x64, ARM, Windows

Modern Binary Exploitation
CSCI 4968 - Spring 2015
Markus Gaaschedelen

MBE - 05/08/2015
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

```
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]
jo short loc_313066
sub eax, [ebp+var_84]
push esi
push eax
push [ebp+arg_0]
call sub_31466A
test eax, eax
jz short loc_31307B
push [ebp+arg_0]
push edi
call sub_314623
test eax, ax
jz short loc_31307D
cmp [ebp+arg_0], ebx
jz short loc_31306F

loc_313066:
    ; CODE XREF: sub_312FD0+5;
    ; Sub_312FD0+56
    push DH
    call sub_31411B

loc_31306D:
    ; CODE XREF: sub_312FD0+5;
    ; Sub_312FD0+49
    call sub_3140F3
test eax, eax
jg short loc_31307D
call sub_3140F3
jmp short loc_31308C

loc_31307D:
    ; CODE XREF: sub_312FD0+5;
    ; Sub_312FD0+34
    call sub_3140F3
and eax, 0FFFFh
or eax, 80070000h

loc_31308C:
    ; CODE XREF: sub_312FD0+5;
    mov [ebp+var_4], eax
```
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 32-bit 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

MBE - 05/08/2015  x64, ARM, Windows
x86 Registers

- **EAX**
  - AX
  - AH
  - AL
- **EBX**
  - BX
  - BH
  - BL
- **ECX**
  - CX
  - CH
  - CL
- **EDX**
  - DX
  - DH
  - DL
- **ESI**
- **EDI**
- **ESP** (stack pointer)
- **EBP** (base pointer)

32 bits
x86 Registers

EAX (32 bits)
x86 Registers

EAX (32bits)

AX (16bits)
x86 Registers

EAX (32bits)

AX (16bits)

AH   AL

AH   AL  <---- (8bits each)
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**)

```assembly
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]
je      short loc_313066
sub     eax, [ebp+var_84]
push    esi

push    esi
push    eax
push    [ebp+arg_0]
call    sub_31466A
test    eax, eax
jz      short loc_31306D
lea     eax, [ebp+arg_0]
push    eax
mov     eax, [ebp+arg_0]
push    edi
call    sub_314623
lea     eax, [ebp+arg_0]
test    eax, eax
jz      short loc_31306D
cmp     [ebp+arg_0], esi
jz      short loc_31308F

loc_313066:
  ; CGRE XREF: sub_312FD0
  ; Sub_312FD0+51
push    0EH
call    sub_31411B

loc_31306D:
  ; CGRE XREF: sub_312FD0
  ; sub_312FD0+49
jg      short loc_31307D
call    sub_3140F3
jg      short loc_31308C
jmp     short loc_31308F

loc_31307D:
  ; CGRE XREF: sub_312FD0
  ; Sub_312FD0+51
or      eax, 80000000h

loc_31308C:
  ; CGRE XREF: sub_312FD0
mov     [ebp+var_4], eax
```

MBE - 05/08/2015  x64, ARM, Windows
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

Top 20 instructions of x86 architecture

http://www.strchr.com/x86_machine_code_statistics

MBE - 05/08/2015
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!
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
**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

**64-Bit Media and Floating-Point Registers**

- MMX0/FPR0
- MMX1/FPR1
- MMX2/FPR2
- MMX3/FPR3
- MMX4/FPR4
- MMX5/FPR5
- MMX6/FPR6
- MMX7/FPR7

**128-Bit Media Registers**

- XMM0
- XMM1
- XMM2
- XMM3
- XMM4
- XMM5
- XMM6
- XMM7
- XMM8
- XMM9
- XMM10
- XMM11
- XMM12
- XMM13
- XMM14
- XMM15

---

- 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

RAX (64bits)

RAX
x86_64 Registers

RAX (64bits)

EAX (32bits)

AL
AH

AX (16bits)
# x86_64 Registers

<table>
<thead>
<tr>
<th>Register encoding</th>
<th>Zero-extended for 32-bit operands</th>
<th>Not modified for 8-bit operands</th>
<th>Low 8-bit</th>
<th>16-bit</th>
<th>32-bit</th>
<th>64-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>AH†</td>
<td>AL</td>
<td>AX</td>
<td>EAX</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>BH†</td>
<td>BL</td>
<td>BX</td>
<td>EBX</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>CH†</td>
<td>CL</td>
<td>CX</td>
<td>ECX</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>DH†</td>
<td>DL</td>
<td>DX</td>
<td>EDX</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td>SI</td>
<td>SI</td>
<td>ESI</td>
<td>RSI</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td>DI</td>
<td>DI</td>
<td>EDI</td>
<td>RDI</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td>BP</td>
<td>BP</td>
<td>EBP</td>
<td>RBP</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td>SP</td>
<td>SP</td>
<td>ESP</td>
<td>RSP</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td>R8B</td>
<td>R8W</td>
<td>R8D</td>
<td>R8</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td>R9B</td>
<td>R9W</td>
<td>R9D</td>
<td>R9</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td>R10B</td>
<td>R10W</td>
<td>R10D</td>
<td>R10</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td>R11B</td>
<td>R11W</td>
<td>R11D</td>
<td>R11</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td>R12B</td>
<td>R12W</td>
<td>R12D</td>
<td>R12</td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td>R13B</td>
<td>R13W</td>
<td>R13D</td>
<td>R13</td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td>R14B</td>
<td>R14W</td>
<td>R14D</td>
<td>R14</td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td>R15B</td>
<td>R15W</td>
<td>R15D</td>
<td>R15</td>
</tr>
</tbody>
</table>

† 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 32bitfastcall 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
The 64bit calling convention is a lot like 32bitfastcall 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!
• **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]
je short loc_313066
sub eax, [ebp+var_84]
push esi
push esi
push eax
push edi
mov [ebp+arg_0], eax
call sub_31462A
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_31306F

loc_313066: ; CODE XREF: sub_312F00
            ; sub_312FD8+51
    push D1h
call sub_31411B

loc_31306D: ; CODE XREF: sub_312F00
            ; 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_312F00
    call sub_3140F3
    and eax, 0FFFFh
    or eax, 80070000h

loc_31308C: ; CODE XREF: sub_312F08
    mov [ebp+arg_4], eax

MBE - 05/08/2015 x64, ARM, Windows
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
...

MBE - 05/08/2015 x64, ARM, Windows
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

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

$$7 \text{ (nibbles) } \times 4 \text{ (bits) } = 28$$

$$2^{28}$$ bruteforce

0.00000000037% 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
...
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
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

<table>
<thead>
<tr>
<th>Register</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>r0-r3, r12</td>
<td>Scratch Registers: used to pass parameters and will be overwritten by subroutines</td>
</tr>
<tr>
<td>r4-r11</td>
<td>Preserved Registers: stack before using, restore before returning</td>
</tr>
<tr>
<td>r13 'sp'</td>
<td>Stack Pointer: not much use on the stack</td>
</tr>
<tr>
<td>r14 'lr'</td>
<td>Link Register: set by BL or BLX on entry of routine, overwritten by further use of BL or BLX</td>
</tr>
<tr>
<td>r15 'pc'</td>
<td>Program Counter</td>
</tr>
</tbody>
</table>

**Register Use in the ARM Procedure Call Standard**
• 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
= 00001000000010001100010110110000

Interpret bytes at 0x080462B0 as ARM

r15 = 0x080462B1
= 0000100000001000110001011010110001

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
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
Loaded DLL’s

<table>
<thead>
<tr>
<th>DLL Name</th>
<th>PI</th>
<th>Ordinal</th>
<th>Hint</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>KERNEL32.DLL</td>
<td>N/A</td>
<td>68</td>
<td>0x0044</td>
<td>CloseHandle</td>
</tr>
<tr>
<td>USER32.DLL</td>
<td>N/A</td>
<td>82</td>
<td>0x0052</td>
<td>CompareFileTime</td>
</tr>
<tr>
<td>COMDLG32.DLL</td>
<td>N/A</td>
<td>121</td>
<td>0x0079</td>
<td>CreateFileA</td>
</tr>
<tr>
<td>ADVAPI32.DLL</td>
<td>N/A</td>
<td>128</td>
<td>0x0080</td>
<td>CreateFileW</td>
</tr>
<tr>
<td>SHELL32.DLL</td>
<td>N/A</td>
<td>191</td>
<td>0x00BF</td>
<td>DeleteCriticalSection</td>
</tr>
<tr>
<td></td>
<td>N/A</td>
<td>218</td>
<td>0x00DA</td>
<td>EnterCriticalSection</td>
</tr>
<tr>
<td></td>
<td>N/A</td>
<td>261</td>
<td>0x005B</td>
<td>ExitProcess</td>
</tr>
<tr>
<td></td>
<td>N/A</td>
<td>282</td>
<td>0x011A</td>
<td>FindClose</td>
</tr>
<tr>
<td></td>
<td>N/A</td>
<td>293</td>
<td>0x0125</td>
<td>FindFirstFileW</td>
</tr>
<tr>
<td></td>
<td>N/A</td>
<td>305</td>
<td>0x0131</td>
<td>FindNextFileW</td>
</tr>
<tr>
<td></td>
<td>N/A</td>
<td>322</td>
<td>0x0142</td>
<td>FlushFileBuffers</td>
</tr>
<tr>
<td></td>
<td>N/A</td>
<td>332</td>
<td>0x014C</td>
<td>FreeEnvironmentStringsW</td>
</tr>
<tr>
<td></td>
<td>N/A</td>
<td>339</td>
<td>0x0153</td>
<td>GetACP</td>
</tr>
<tr>
<td></td>
<td>N/A</td>
<td>348</td>
<td>0x015C</td>
<td>GetCPInfo</td>
</tr>
<tr>
<td></td>
<td>N/A</td>
<td>369</td>
<td>0x0171</td>
<td>GetCommandLineW</td>
</tr>
</tbody>
</table>

MBE - 05/08/2015
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
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(),
Most people think **kernel32.dll** is required by every Windows process, but **ntdll.dll** is in fact the only one that **MUST** be loaded.
Windows Exploitation Basics

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

An exception will cause 0x7c1408ac to be called as an exception handler as:

```assembly
EXCEPTION_DISPOSITION Handler(
    PEXCEPTION_RECORD Exception,
    PVOID EstablisherFrame,
    PCONTEXT ContextRecord,
    PVOID DispatcherContext);
```
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.
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
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 (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

Valid SEH Chain

N H
app!_except_handler4

N H
k32!_except_handler4

N H
ntdll!FinalExceptionHandler

Invalid SEH Chain

N H
0x7c1408ac

N H
0x414106eb

Can’t reach final record!
Bypassing SEHOP

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