Intro to x64 Reversing

SummerCon 2011 - NYC
Jon Larimer
email: jlarimer@gmail.com
twitter: @shydemeanor
Before we begin...

- This presentation assumes you can reverse x86 code
- You might learn something even if you can't, so don't leave
- If I go to fast, yell at me
- Find a mistake, I drink

**THERE WILL BE A QUIZ!**
- If you answer wrong, you drink
Agenda

• Intro / History of x64
• The x64 Platform
• Microsoft x64 ABI
• SysV x64 ABI
• Tools for reversing x64
x64 reversing challenges

• If you're used to reversing 32 bit x86 code, x64 can be confusing at first

• Easy parts
  • Instructions are mostly the same as you're used to
  • There are a few more registers

• Hard parts
  • Calling convention is totally different
  • Debugging optimized code can be tricky
Name soup!

- AMD
  - x86-64
  - AMD64
- Intel
  - IA-32e
  - EM64T
  - Intel 64
- Oracle/Microsoft
  - x64

- BSD - amd64
- Linux kernel - x86_64
- GCC - amd64
- Debian/Ubuntu - amd64
- Fedora/SuSE - x86_64
- Solaris - amd64

Note: IA-64 is Itanium, NOT x86-x64!
History of x64

- **1999** - AMD announces x86-64
- **2000** - AMD releases specs
- **2001** - First x86-64 Linux kernel available
- **2003** - First AMD64 Operton released
- **2004** - Intel announces IA-32e/EM64T, releases first x64 Xeon processor
- **2005** - x64 versions of Windows XP and Server 2003 released
- **2009** - Mac OS 10.6 (Snow Leopard) includes x64 kernel
- **2009** - Windows Server 2008 R2 only available in x64 version
- **2010** - 50% of Windows 7 installs running the x64 version
- **2011** - 40% of Steam users in April 2011 HW survey use Win7 x64
The x64 Platform
What is x64?

- Extension to 32 bit x86 - x64 “long mode”
  - Can address up to 64 bits (16EB) of virtual memory*
  - Can address up to 52 bits (4PB) of physical memory**
- 64 bit general purpose registers - \texttt{RAX}, \texttt{RBX}, ...
- 8 new GP registers (\texttt{R8-R15})
- 8 new 128 bit XMM registers (\texttt{XMM8-XMM15})
- New 64 bit instructions: \texttt{cdqe}, \texttt{lodsq}, \texttt{stosq}, etc
- Ability to reference data relative to instruction pointer (\texttt{rip})

* Limited by processor implementation, most only support 48 bits now...
** Intel currently supports 40 bits of physical memory
Long mode

• 64 bit flat (linear) addressing
  • Segment base is always 0 except for FS and GS
  • Stack (SS), Code (CS), Data (DS) always in the same segment
• Default address size is 64 bits
• Default operand size is 32 bits
  • 64 bit operands (RAX, RBX, ...) are specified with “REX prefix” in the opcode encoding
• 64 bit instruction pointer (RIP)
• 64 bit stack pointer (RSP)
Canonical addresses

- Current implementations only support 48 bit linear addresses
- Canonical form means most significant bit of address is extended to bit 63
  - Bits 0-47 are the address, bits 48-63 are the same as bit 47
- Windows uses high addresses for kernel, low addresses for user mode
- Non-canonical address access results in #GP
x64 registers

• 32 bit registers extended to 64 bits
  • eax → rax
  • ebx → rbx
  • esp → rsp

• 8 additional 64 bit registers
  • r8, r9, r10, ... r15

• 8 additional 128 bit XMM (SSE) registers
  • xmm8, xmm9, ... xmm15

• Used for vector and floating point arithmetic
Intel/AMD  AVX

- AVX is **Advanced Vector eXtension**
- Adds 8 256 bit registers
  - $\text{ymm0-ymm7}$
- Low 128 bits of AVX registers overlap with XMM (SSE) registers
  - $\text{xmm0-xmm7}$
- Also a few new instructions
- First CPUs with AVX were the Intel **Sandy Bridge** processors released Q1 2011
# x64 Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>63</th>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAX</td>
<td></td>
<td>EAX</td>
<td></td>
</tr>
<tr>
<td>RBX</td>
<td></td>
<td>EBX</td>
<td></td>
</tr>
<tr>
<td>RCX</td>
<td></td>
<td>ECX</td>
<td></td>
</tr>
<tr>
<td>RDX</td>
<td></td>
<td>EDX</td>
<td></td>
</tr>
<tr>
<td>RBP</td>
<td></td>
<td>EBP</td>
<td></td>
</tr>
<tr>
<td>RSI</td>
<td></td>
<td>ESI</td>
<td></td>
</tr>
<tr>
<td>RDI</td>
<td></td>
<td>EDI</td>
<td></td>
</tr>
<tr>
<td>RSP</td>
<td></td>
<td>ESP</td>
<td></td>
</tr>
<tr>
<td>R8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R11</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R13</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R14</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R15</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

RIP and EIP are used to track program execution. RFLAGS is used for status flags. The top half of RFLAGS is reserved and always 0.

NOTE: Top half of RFLAGS is reserved, always 0

= new in x64
### Register operation in x64 mode

<table>
<thead>
<tr>
<th></th>
<th>63</th>
<th>31</th>
<th>15</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EAX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AH</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **zero-extended**
- **not modified**

<table>
<thead>
<tr>
<th></th>
<th>63</th>
<th>31</th>
<th>15</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>R8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R8D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R8W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R8B/R8L</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **zero-extended**
- **not modified**
POP QUIZ #1!

- How many bits is \textbf{R9D}?  
- How many bits is \textbf{RSP}?  
- How many bits is \textbf{R12W}?  
- How many bits is \textbf{R10B}?  
- How many bits is \textbf{R16}?  
POP QUIZ #1!

• How many bits is **R9D**? **32**
• How many bits is **RSP**?
• How many bits is **R12W**?
• How many bits is **R10B**?
• How many bits is **R16**?
POP QUIZ #1!

• How many bits is **R9D**? 32
• How many bits is **RSP**? 64
• How many bits is **R12W**?
• How many bits is **R10B**?
• How many bits is **R16**?
POP QUIZ #1!

- How many bits is **R9D**? 32
- How many bits is **RSP**? 64
- How many bits is **R12W**? 16
- How many bits is **R10B**?
- How many bits is **R16**?
POP QUIZ #1!

- How many bits is R9D? 32
- How many bits is RSP? 64
- How many bits is R12W? 16
- How many bits is R10B? 8
- How many bits is R16?
POP QUIZ #1!

- How many bits is R9D? 32
- How many bits is RSP? 64
- How many bits is R12W? 16
- How many bits is R10B? 8
- How many bits is R16? Not a register...
POP QUIZ #2!

• What's in **RAX** after each instruction?

  MOV  **RAX**,  1111111111111111h

  INC  **AL**

  INC  **AX**

  INC  **EAX**
POP QUIZ #2!

- What's in RAX after each instruction?

MOV RAX, 1111111111111111h
RAX = 0x1111111111111111

INC AL

INC AX

INC EAX
POP QUIZ #2!

• What's in **RAX** after each instruction?

  MOV  **RAX**,  `1111111111111111h`
  RAX = 0x1111111111111111

  INC  **AL**
  RAX = 0x1111111111111112

  INC  **AX**

  INC  **EAX**
POP QUIZ #2!

What's in \textbf{RAX} after each instruction?

\begin{align*}
\text{MOV } & \textbf{RAX}, \quad 1111111111111111h \\
& \textbf{RAX} = 0x1111111111111111 \\
\text{INC } & \textbf{AL} \\
& \textbf{RAX} = 0x1111111111111112 \\
\text{INC } & \textbf{AX} \\
& \textbf{RAX} = 0x1111111111111113 \\
\text{INC } & \textbf{EAX}
\end{align*}
POP QUIZ #2!

- What's in \textbf{RAX} after each instruction?

\begin{itemize}
  \item \textbf{MOV} \textbf{RAX}, \textbf{1111111111111111h}  
    \textbf{RAX} = \textbf{0x1111111111111111}
  \item \textbf{INC} \textbf{AL}  
    \textbf{RAX} = \textbf{0x1111111111111112}
  \item \textbf{INC} \textbf{AX}  
    \textbf{RAX} = \textbf{0x1111111111111113}
  \item \textbf{INC} \textbf{EAX}  
    \textbf{RAX} = \textbf{0x0000000011111114}
\end{itemize}
64 bit instructions

- **CDQE** Convert doubleword to quadword (sign-extend **EAX** into **RAX**)
- **CMPSQ** Compare qword at **RSI** with qword at **RDI**
- **CMPXCHG16B** Compare **RDX**:**RAX** with **m128**
- **LODSQ** Load qword at address **RSI** into **RAX**
- **MOVSQ** Move qword from address **RSI** to **RDI**
- **MOVZX** zero-extend doubleword to quadword
- **STOSQ** Store **RAX** at address **RDI**
- **SYSCALL** Fast system call, replacement for **SYSENTER**
- **SYSRET** Fast system call, replacement for **SYSEXIT**
RIP-relative addressing

- Instruction-pointer-relative operands only used for jumps/branches in x86
  - Can't access EIP register explicitly in instructions
- Can be used for data access in x64 now:
  - `mov rax, qword ptr [rip+0x1000]`
- Faster loading of position-independent code
  - Windows: Fewer base relocations in PE files
  - Linux: No GOT pointer setup in function prologue
  - No pre-linking and no performance hit for ASLR on x64
RIP-relative addressing

IDA has “Explicit RIP addressing” mode in analysis options so you can see when rip-relative addresses are used:
Application Binary Interface

- The ABI describes how to call functions
  - Passing parameters
  - Return value
  - Stack frame
  - Exceptions
- "Calling convention"

- There are two widely used x64 ABIs:
  - Microsoft's x64 ABI (Windows)
  - SysV x64 ABI (Linux, BSD, Mac)
Microsoft x64 ABI
Microsoft x64 ABI

- There's only one calling convention (no cdecl/stdcallfastcall)
- Calling convention modeled afterfastcall
  - First 4 parameters passed in registers, rest on stack
  - Return in RAX or XMM0
- Some registers are considered volatile across function calls, some are not
  - A function needs to save non-volatile registers if it uses them
MS x64 ABI: Parameters & Return

• First four parameters passed in registers
  - **RCX, RDX, R8, R9** for integers
  - **XMM0, XMM1, XMM2, XMM3** for floats
    - For variable arguments (varargs), floating point values are stored in the floating point and integer registers!
• 1:1 correspondence between parameters and registers
  - i.e., Parameter 2 is always **RDX** or **XMM1**
  - Any parameter > 8 bytes passed by reference (no splitting)

• Additional parameters on stack

• Return value in **RAX** or **XMM0**
  - **XMM0** used for floats, doubles, and 128 bit types (**_m128**)
**MS x64 ABI: struct parameters**

- If a `struct` can be packed into 8 bytes, it's passed in a register
  - Or on the stack if it's the 5th+ argument
- All `structs` over 8 bytes are passed by reference
- **Caller** allocates space and copies the `struct` before passing to the **callee**
  - This is to avoid problems with the **callee** modifying the **caller's** copy
MS x64 ABI: Parameters

Calling a function:

```
RAX  func(  RCX    RDX    R8     R9
           XMM0, XMM1, XMM2, XMM3, [rsp+20h], [rsp+28h], ...);
```

Inside called function:

```
; [rsp] holds return address
mov  [rsp+8h], ecx ; 1st param
mov  [rsp+10h], edx ; 2nd param
mov  [rsp+18h], r8 ; 3rd param
mov  [rsp+20h], r9 ; 4th param
; 5th parameter is [rsp+28h]
; 6th parameter is [rsp+30h]
sub  rsp, 28h
; now 1st parameter is [rsp+30h]
```

Yellow = integers
Green = floats
Cyan = both

Usually only see first four parameters stored in home space in debug code
MS x64 ABI: Params example

```c
printf("%i %f %i %i %f\r\n", 1, 2.0, -4, 60, 5.5);
```
MS x64 ABI: struct param example

In this example, the structure is passed by reference, but a new copy is created on the stack for the called function
POP QUIZ #3

• What registers are used for the first four integer parameters of a function?

• True/False: If a structure has two 64 bit values, it can be passed to a function split across two registers (i.e., r8 and r9)
POP QUIZ #3

• What registers are used for the first four integer parameters of a function?

  \textbf{ECX, EDX, R8, R9}

• True/False: If a structure has two 64 bit values, it can be passed to a function split across two registers (i.e., \texttt{r8} and \texttt{r9})
POP QUIZ #3

• What registers are used for the first four integer parameters of a function?
  
  \textbf{ECX, EDX, R8, R9}

• True/False: If a structure has two 64 bit values, it can be passed to a function split across two registers (i.e., \texttt{r8} and \texttt{r9})
  
  • \textbf{FALSE!}
Some registers are volatile and can be destroyed by functions:
- `RAX`, `RCX`, `RDX`, `R8`, `R9`, `R10`, `R11`
- You can't rely on them being the same after calling a function (the compiler might be able to...)

Some registers are non-volatile and must be saved by functions that use them:
- `RBX`, `RBP`, `RDI`, `RSI`, `R12`, `R13`, `R14`, `R15`
- You can rely on them being the same after calling a function
- A function that needs these registers must save them to the stack and pop them off before returning
MS x64 ABI: The stack

- Function prologue needs to allocate stack space for saved registers, local variables, arguments to callees
- Parameters are always at bottom of stack, right above return address
  - There's always space for 4 parameters, even if they're not used (home space)
- Stack is always 16 byte aligned
  - This means address ends in zero hex
  - Except within prologue
  - Unless the function doesn't call any other functions
- All memory beyond RSP is volatile (could be used by the OS or a debugger)
- No frame pointer (i.e., no mov rbp, esp in prologue) unless stack is dynamically allocated (alloca)
MS x64 ABI: Stack home space

- **Caller**'s prologue allocates stack space for arguments to **callee**

- For non-leaf functions, space for four arguments is always allocated \((4 \times 8\text{ bytes} = 32 \text{ = 0x20})\)
  - `sub esp, 0x20`
  - Keep in mind that after this instruction, stack needs to be aligned on 16 byte boundary (end in 0 hex)
    - So you'll usually see `sub esp, 0x28` instead

- In debug code, the **callee** usually puts the register parameters there in the prologue

- In optimized, code, all bets are off, **callee** can do whatever it wants
MS x64 ABI: Stack diagram

```
13FD81080 ; int __stdcall WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nShowCmd)
13FD81080 WinMain proc near ; CODE XREF: __tmainCRTStartup+14Flp
13FD81080
13FD81080
13FD81080 lpText = qword ptr -18h
13FD81080 hInstance = qword ptr 8
13FD81080 hPrevInstance = qword ptr 10h
13FD81080 lpCmdLine = qword ptr 18h
13FD81080 nShowCmd = dword ptr 20h
13FD81080
13FD81080 mov [rsp+nShowCmd], r9d
13FD81085 mov [rsp+lpCmdLine], r8 ; save caller's parameter registers even
13FD81085 ; though it's not totally necessary here,
13FD81085 ; but it can help with debugging...
13FD8108A mov [rsp+hPrevInstance], rdx
13FD8108F mov [rsp+hInstance], rcx
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1008</td>
<td>return addr</td>
</tr>
<tr>
<td>1010</td>
<td>rcx</td>
</tr>
<tr>
<td>1018</td>
<td>rdx</td>
</tr>
<tr>
<td>1020</td>
<td>r8</td>
</tr>
<tr>
<td>1028</td>
<td>r9</td>
</tr>
<tr>
<td>(rsp)</td>
<td></td>
</tr>
<tr>
<td>(rsp+08)</td>
<td>hInstance</td>
</tr>
<tr>
<td>(rsp+10)</td>
<td>hPrevInstance</td>
</tr>
<tr>
<td>(rsp+18)</td>
<td>lpCmdLine</td>
</tr>
<tr>
<td>(rsp+20)</td>
<td>nShowCmd</td>
</tr>
</tbody>
</table>
mov [rsp+nShowCmd], r9d
mov [rsp+lpCmdLine], r8 ; save caller's parameter registers even
t; though it's not totally necessary here,
b; but it can help with debugging...
tsub rsp, 38h ; save 0x38 bytes on stack for callee args,
t; local var, and alignment
tmov r9d, 4 ; 4th argument in r9
tmov r8d, 3 ; 3rd argument in r8
tmov edx, 2 ; second argument in edx
tmov ecx, 1 ; first argument in ecx
tcall get_string
tmov [rsp+38h+lpText], rax ; return value from get_string is in rax,
t; save it as local variable lpText
MS x64 ABI: Stack diagram

```assembly
13FDB1000 ; int __cdecl get_string(int arg_a, int arg_b, int arg_c, int arg_d)
13FDB1000 get_string proc near ; CODE XREF: WinMain+2E↓p
13FDB1000
13FDB1000
13FDB1000 var_38 = dword ptr -38h
13FDB1000 var_30 = dword ptr -30h
13FDB1000 var_28 = dword ptr -28h
13FDB1000 var_18 = qword ptr -18h
13FDB1000 arg_a = dword ptr 8
13FDB1000 arg_b = dword ptr 10h
13FDB1000 arg_c = dword ptr 18h
13FDB1000 arg_d = dword ptr 20h
13FDB1000
13FDB1000 mov [rsp+arg_d], r9d
13FDB1005 mov [rsp+arg_c], r8d
13FDB100A mov [rsp+arg_b], edx
13FDB100E mov [rsp+arg_a], ecx
13FDB1012 sub rsp, 58h
```
### MS x64 ABI: Stack diagram

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0FD8</td>
<td>return addr</td>
<td>(rsp) home space</td>
</tr>
<tr>
<td>0FE0</td>
<td>ecx</td>
<td>(rsp+8) arg_a</td>
</tr>
<tr>
<td>0FD8</td>
<td>edx</td>
<td>(rsp+10) arg_b</td>
</tr>
<tr>
<td>0FE0</td>
<td>r8</td>
<td>(rsp+18) arg_c</td>
</tr>
<tr>
<td>0FE8</td>
<td>r9</td>
<td>(rsp+20) arg_d</td>
</tr>
<tr>
<td>OFF0</td>
<td>lpText</td>
<td>(rsp+28) lpCmdLine</td>
</tr>
<tr>
<td>OFF8</td>
<td>?? ??</td>
<td>(rsp+30) ?? ??</td>
</tr>
<tr>
<td>1000</td>
<td>?? ??</td>
<td>(rsp+38) ?? ??</td>
</tr>
<tr>
<td>1008</td>
<td>return addr</td>
<td>(rsp+40) (return to _tmainCRT..)</td>
</tr>
<tr>
<td>1010</td>
<td>rcx</td>
<td>(rsp+48) hInstance</td>
</tr>
<tr>
<td>1018</td>
<td>rdx</td>
<td>(rsp+50) hPrevInstance</td>
</tr>
<tr>
<td>1020</td>
<td>r8</td>
<td>(rsp+58) lpCmdLine</td>
</tr>
<tr>
<td>1028</td>
<td>r9</td>
<td>(rsp+60) nShowCmd</td>
</tr>
</tbody>
</table>
MS x64 ABI: Stack Example #2

- Optimized code
- Note that the **WinMain** parameters are not saved in their home space
- Also note that 0x28 bytes of stack space are still reserved for the parameters to MessageBoxA
System V x64 ABI
System V x64 ABI

• Used by Linux, BSD, Mac, others

• Totally different than MS x64 ABI
  • Also totally different than GCC's x86 Linux ABI

• Calling convention uses many registers:
  • 6 registers for integer arguments
  • 8 registers for float/double arguments

• Some registers considered volatile and can change across function calls, others must be saved by the callee
SysV ABI: Parameters

- First available register for the parameter type is used
- **6** registers for integer parameters
  - \texttt{RDI, RSI, RDX, RCX, R8, R9}
- **8** registers for float/double/vector parameters
  - \texttt{XMM0-XMM7}
- No overlap, so you could have **14** parameters stored in registers
- \texttt{struct} params can be split between registers
- Everything else is on the stack
- **RAX** holds number of vector registers (\texttt{XMMx})
SysV ABI: Parameter sequence

• Examples!

• `int func1(int a, float b, int c)`
  • `rax func1(rdi, xmm0, rsi)`

• `float func2(float a, int b, float c)`
  • `xmm0 func2(xmm0, rdi, xmm1)`

• `float func3(float a, int b, int c)`
  • `xmm0 func3(xmm0, rdi, rsi)`

• Notice anything interesting about `func1` and `func3`?
SysV ABI: Parameter example #1

```
printf("%i %i %f %i %f %i\n", 1, 2, 3.0, 4, 5.0, 6);
```

```assembly
.text:00000000004004F4 ; int __cdecl main(int argc, char **argv)
.text:00000000004004F4
.text:00000000004004F4
.text:00000000004004F4
.text:00000000004004F4
.text:00000000004004F4
.text:00000000004004F4
.text:00000000004004F4
.push rbp ; save rbp
mov rbp, rsp ; make rbp stack frame pointer
sub rsp, 10h ; clear space for saving 2 volatile registers
mov [rbp+var_4], edi ; save 1st parameter on the stack
mov [rbp+var_10], rsi ; save 2nd parameter on the stack
mov eax, offset format ; "%i %i %f %i %f %i\n"
movsd xmm1, cs:five_point_oh ; put 6th argument (5.0) in xmm1
movsd xmm0, cs:three_point_oh ; put 4th argument (3.0) in xmm0
mov r8d, 6 ; put 7th argument (6) in r8d
mov ecx, 4 ; put 5th argument (4) in ecx
mov edx, 2 ; put 3rd argument (2) in edx
mov esi, 1 ; put 2nd argument (1) in esi
mov rdi, rax ; put 1st argument (format string) in rdi
mov eax, 2 ; eax holds number of SSE registers used (2)
call _printf
leave
ret
.endp
```
SysV ABI: Parameter example #2

typedef struct {
    int a, b;
    double d;
} structParm;

structParm s;

int e, f, g, h, i, j, k;
long double ld;
double m, n;
__m256 y;

extern void func (int e, int f, structParm s,
    int g, int h, long double ld, double m,
    __m256 y, double n, int i, int j, int k);

func (e, f, s, g, h, ld, m, y, n, i, j, k);

(This example is from the SysV x64 ABI specs)
SysV ABI: The stack

- Nothing new here, except changes due to 64 bit platform
- Aligned on 16 byte boundaries
- GCC still uses RBP as a frame pointer by default
- No required home space like MS's ABI
  - Sometimes parameters are saved on the stack
  - It's in local variables and not behind the return address
- Functions can use stack space up to \texttt{RSP+256}
  - Beyond that is the \texttt{RED ZONE}
x64 Reversing Tools
Tools for x64 Reversing: IDA
Tools for x64 Reversing: Windbg
Tools for x64 Reversing: Visual DuxDebugger
Tools for x64 Reversing: edb
Other reversing tools for x64

- Dynamic instrumentation
  - PIN
  - DynamoRIO
- Virtual machines
  - BOCHS
  - QEMU
- That thing @msuiche is working on
- vdb/vtrace
How to get better at reversing

• Take a binary, any binary, but smaller is probably easier
• Reverse it all
  • Name every function, parameter, and variable
  • Comment almost every line of assembly
  • Do this without running it, unless you absolutely have to
• You'll be a pro in no time!
• Also, read the Rolf Rolles interview in HITB 005
x64 References!

- **x64 architecture**

- **MS x64 ABI**
  - x64 Software Conventions: http://msdn.microsoft.com/en-us/library/7kcdt6fy%28VS.80%29.aspx
  - X64 Deep Dive: http://www.codemachine.com/article_x64deepdive.html

- **SysV x64 ABI**
  - System V Application Binary Interface: http://www.x86-64.org/documentation/abi.pdf
Questions?

• **Contact info:**
  - E-mail: jlarimer@gmail.com
  - Twitter: @shydemeanor
  - Reddit: r0swell