
- If GetProcAddress() is imported from kernel32.dll, you can easily lookup functions
  - Same as dlsym() on Linux



Windows Exploitation Basics

```
GetProcAddress(k32h, “CreateProcess”);

...

Looking up the CreateProcess function
```



Windows XP Security

- Windows XP SP2 marked the start of the modern security era (Summer 2004)

Protection – Bypass – ???

Windows XP Security

- Windows XP SP2 marked the start of the modern security era (Summer 2004)
  - Hardware Enforced DEP – ROP

Protection – Bypass – ???

Windows XP Security

- Windows XP SP2 marked the start of the modern security era (Summer 2004)
  - Hardware Enforced DEP – ROP
  - Stack Cookies (GS) – Leak & replace, write past, SEH

Protection – Bypass – ???

Windows XP Security

- Windows XP SP2 marked the start of the modern security era (Summer 2004)
  - Hardware Enforced DEP – ROP
  - Stack Cookies (GS) – Leak & replace, write past, SEH
  - Safe heap unlinking – Heap metadata exploits

Protection – Bypass – ???

Windows XP Security

- Windows XP SP2 marked the start of the modern security era (Summer 2004)
  - Hardware Enforced DEP – ROP
  - Stack Cookies (GS) – Leak & replace, write past, SEH
  - Safe heap unlinking – Heap metadata exploits
  - SafeSEH – ?

Protection – Bypass – ???

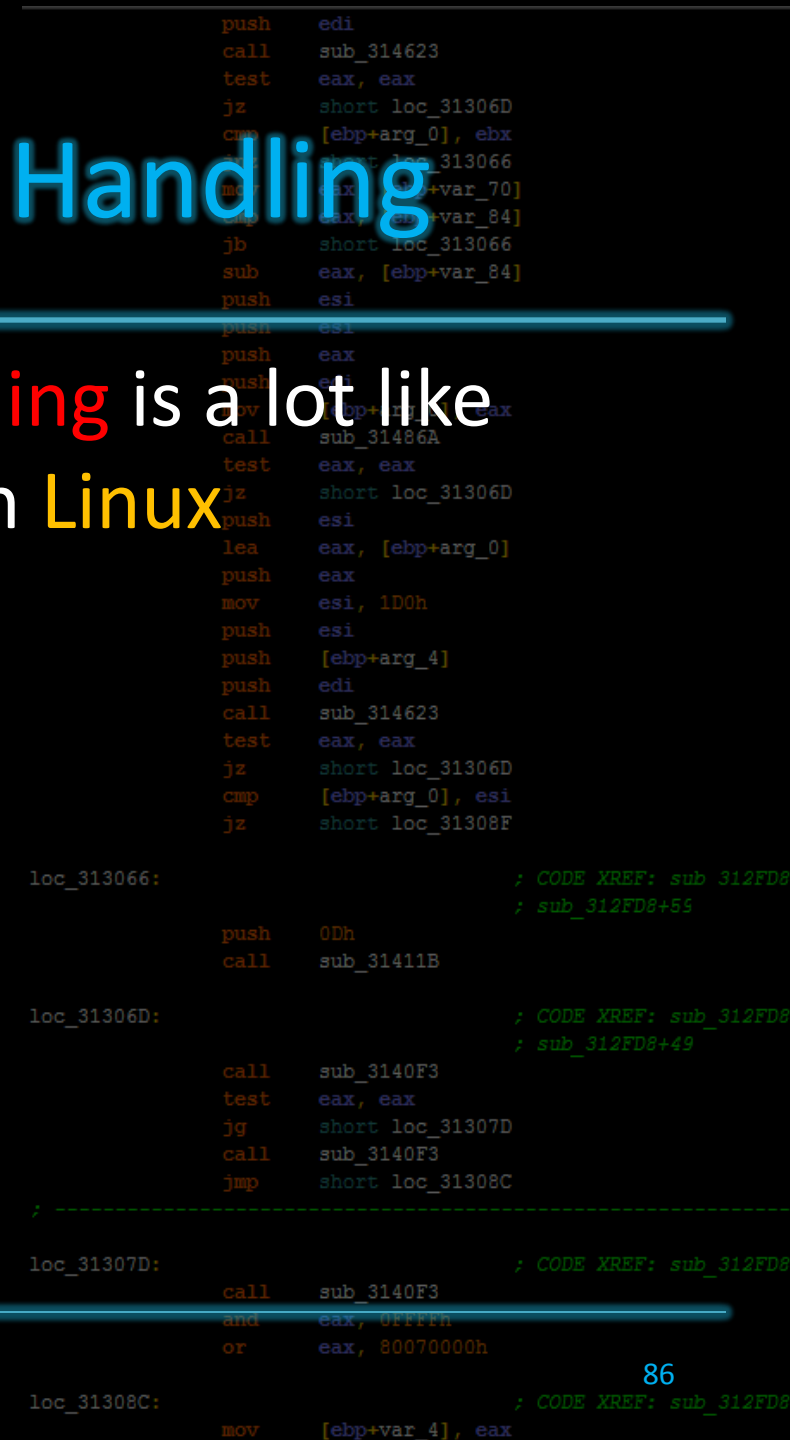
Windows XP Security

- Windows XP SP2 marked the start of the modern security era (Summer 2004)
  - Hardware Enforced DEP – ROP
  - Stack Cookies (GS) – Leak & replace, write past, SEH
  - Safe heap unlinking – Heap metadata exploits
  - SafeSEH – ? What is SEH/SafeSEH ?

Protection – Bypass – ???

Structured Exception Handling

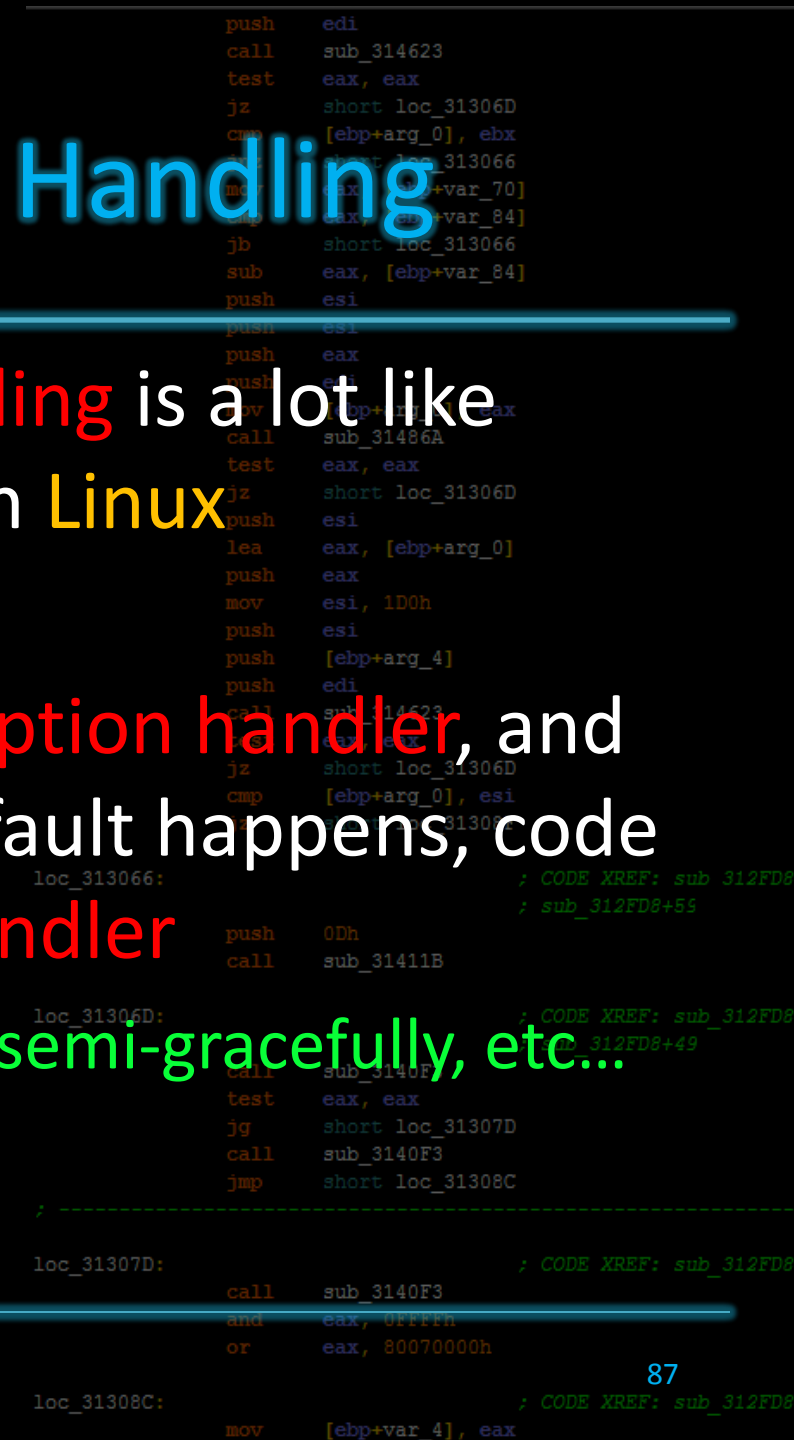
- Structured Exception Handling is a lot like assigning signal handlers on Linux



Structured Exception Handling

- Structured Exception Handling is a lot like assigning signal handlers on Linux

- You simply register an exception handler, and if something bad like a segfault happens, code flow is redirected to the handler
  - Print an error message, exit semi-gracefully, etc…

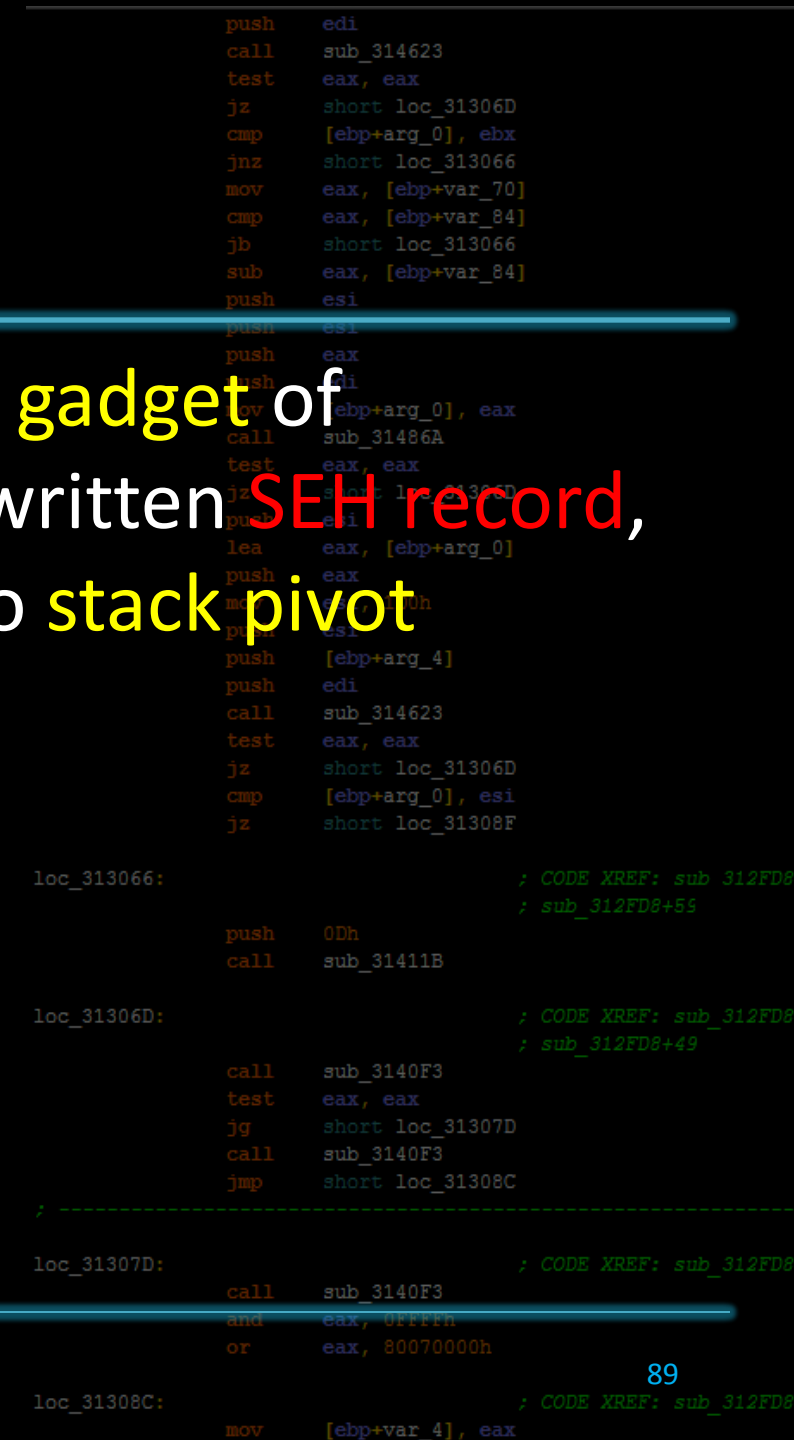


Exploiting SEH

- Exception records are placed on the stack, so they're relatively easy to corrupt

Exploiting SEH

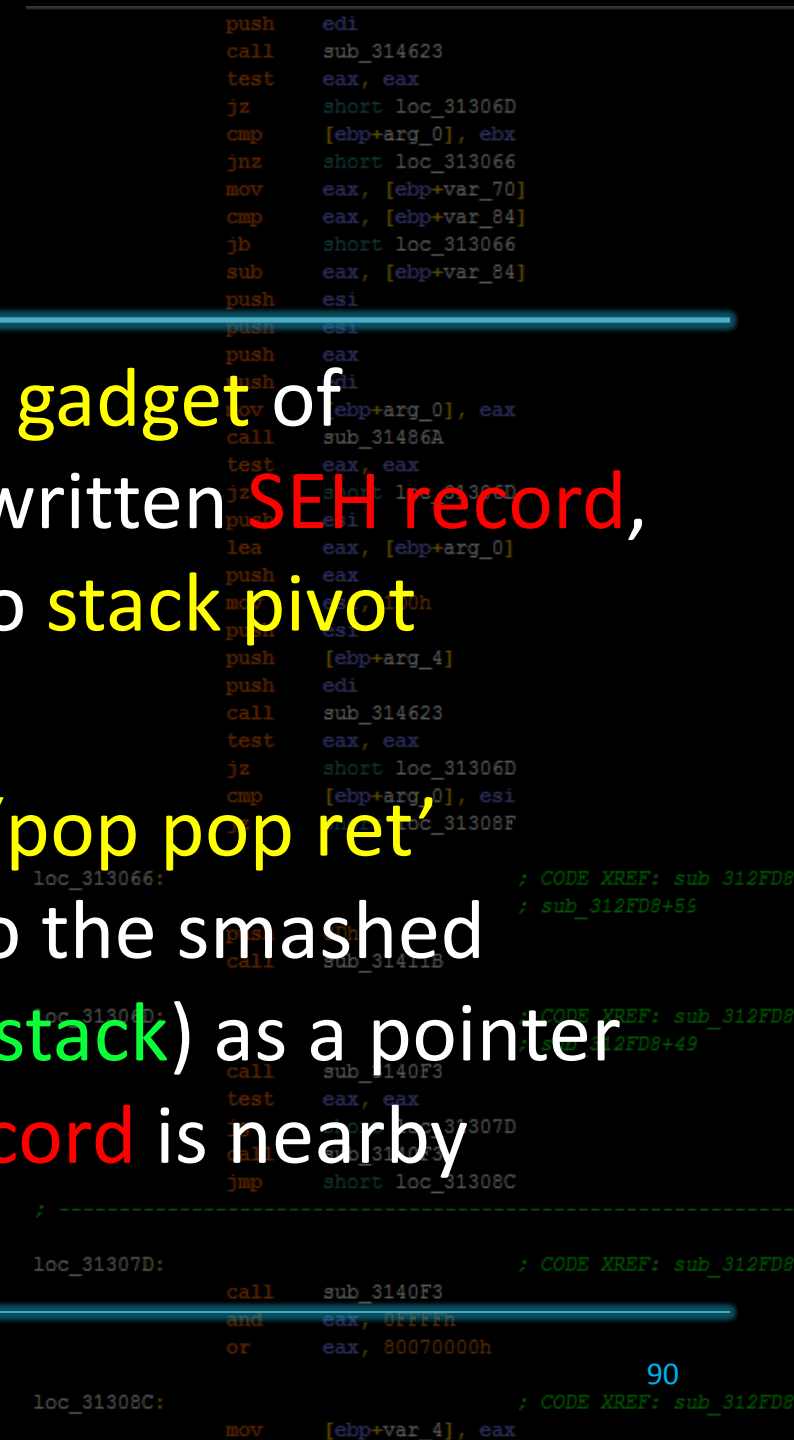
- Because you only have one gadget of execution through an overwritten SEH record, you usually have to use it to stack pivot



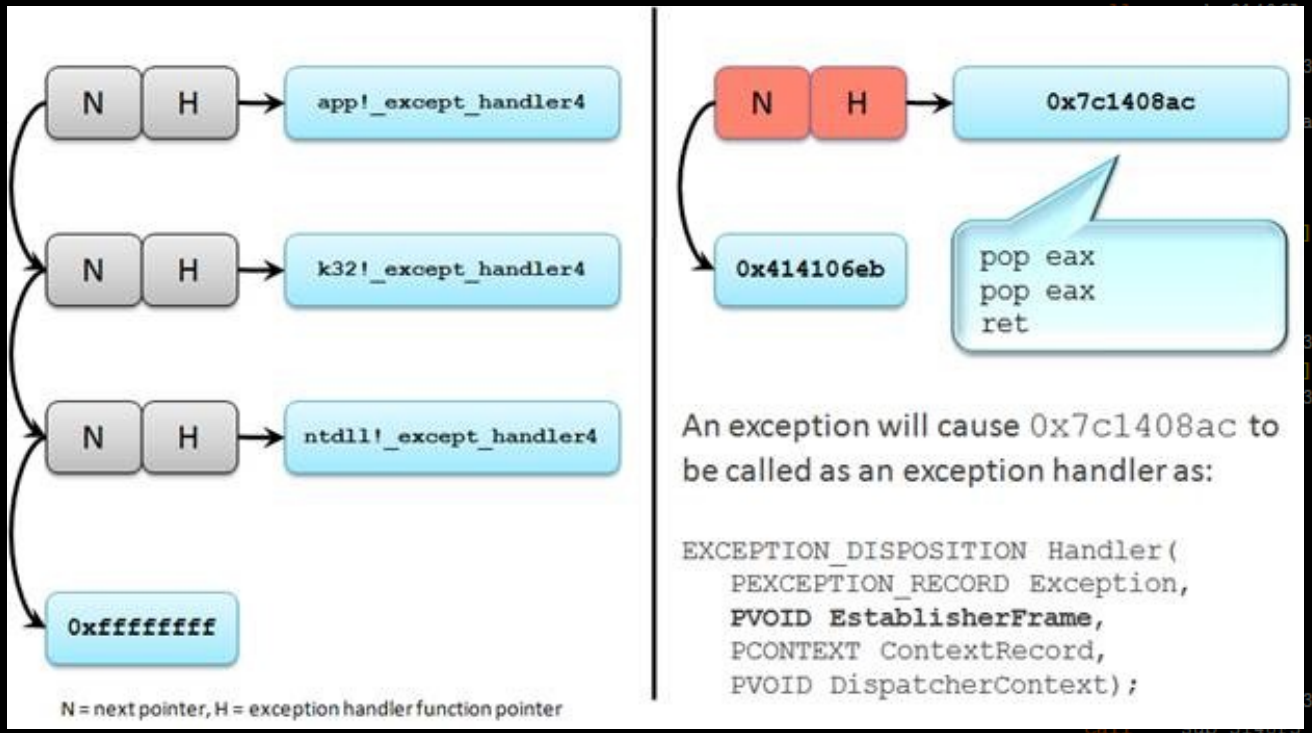
Exploiting SEH

- Because you only have one gadget of execution through an overwritten SEH record, you usually have to use it to stack pivot

- Classically you could use a 'pop pop ret' gadget to easily return onto the smashed stack (assumes executable stack) as a pointer to your overwritten SEH record is nearby

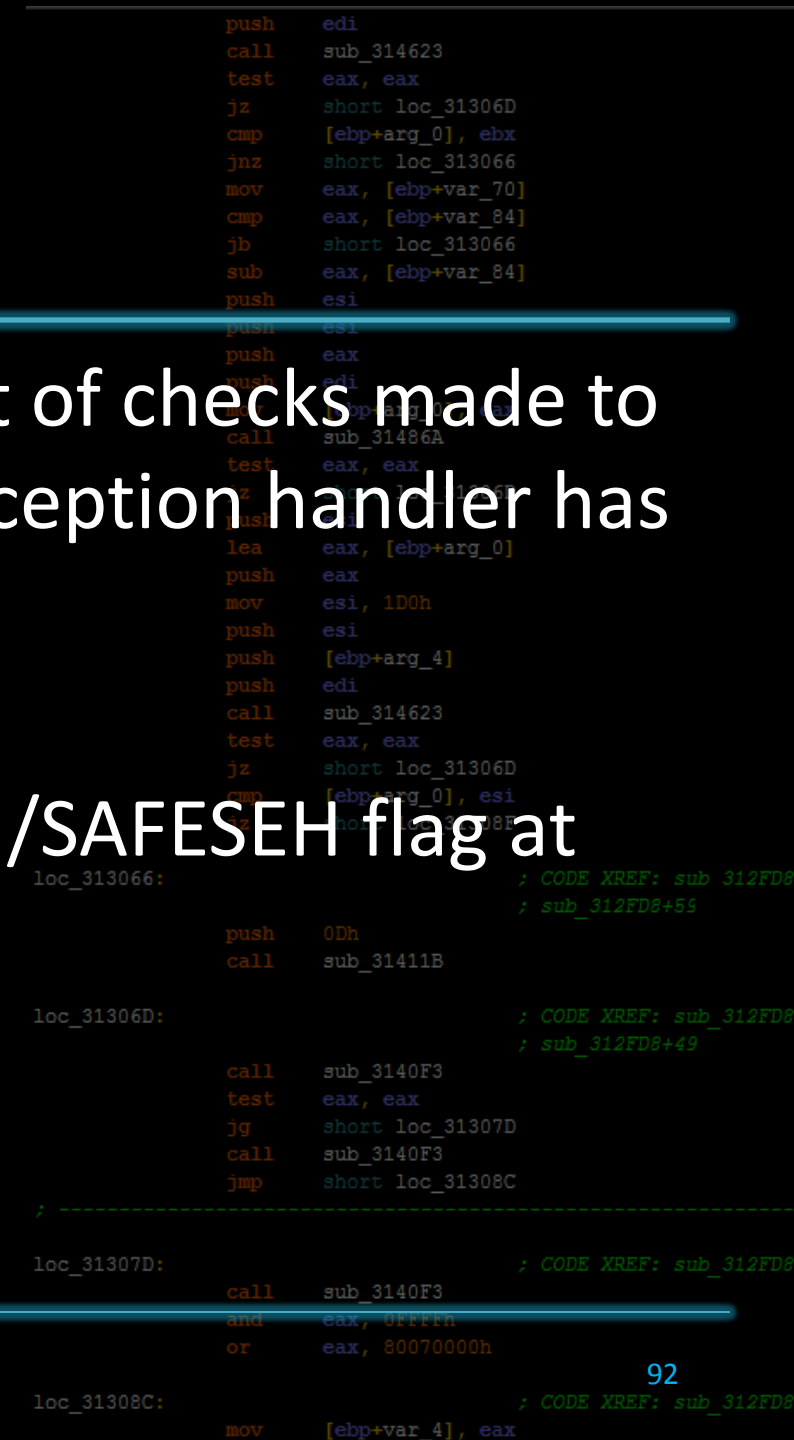


Exploiting SEH



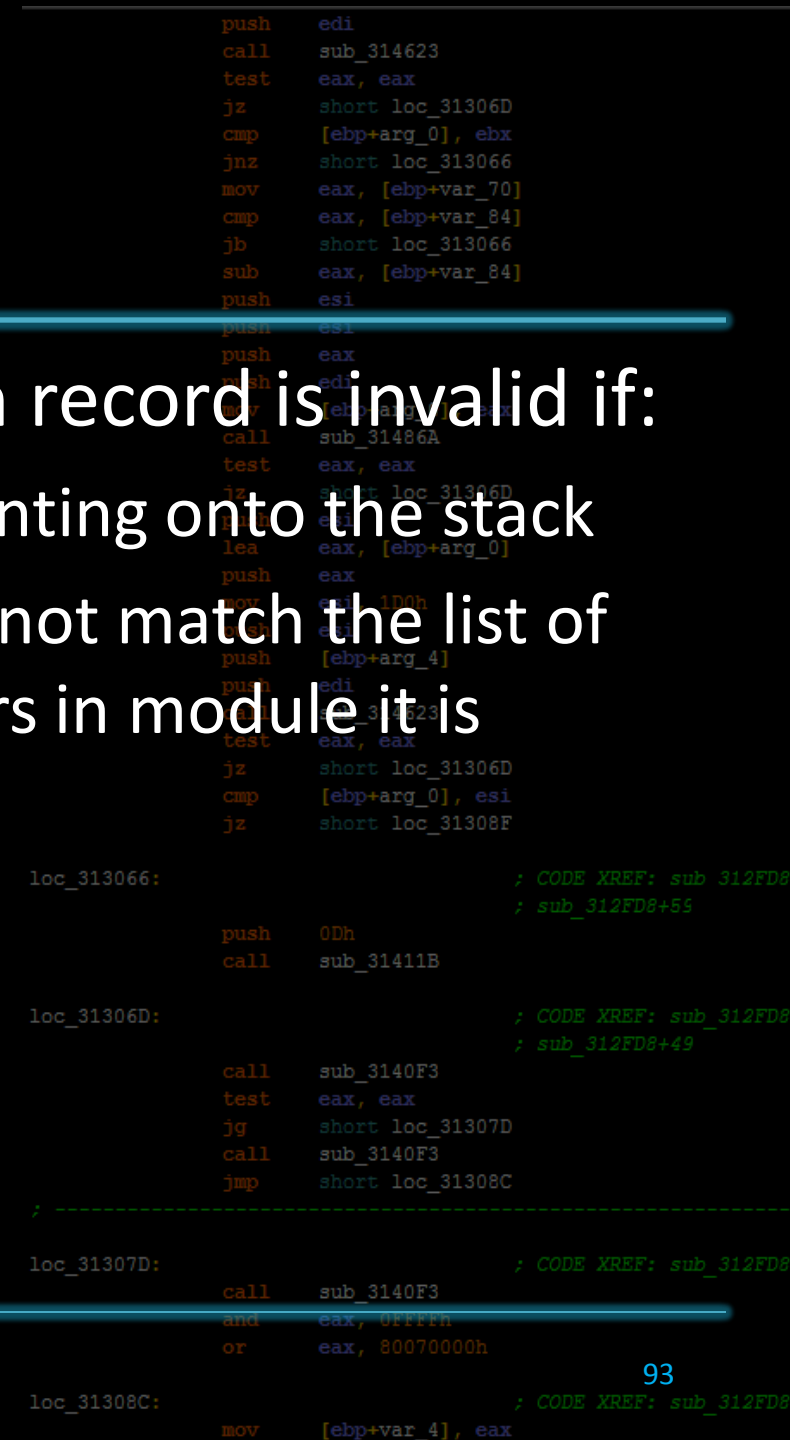
SafeSEH

- SafeSEH is an additional set of checks made to ensure that a registered exception handler has not been corrupted
- You can enable it using the /SAFESEH flag at compile time



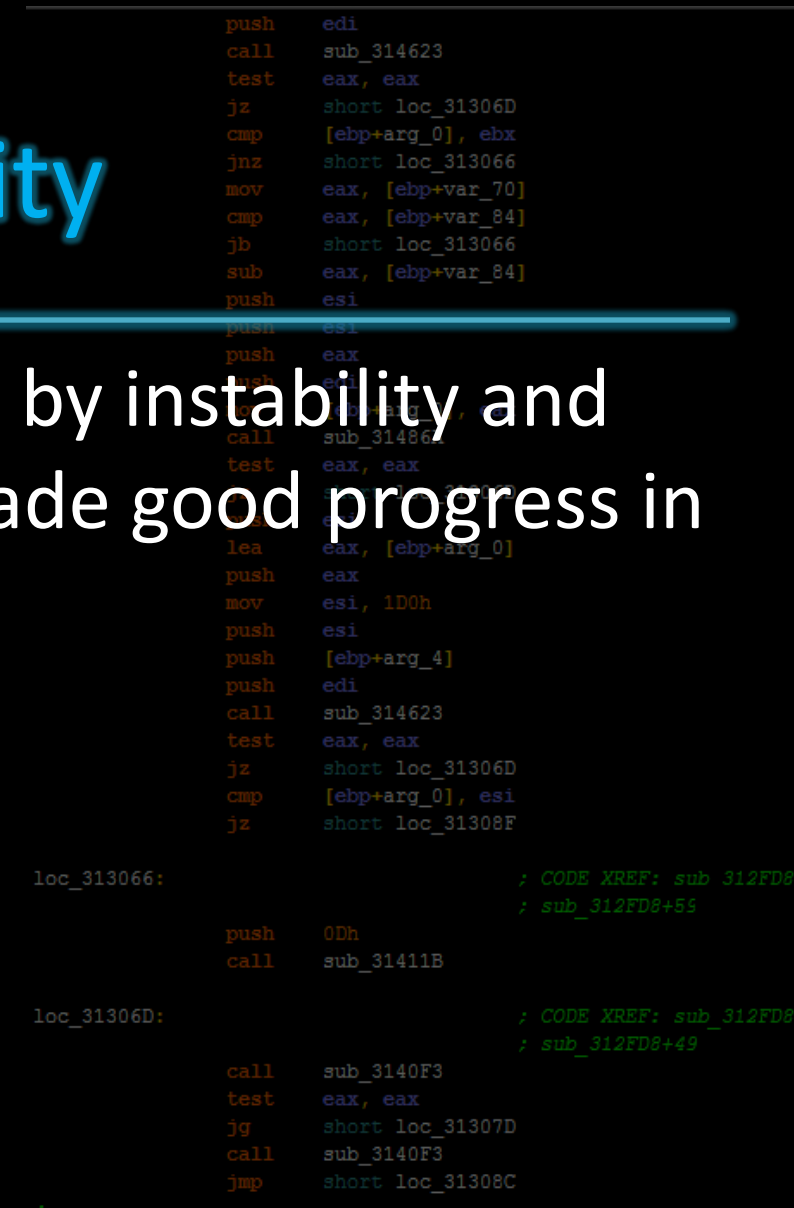
Bypassing SafeSEH

- With SafeSEH, an exception record is invalid if:
  - The exception handler is pointing onto the stack
  - The exception handler does not match the list of registered exception handlers in module it is pointing into



Windows Vista Security

- Windows Vista was marred by instability and performance issues, but made good progress in terms of security



Protection – Bypass – ???

Windows Vista Security

- Windows Vista was marred by instability and performance issues, but made good progress in terms of security
  - ASLR – Info leaks, partial overwrites, non aslr'd code

Protection – Bypass – ???

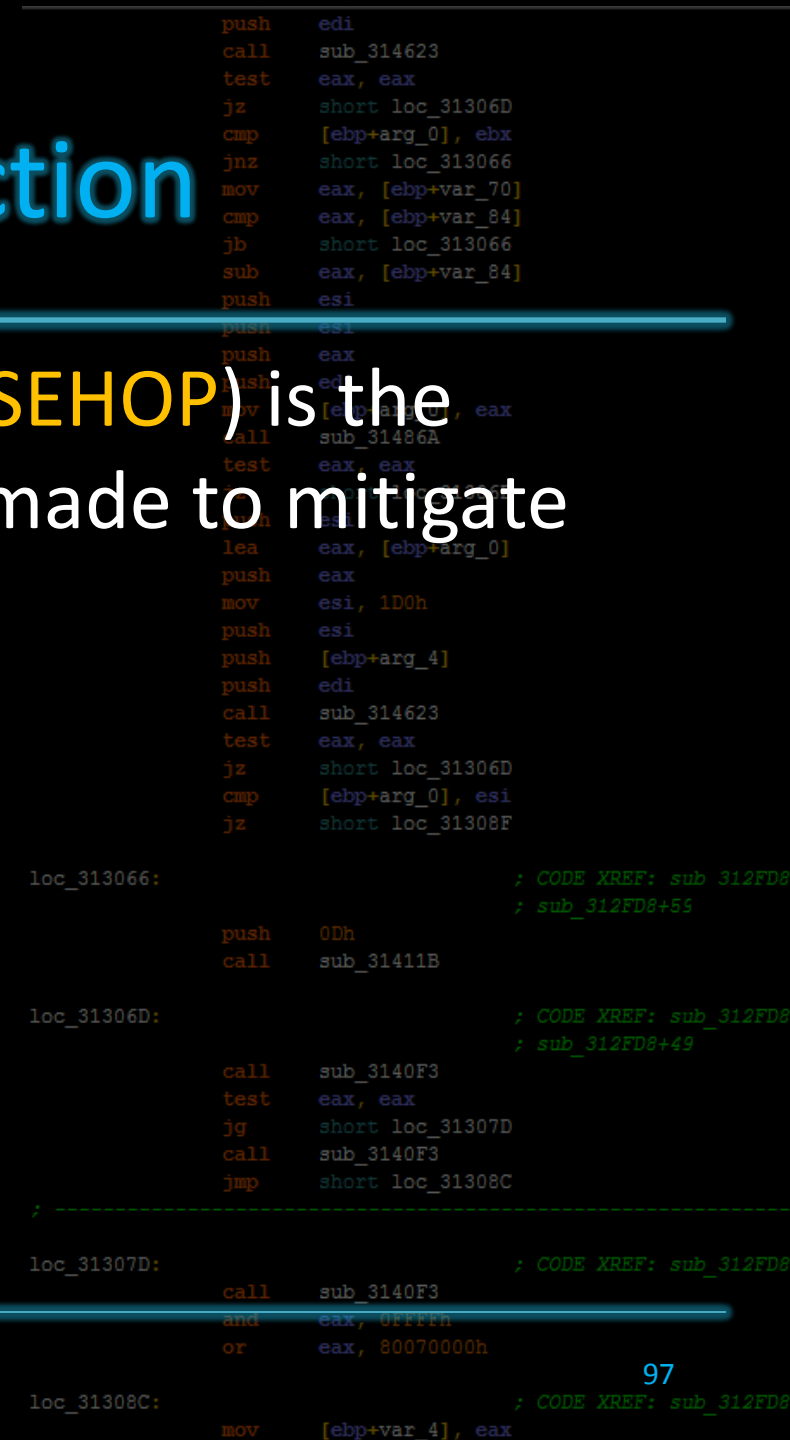
Windows Vista Security

- Windows Vista was marred by instability and performance issues, but made good progress in terms of security
  - ASLR – Info leaks, partial overwrites, non aslr'd code
  - SEHOP – ?

Protection – Bypass – ???

SEH Overwrite Protection

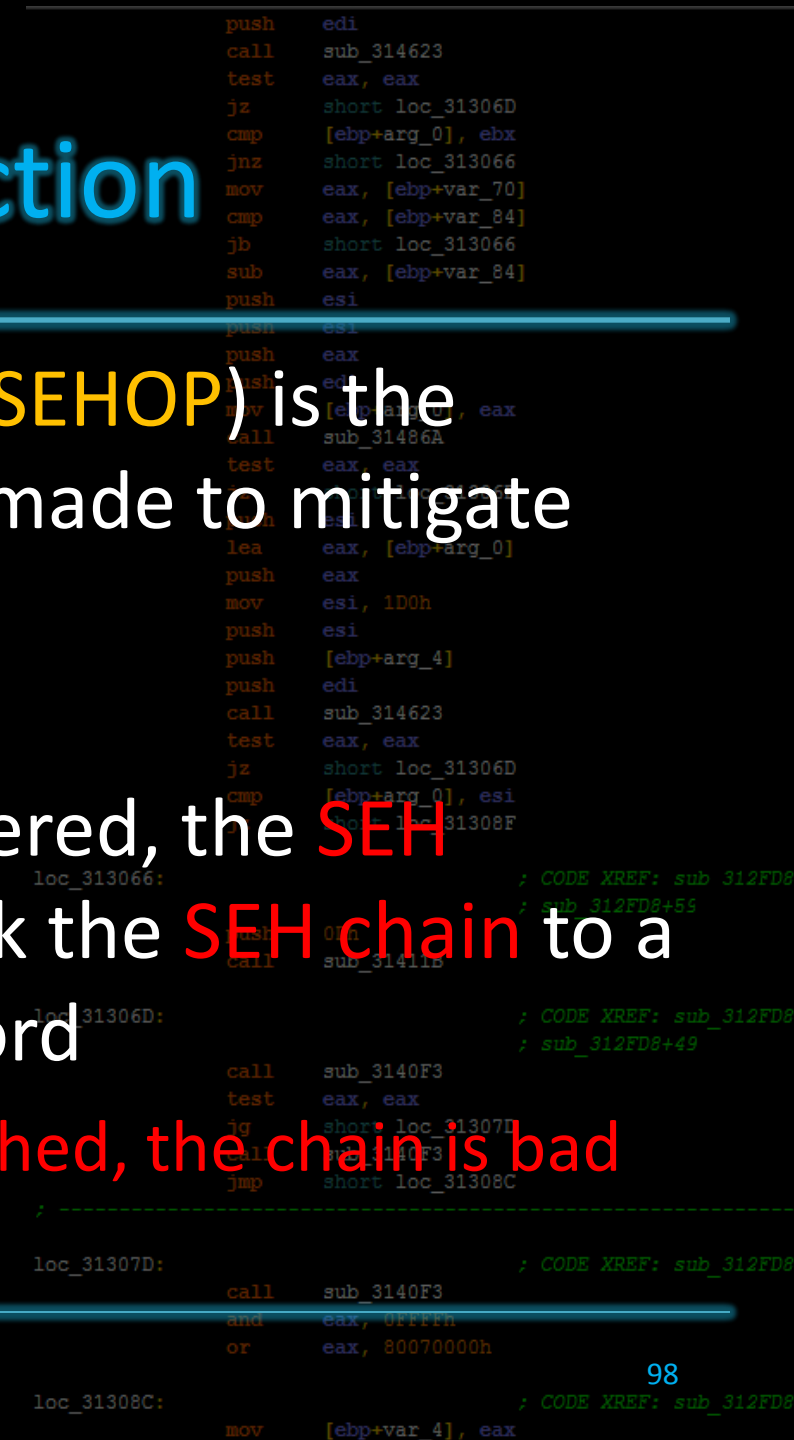
- SEH Overwrite Protection (SEHOP) is the second attempt Microsoft made to mitigate SEH exploitation



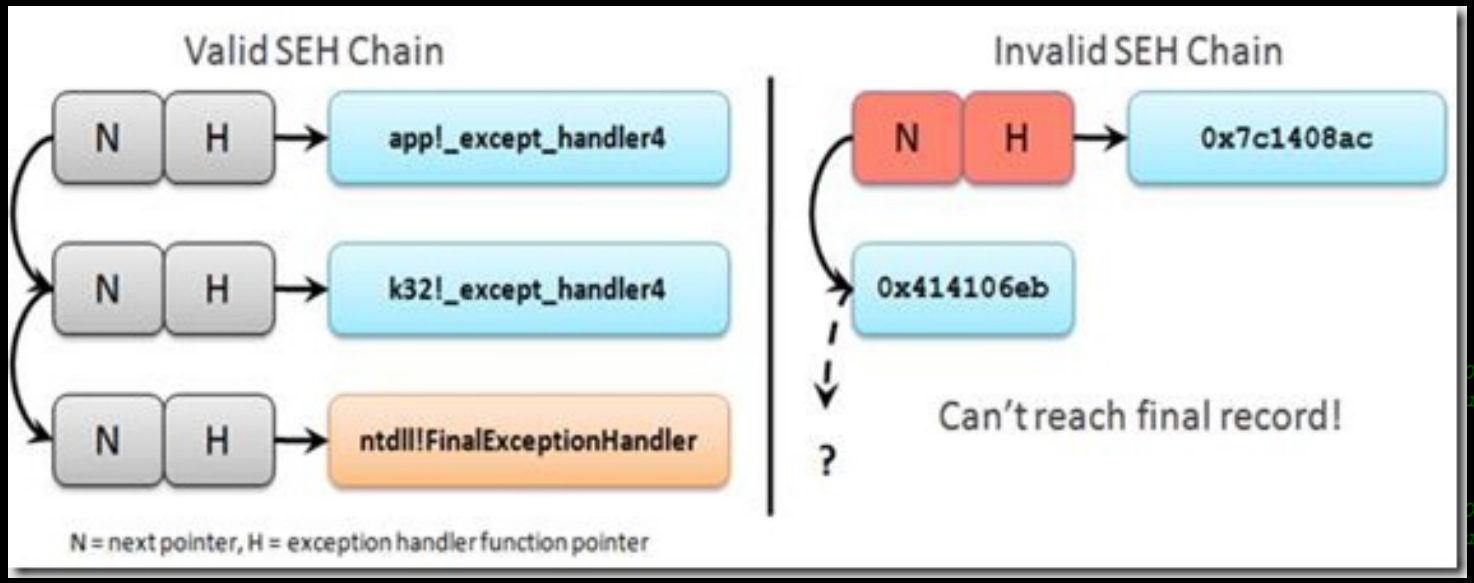
SEH Overwrite Protection

- SEH Overwrite Protection (SEHOP) is the second attempt Microsoft made to mitigate SEH exploitation

- When an exception is triggered, the SEH dispatcher attempts to walk the SEH chain to a symbolic 'terminating' record
  - If this record cannot be reached, the chain is bad

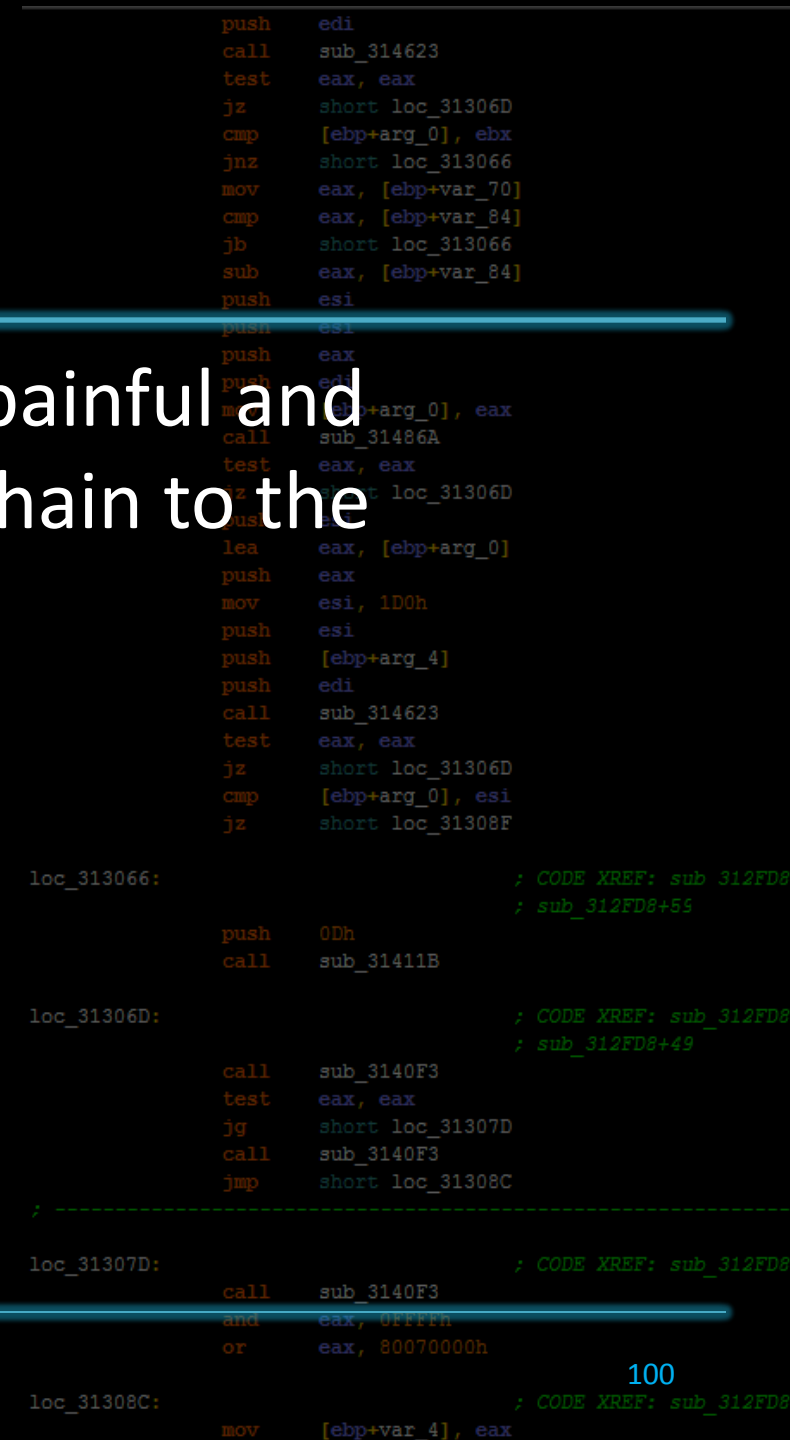


SEH Overwrite Protection



Bypassing SEHOP

- Bypassing SEHOP is pretty painful and basically involves faking a chain to the terminating record



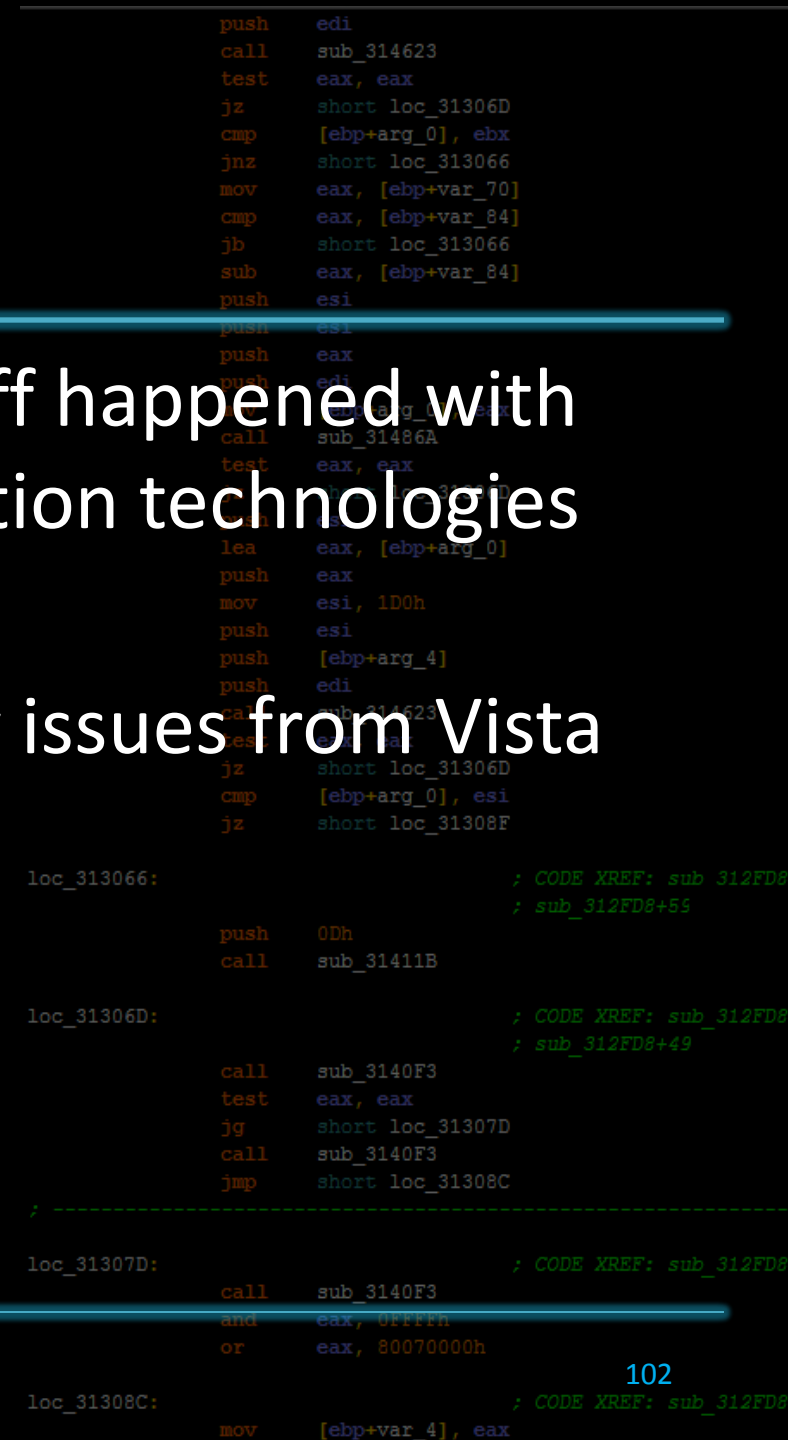
Windows Vista Security

- Windows Vista was marred by instability and performance issues, but made good progress in terms of security
  - ASLR – Info leaks, partial overwrites, non aslr'd code
  - SEHOP – Faking SEH Chains
  - Heap Hardening – More heap metadata checks

Protection – Bypass – ???

Windows 7 Security

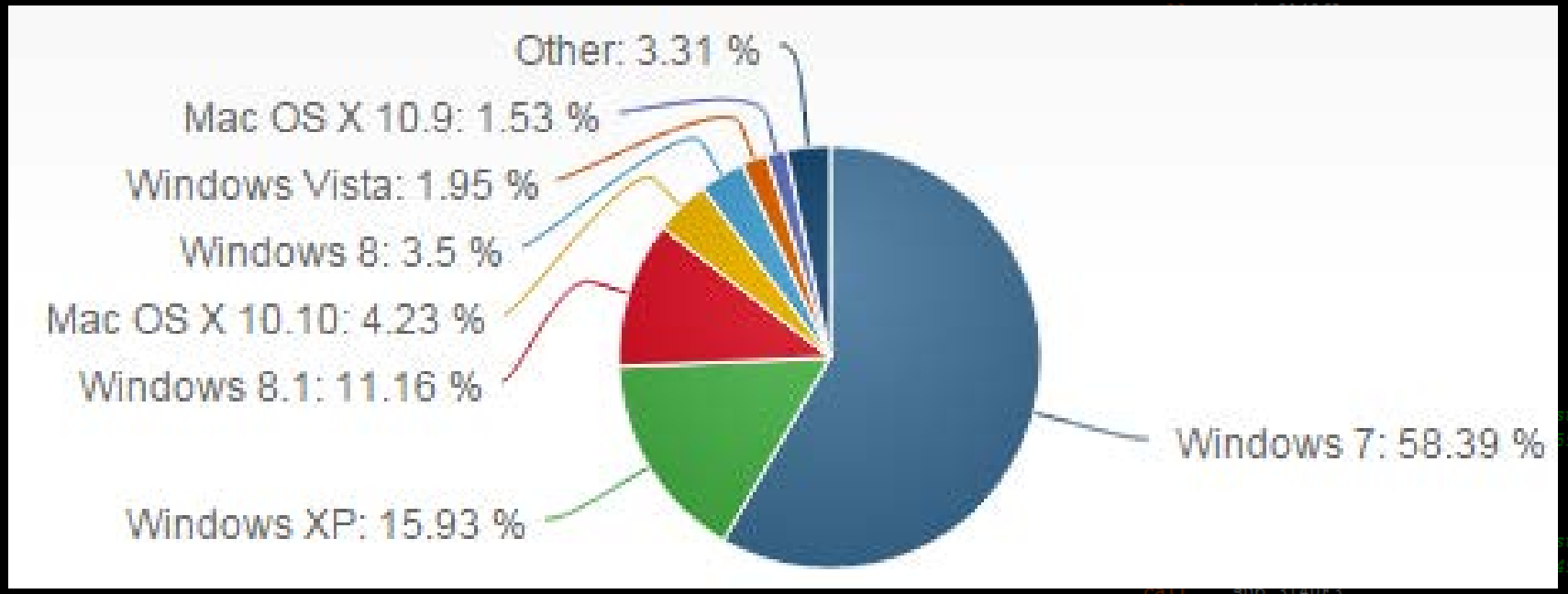
- I don’t think much new stuff happened with Windows 7 in terms mitigation technologies
- Mostly cleaning up stability issues from Vista



Windows 8 Security

- Windows 8/8.1 took a big step forward in sec
  - Enhanced GS (Stack Cookies)
  - VTGuard – Like a Vtable Canary
  - Heap Hardening
    - Allocation order randomization – Non-deterministic alloc. order
    - Guard pages – A bit like canaries between heap pages
  - ASLR Entropy Improvements – More entropy all around
  - PatchGuard – Prevent the kernel from being live patched
  - Secure Boot – Eliminate root/boot kits with chain of trust
  - Control Flow Guard – Whitelist indirect calls

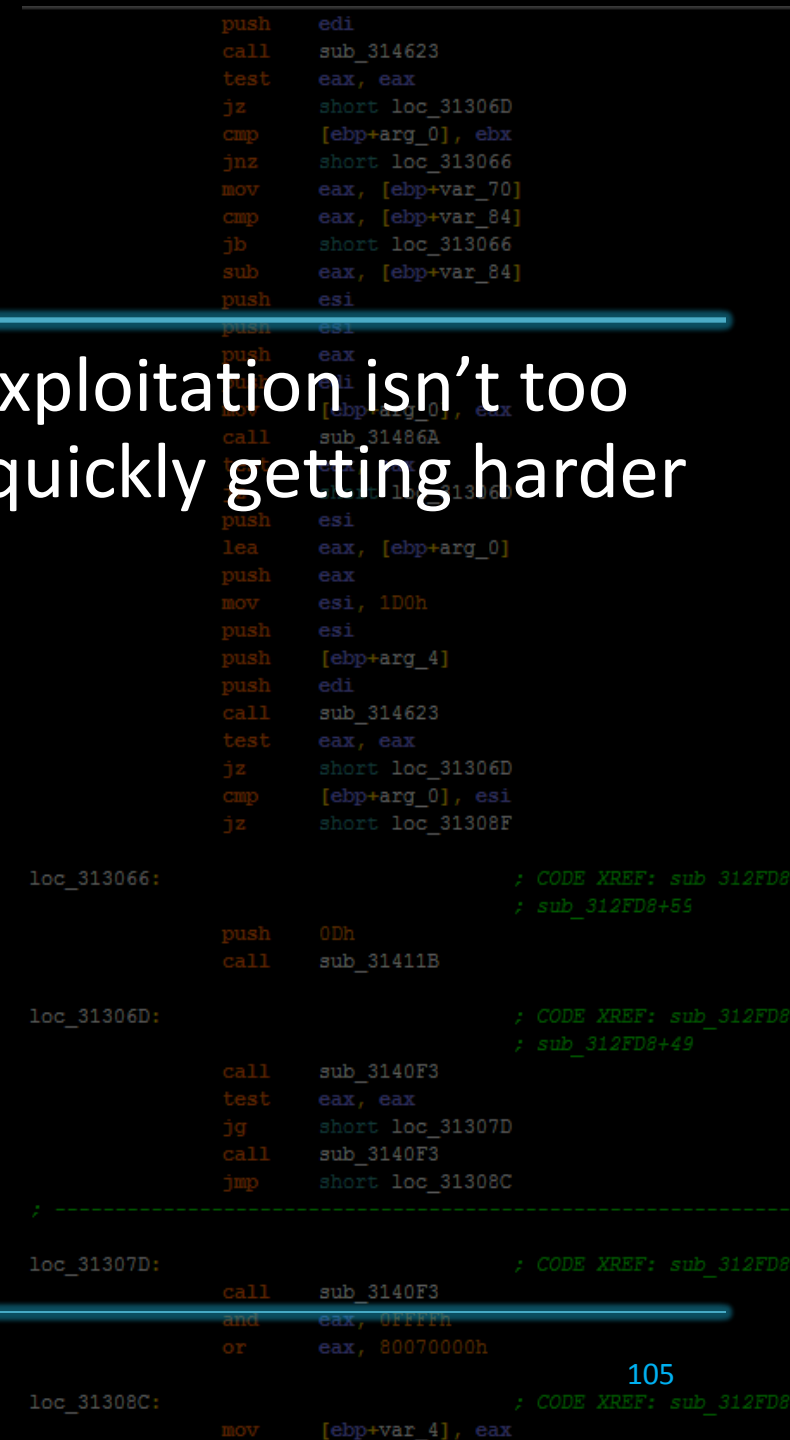
Desktop Market Share, May 2015



Windows market share, ~90.93%

Windows Summary

- In the end, Windows based exploitation isn't too different from Linux, but it's quickly getting harder



Windows Summary

- In the end, Windows based exploitation isn't too different from Linux, but it's quickly getting harder

- Some main takeaways
  - Differing 64bit calling convention
  - Syscalls aren't really a thing on Windows
  - New class of vulnerabilities, SEH Exploitation
    - New protections, SafeSEH, SEHOP
  - Better ASLR & Heap internals
  - Its mitigation technologies are rapidly evolving