Introduction to Intel x86-64
Assembly, Architecture, Applications, & Alliteration

aka

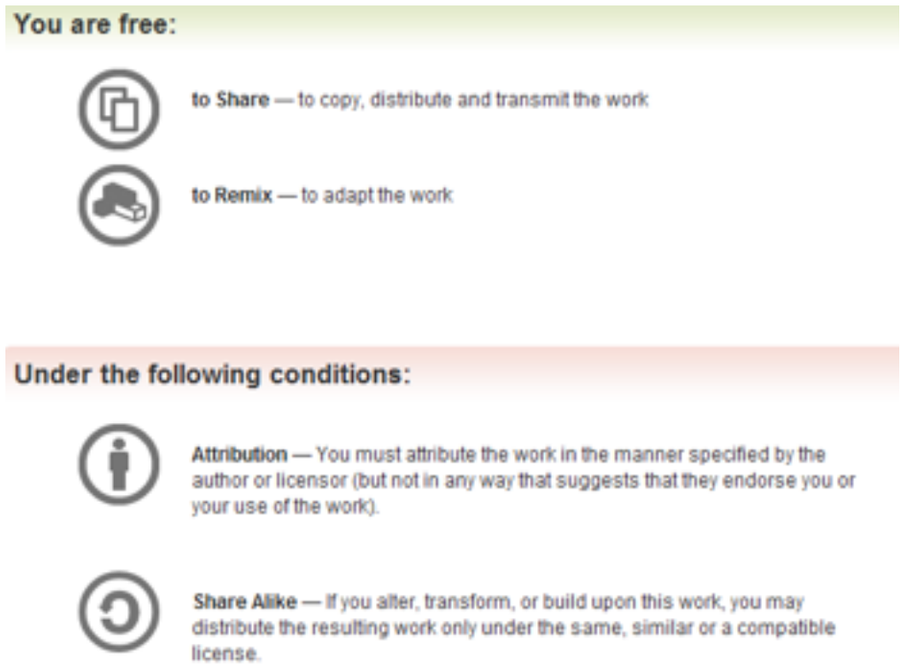
Understanding x86-64 Assembly for Reverse Engineering & Exploits

Xeno Kovah – 2014-2015

xeno@legbacore.com

All materials is licensed under a Creative Commons “Share Alike” license.

- http://creativecommons.org/licenses/by-sa/3.0/



Attribution condition: You must indicate that derivative work
"Is derived from Xeno Kovah's 'Intro x86-64' class, available at http://OpenSecurityTraining.info/IntroX86-64.html"

Attribution condition: You must indicate that derivative work

"Is derived from Xeno Kovah's ‘Intro x86-64’ class, available at http://OpenSecurityTraining.info/IntroX86-64.html"

Special Thanks To:

- Veronica Kovah & Sam Cornwell, for helping with the update for 64 bit!

Additional Content/Ideas/Info Provided By:

- Jon A. Erickson, Christian Arllen, Dave Keppler, Dillon Beresford
- Who suggested what, is inline with the material

- Your name here! Just suggest/contribute some content that ultimately makes its way into the class

Why learn x86 assembly?

The first time you see assembly language

http://securityreactions.tumblr.com/post/97147746722/the-first-time-you-see-assembly-language

Why learn x86 assembly?

What it's like when you finally understand assembly

http://securityreactions.tumblr.com/post/97147718552/what-its-like-when-you-finally-understand-assembly

Why learn x86 assembly?

- Because x86 is pervasive on PCs & servers (and you better believe that Intel is going to claw their way on to mobile ;))
- Because it's basically a given that some talk at a security conference will at some point flash some x86 assembly in order to explain what's going on. But even more talks just assume you know it and will be able to fill in the implied asm next steps.
- Because it's essential to writing memory corrupting exploits on PCs & servers
- Because it's essential to reverse engineering programs (goodware or malware) on PCs & servers
- Because there are plenty of people who know network security but those who know host-based security are more rare and therefore more valuable
- Because all the other architectures are super simple by comparison and easier to learn afterwards
- Because a lot of the top hackers who have come before you knew x86 assembly, and in order to get to where they got, you need to know what they knew

Packet Parsing

```
DFF23BCD mov     eax, [esp+arg_0_rbu_packet]
DFF23BD1 mov     eax, [eax]      ; base of RBU_Packet
DFF23BD3 mov     ds:g_foundKPR, 1
DFF23BDC mov     ecx, [eax+0Ch]
DFF23BDF mov     ds:g_pktSetId, ecx ; rbu_packet.pktSetId
DFF23BE5 movzx   edx, word ptr [eax+12h]
DFF23BE9 mov     ds:g_totPkts, edx ; rbu_packet.totPkts
DFF23BEF movzx   ecx, word ptr [eax+8]
DFF23BF3 shl     ecx, 4
DFF23BF6 mov     ds:g_hdrSize, ecx ; rbu_packet.hdrSize
DFF23BFC movzx   eax, word ptr [eax+4]
DFF23C00 shl     eax, 0Ah
DFF23C03 sub     eax, ecx
DFF23C05 dec     edx
DFF23C06 imul    edx, eax
DFF23C09 cmp     edx, 800000h
DFF23C0F mov     ds:g_pktSizeMinusHdrSize, eax ; 0x7fe0
```

- SMM first locates the RBU packet by scanning for an ASCII signature upon page aligned boundaries.
- Once located, members of the RBU packet are stored in an SMM data area for use in later calculations…

MITRE

context

BrokerCreateKnownObject

```
; Attributes: bp-based frame

; __int32 __stdcall CSIUserBrokerObject::BrokerCreateKnownObject(CSIUserBrokerObject *__hidden this, const struct _GUID *, const struct _GUID *, struct IUnknown **)
?BrokerCreateKnownObjectMCSIUserBrokerObjectUAGXGL_AGGG_GGIGAAGPAPAGIUnknownAAAZ proc near

this= dword ptr  8
rclsid= dword ptr  0Ch
riid= dword ptr  10h
ppv= dword ptr  14h

; FUNCTION CHUNK AT 10006169 SIZE 00000027 BYTES
; FUNCTION CHUNK AT 10162174 SIZE 00000005 BYTES
; FUNCTION CHUNK AT 1016225C SIZE 00000005 BYTES

mov     edi, edi
push    ebp
mov     ebp, esp
push    esi             ; struct _GUID *
mov     esi, [ebp+rclsid]
mov     ecx, offset _CLSID_CShdocvwBroker
push    edi             ; struct _GUID *
mov     edx, esi
mov     edi, 80070005h
call    ?IsEqualGUID@@YGHAAU_GUID@@0@Z ; IsEqualGUID(_GUID const &,_GUID const &)
test    eax, eax
jz      loc_10162174
```

Full System Emulation: Achieving Successful Automated Dynamic Analysis of Evasive Malware, Kruegel, BlackHat USA 2014

VM Approach versus CPU Emulation

lastline

```
callq   0x100070478     ; symbol stub for: _open
callq   0x1000704b4     ; symbol stub for: _read
callq   0x1000702b6     ; symbol stub for: _close
```

```
cmpl    $0x0c,%ebx
je      0x10000f21e
xorl    %esi,%esi
movq    %r15,%rdi
xorl    %eax,%eax
callq   0x100070478     ; symbol stub for: _open
movl    %eax,%r12d
testl   %eax,%eax
js      0x10000f21e
leaq    0xffffff70(%rbp),%rcx
movq    %rcx,0xfffffec0(%rbp)
movl    $0x00000050,%edx
movq    %rcx,%rsi
movl    %eax,%edi
callq   0x1000704b4     ; symbol stub for: _read
movq    %rax,%r13
movl    %eax,%r14d
movl    %r12d,%edi
callq   0x1000702b6     ; symbol stub for: _close
cmpl    $0x02,%r13d
jle     0x10000f21e
```

BLUE COAT

Disassembly of code run in DriverEntry

```
lea     rax, DriverUnload
mov     [rsi+68h], rax
lea     rax, Dispatch_InternalDeviceControl
xor     ecx, ecx
mov     [rsi+0E8h], rax ; Set IRP_MJ_INTERNAL_DEVICE_CONTROL
lea     rax, Dispatch_Dummy
mov     r8d, 'PedI'
mov     [rsi+70h], rax  ; Set IRP_MJ_CREATE
mov     [rsi+80h], rax  ; Set IRP_MJ_WRITE
lea     rax, Dispatch_DeviceControl
mov     [rsi+0E0h], rax ; Set IRP_MJ_DEVICE_CONTROL
lea     rax, Dispatch_Power
mov     [rsi+120h], rax ; Set IRP_MJ_POWER
lea     rax, Dispatch_PnP
mov     [rsi+148h], rax ; Set IRP_MJ_PNP
lea     rax, Dispatch_SystemControl
mov     [rsi+128h], rax ; Set IRP_MJ_SYSTEM_CONTROL
```

Exposing Bootkits with BIOS Emulation - Haukli, BlackHat USA 2014

Gauss encryption

```
mov             ecx, (LENGTHOF tToCrypt)-1
mov             edx, OFFSET tToCrypt
mov             ebx, OFFSET tEncrypt
L1:
    XOR                     eax, ACDC

    mov                     [ebx], eax
    inc                     edx

    inc                     EBX

LOOP L1

mov             edx, OFFSET tOutEncr
call            WriteString
mov             edx, OFFSET tEncrypt
call            WriteString
call            Crlf
ret
```

About Me

- Security nerd – T-Shaped![1]
- Started LegbaCore in January 2015
- Realmz ~1996, Mac OS 8, BEQ->BNE FTW!
- x86 ~2002
- Know or have known ~5 assembly languages(x86, SPARC, ARM, PPC, 68HC12). x86 is by far the most complex.
- Routinely read assembly when debugging my own code, reading exploit code, and reverse engineering things

- [1]http://www.valvesoftware.com/company/Valve_Handbook_LowRes.pdf

5 min

About You?

- What is your name?

(What are you looking to get out of the class?)

- Where do you work?
- What is your job?
- Do you know which environment you will be using this knowledge in?

10 min

About the Class

- The intent of this class is to expose you to the most commonly generated assembly instructions, and the most frequently dealt with architecture hardware.
  - 64 bit instructions/hardware
  - Implementation of a Stack
  - Common tools
- Many things will therefore be left out or deferred to later classes.
  - Floating point instructions/hardware
  - 16 bit instructions/hardware
  - Complicated or rare instructions
  - Instruction pipeline, caching hierarchy, alternate modes of operation, hw virtualization, etc (see other classes for those)

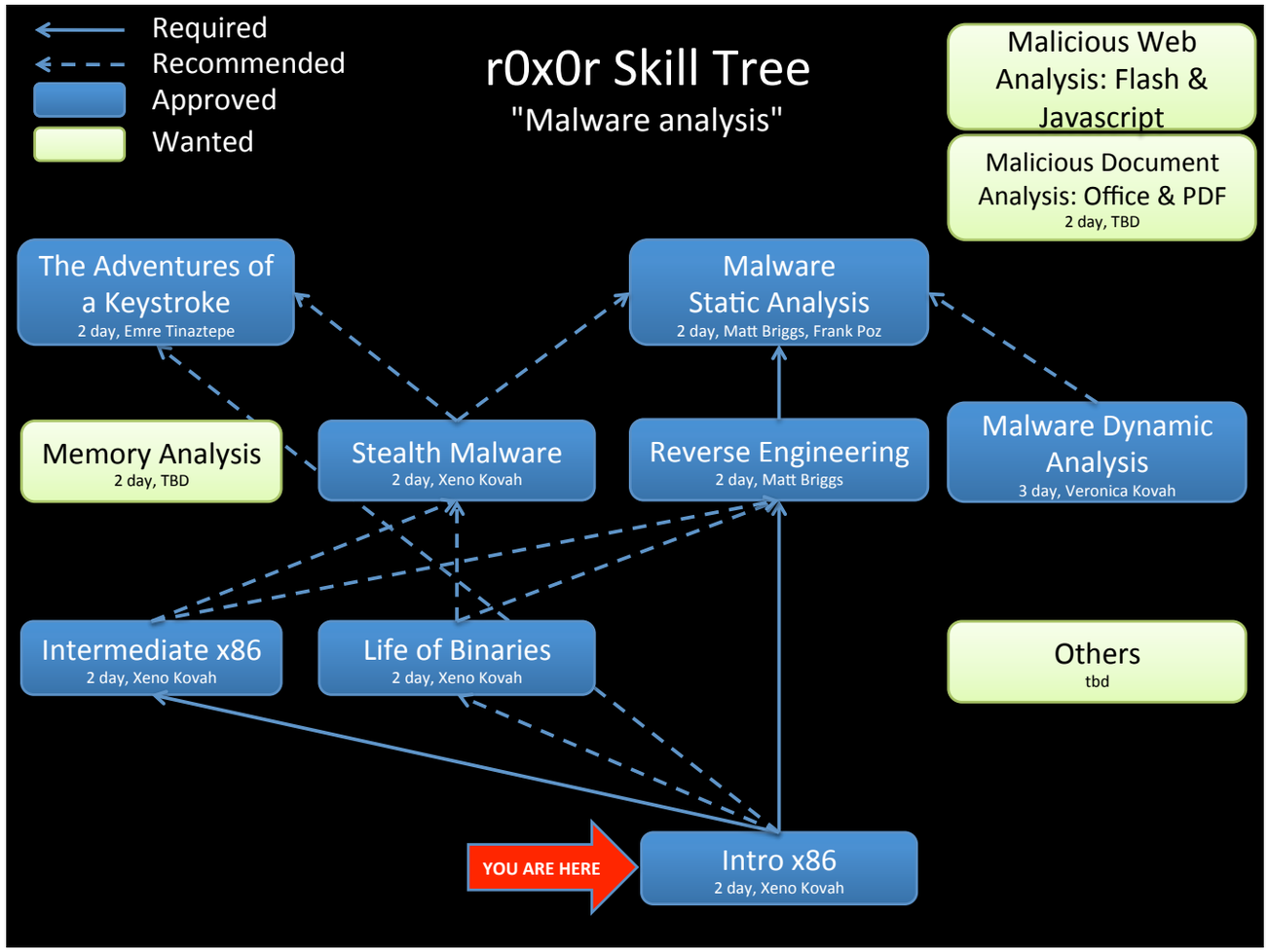
2 min

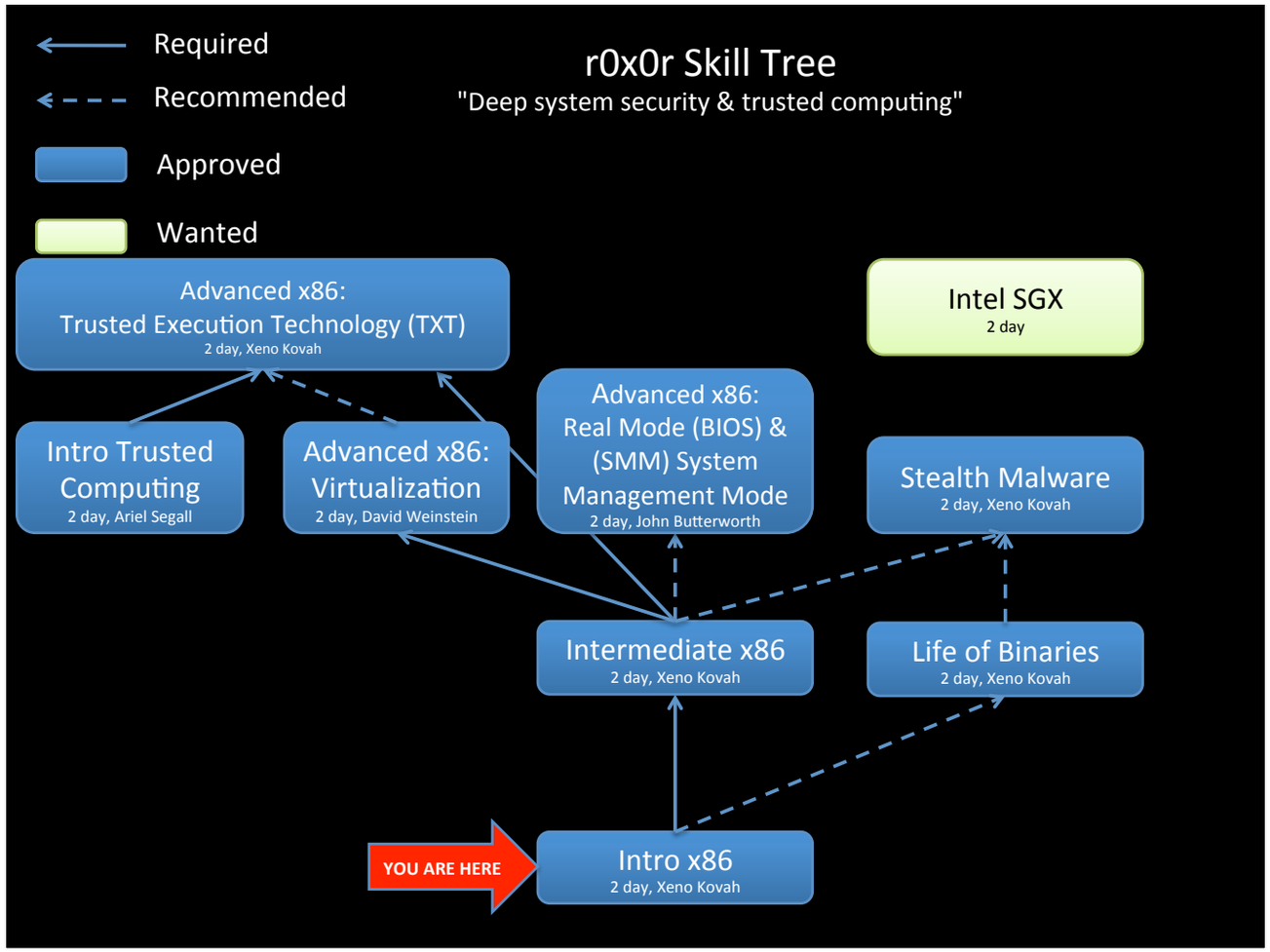
About the Class 2

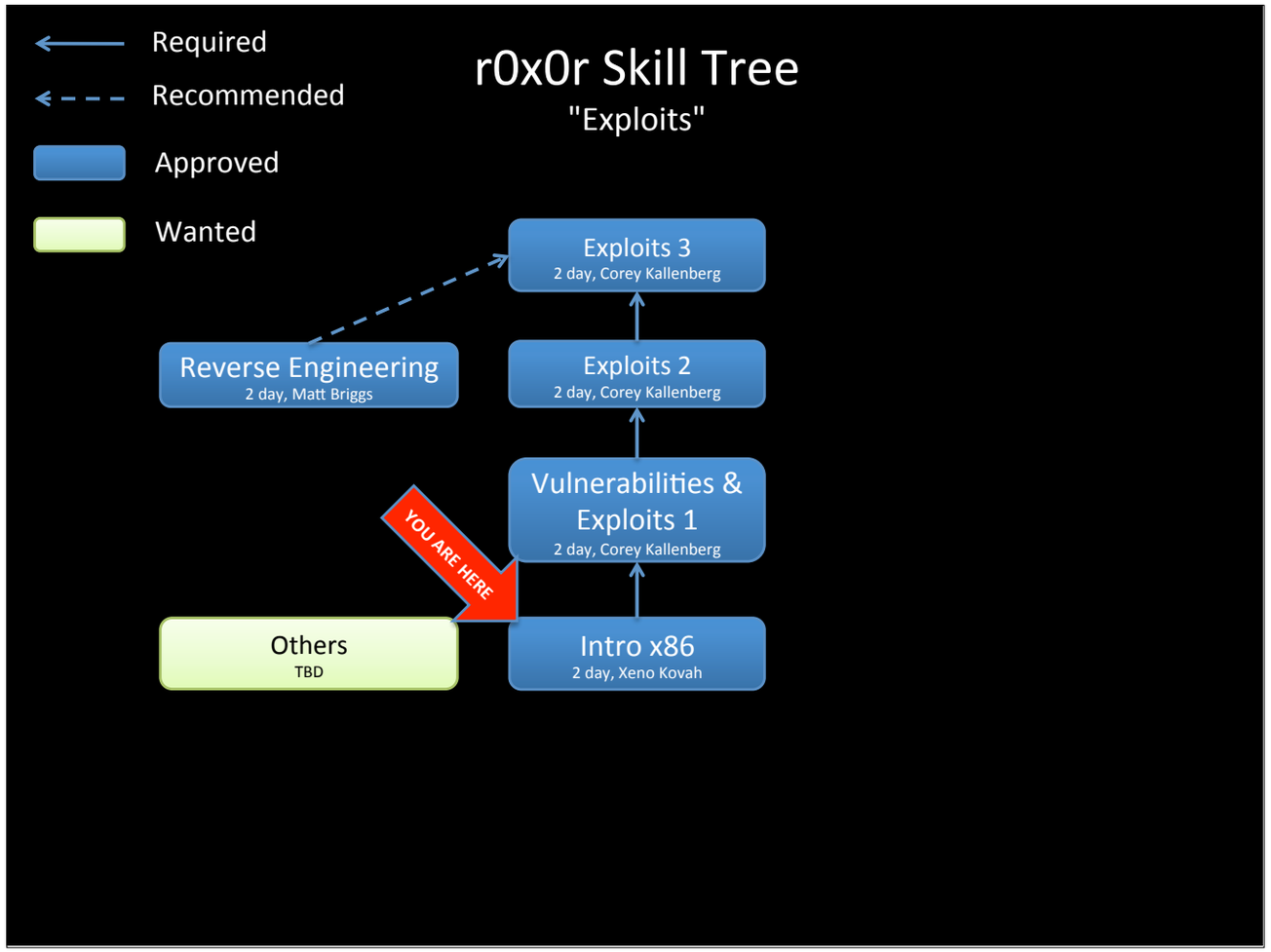
- The hope is that the material covered will be provide the required background to delve deeper into areas which may have seemed daunting previously.
- Because I can’t anticipate the needs of all job classes, if there are specific areas which you think would be useful to certain job types, let me know. The focus areas are currently primarily influenced by my security background, but I would like to make the class as widely applicable as possible.

When you're "done" with this class… you're not done. You've just begun.

- I want peers, not peons
- I want people who can do what I can do, and ultimately exceed me
  - I need people who are better than me to compete against, in order to get better myself
- Therefore I'm trying to teach as many people what I know as possible
- To this end I started OpenSecurityTraining.info
- And I *highly* recommend you continue your education there once this class is done


Required
Recommended
Approved
Wanted
r0x0r Skill Tree
"Malware analysis"
Malicious Web Analysis: Flash & Javascript
Malicious Document Analysis: Office & PDF
2 day, TBD
The Adventures of a Keystroke
2 day, Emre Tinaztepe
Malware Static Analysis
2 day, Matt Briggs, Frank Poz
Memory Analysis
2 day, TBD
Stealth Malware
2 day, Xeno Kovah
Reverse Engineering
2 day, Matt Briggs
Malware Dynamic Analysis
3 day, Veronica Kovah
Intermediate x86
2 day, Xeno Kovah
Life of Binaries
2 day, Xeno Kovah
Others
tbd
YOU ARE HERE
Intro x86
2 day, Xeno Kovah


Required
Recommended
Approved
Wanted
r0x0r Skill Tree
"Deep system security & trusted computing"
Advanced x86:
Trusted Execution Technology (TXT)
2 day, Xeno Kovah
Intel SGX
2 day
Intro Trusted
Computing
2 day, Ariel Segall
Advanced x86:
Virtualization
2 day, David Weinstein
Advanced x86:
Real Mode (BIOS) &
(SMM) System
Management Mode
2 day, John Butterworth
Stealth Malware
2 day, Xeno Kovah
Intermediate x86
2 day, Xeno Kovah
Life of Binaries
2 day, Xeno Kovah
YOU ARE HERE
Intro x86
2 day, Xeno Kovah

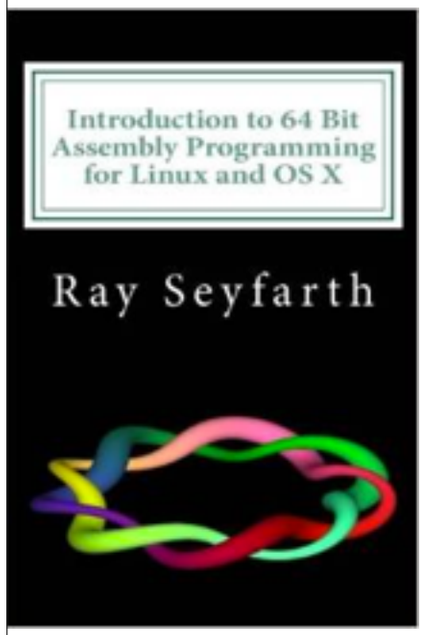

Required
Recommended
Approved
Wanted
r0x0r Skill Tree
"Exploits"
Exploits 3
2 day, Corey Kallenberg
Reverse Engineering
2 day, Matt Briggs
Exploits 2
2 day, Corey Kallenberg
Vulnerabilities & Exploits 1
2 day, Corey Kallenberg
YOU ARE HERE
Others
TBD
Intro x86
2 day, Xeno Kovah

Agenda

- Day 1 - Part 1 - Architecture Introduction, Windows tools
- Day 1 - Part 2 - Windows Tools & Analysis, Learning New Instructions
- Day 2 - Part 1 - Linux Tools & Analysis
- Day 2 - Part 2 - Inline Assembly, Read The Fun Manual, Choose Your Own Adventure

2 min

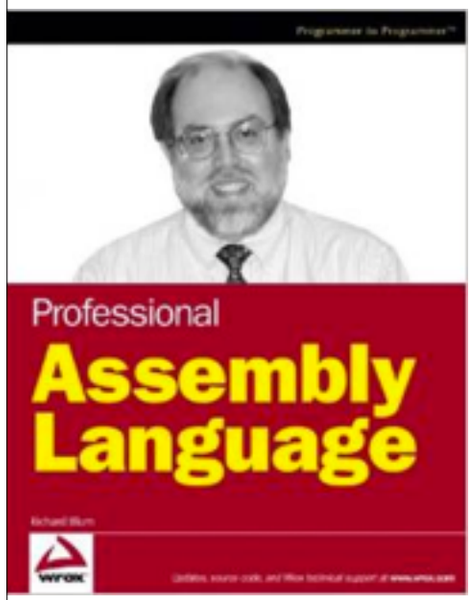
Book (64 bit)



- "Introduction to 64 Bit Assembly Programming for Linux and OS X: Third Edition" by Ray Seyfarth
- Optional book for the class, to give you alternative explanations to my own
- When you see "*Book*" page references in the bottom of slides, it is referring to this book.

2 min

Book (32 bit)



- “Professional Assembly Language” by Richard Blum.
- This optional book was originally picked after the creation of the 32 bit class because it uses AT&T assembly syntax & linux as an example, in contrast to the majority of my class which is Intel syntax & Windows
- Therefore it just serves as an alternative source of explanation in case something from the class isn’t clear and you want a second opinion

2 min

Miss Alaineous

- Questions: Ask ‘em if you got ‘em
  - If you fall behind and get lost and try to tough it out until you understand, it’s more likely that you will stay lost, so ask questions ASAP.
- Browsing the web and/or checking email during class is a good way to get lost
- 2 hours, 10 min break, 1.5 hours, lunch, 1 hour/5 min break after that
- It’s called x86 because of the progression of Intel chips from 8086, 80186, 80286, etc. I just had to get that out of the way. :)

Miss Alaineous 2

- Intel originally wanted to break from x86 when moving to 64 bit. This was Itanium aka IA64 (Intel Architecture 64 bit). However, AMD decided to extend x86 to 64 bits itself, leading to the AMD64 architecture. When Itanium had very slow adoption, Intel decided to bite the bullet and license the 64 bit extensions from AMD.
- In the Intel manuals you will see the 64 bit extensions referred to as IA32e or EMT64 or Intel 64 (but never IA64. Again, that's Itanium, a completely different architecture).
- You might sometimes see it called amd64 or x64 by MS or some linux distributions
- In this class we're going to go with x86-64

What you’re going to learn:

```
#include <stdio.h>
int main(){
    printf(“Hello World!\n”);
    return 0x1234;
}
```

Is the same as…

```
main:
000000013F511000  sub         rsp,28h
000000013F511004  lea         rcx,[__globallocalestatus-10h (13F513000h)]
000000013F51100B  call        qword ptr [__imp_printf (13F512100h)]
000000013F511011  mov         eax,1234h
000000013F511016  add         rsp,28h
000000013F51101A  ret
```

Windows Visual C++ 2012 Express
/GS (buffer overflow protection) option turned off
Disassembled with Visual C++

which could be viewed as…

```
0000000140001000 <.text>:
   140001000:   48 83 ec 28             sub    $0x28,%rsp
   140001004:   48 8d 0d ad 11 00 00    lea    0x11ad(%rip),%rcx # 0x1400021b8
   14000100b:   ff 15 07 11 00 00       callq  *0x1107(%rip)     # 0x140002118
   140001011:   b8 34 12 00 00          mov    $0x1234,%eax
   140001016:   48 83 c4 28             add    $0x28,%rsp
   14000101a:   c3                      retq
```

Windows Visual C++ 2012 Express
/GS (buffer overflow protection) option turned off
Disassembled with objdump -d from cygwin

which is equivalent to…

```
08048374 <main>:
 8048374:       8d 4c 24 04                   lea    0x4(%rsp),%rcx
 8048378:       83 e4 f0                      and    $0xfffffff0,%rsp
 804837b:       ff 71 fc                      pushl  -0x4(%rcx)
 804837e:       55                            push   %rbp
 804837f:       89 e5                         mov    %rsp,%rbp
 8048381:       51                            push   %rcx
 8048382:       83 ec 04                      sub    $0x4,%rsp
 8048385:       c7 04 24 60 84 04 08          movl   $0x8048460,(%rsp)
 804838c:       e8 43 ff ff ff                call   80482d4 <puts@plt>
 8048391:       b8 2a 00 00 00                mov    $0x1234,%eax
 8048396:       83 c4 04                      add    $0x4,%rsp
 8048399:       59                            pop    %rcx
 804839a:       5d                            pop    %rbp
 804839b:       8d 61 fc                      lea    -0x4(%rcx),%rsp
 804839e:       c3                            ret
 804839f:       90                            nop
```

Ubuntu 12.04, GCC 4.2.4
Disassembled with “objdump -d”

which is equivalent to…

```
_main:
0000000100000f40      pushq   %rbp
0000000100000f41      movq    %rsp, %rbp
0000000100000f44      subq    $0x10, %rsp
0000000100000f48      leaq    0x3f(%rip), %rdi ## literal pool for:
   "Hello World!"
0000000100000f4f      movl    $0x0, -0x4(%rbp)
0000000100000f56      movb    $0x0, %al
0000000100000f58      callq   0x100000f6e ## symbol stub for: _printf
0000000100000f5d      movl    $0x1234, %ecx
0000000100000f62      movl    %eax, -0x8(%rbp)
0000000100000f65      movl    %ecx, %eax
0000000100000f67      addq    $0x10, %rsp
0000000100000f6b      popq    %rbp
0000000100000f6c      ret
```

Mac OS 10.9.4, Apple LLVM version 5.1 (clang-503.0.40)
Disassembled from command line with “otool -tV”

But it all boils down to…

```
.text:0000000140001000 main
.text:0000000140001000
.text:0000000140001000 sub     rsp, 28h
.text:0000000140001004 lea     rcx, Format     ; "Hello World!\n"
.text:000000014000100B call    cs:__imp_printf
.text:0000000140001011 mov     eax, 1234h
.text:0000000140001016 add     rsp, 28h
.text:000000014000101A retn
```

Windows Visual C++ 2012, /GS (buffer overflow protection) option turned off
Optimize for minimum size (/O1) turned on
Disassembled with IDA Pro 6.6 (with some omissions for fitting on screen)

Take Heart!

- By one measure, only 14 assembly instructions account for 90% of code!
  - http://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Bilar.pdf
- I think that knowing about 20-30 (not counting variations) is good enough that you will have the check the manual very infrequently
- You've already seen 10 instructions, just in the hello world variations!