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"
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

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

- 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...
BrokerCreateKnownObject

Digging for Sandbox Escapes - Forshaw, BlackHat USA 2014
Full System Emulation: Achieving Successful Automated Dynamic Analysis of Evasive Malware, Kruegel, BlackHat USA 2014
Exposing Bootkits with BIOS Emulation - Haukli, BlackHat USA 2014
About Me

• Security nerd – T-Shaped!
• 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

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?
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)
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
r0x0r Skill Tree
"Malware analysis"

- The Adventures of a Keystroke
  2 day, Emre Tinaztepe
- Memory Analysis
  2 day, TBD
- Intermediate x86
  2 day, Xeno Kovah
- Life of Binaries
  2 day, Xeno Kovah
- Stealth Malware
  2 day, Xeno Kovah
- Malware Static Analysis
  2 day, Matt Briggs, Frank Pou
- Reverse Engineering
  2 day, Matt Briggs
- Malware Dynamic Analysis
  3 day, Veronica Kovah
- Malicious Web Analysis: Flash & Javascript
  2 day, TBD
- Malicious Document Analysis: Office & PDF
  2 day, TBD
- Malicious Web Analysis: Flash & Javascript
  2 day, TBD
- Malicious Document Analysis: Office & PDF
  2 day, TBD

Required
Recommended
Approved
Wanted

YOU ARE HERE

Intro x86
2 day, Xeno Kovah
r0x0r Skill Tree
"Deep system security & trusted computing"

Advanced x86: Trusted Execution Technology (TXT)
2 day, Ariel Segall

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

Intermediate x86
2 day, Xeno Kovah

Stealth Malware
2 day, Xeno Kovah

Life of Binaries
2 day, Xeno Kovah

Intel SGX
2 day

YOU ARE HERE
r0x0r Skill Tree
"Exploits"

- **Required**
- **Recommended**
- **Approved**
- **Wanted**

- **Exploits 3**
  - 2 day, Corey Kallenberg

- **Exploits 2**
  - 2 day, Corey Kallenberg

- **Vulnerabilities & Exploits 1**
  - 2 day, Corey Kallenberg

- **Intro x86**
  - 2 day, Xeno Kouh

- **Reverse Engineering**
  - 2 day, Matt Briggs

- **Others**
  - TBD

YOU ARE HERE
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
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.
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.
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. :)


• 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:

```c
#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...

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>140001000</td>
<td><code>sub $0x28,%rsp</code></td>
<td></td>
</tr>
<tr>
<td>140001004</td>
<td><code>lea 0x11ad(%rip),%rcx</code></td>
<td># 0x1400021b8</td>
</tr>
<tr>
<td>14000100b</td>
<td><code>callq *0x1107(%rip)</code></td>
<td># 0x140002118</td>
</tr>
<tr>
<td>140001011</td>
<td><code>mov $0x1234,%eax</code></td>
<td></td>
</tr>
<tr>
<td>140001016</td>
<td><code>add $0x28,%rsp</code></td>
<td></td>
</tr>
<tr>
<td>14000101a</td>
<td><code>retq</code></td>
<td></td>
</tr>
</tbody>
</table>

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

```assembly
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 %rbp, %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, %rbp
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…

.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!

- 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!