

Intro to x64 Reversing



SummerCon 2011 - NYC

Jon Larimer

email: jarimer@gmail.com

twitter: [@shydemeanor](https://twitter.com/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 Opteron 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 - **RAX, RBX, ...**
 - 8 new GP registers (**R8-R15**)
 - 8 new 128 bit XMM registers (**XMM8-XMM15**)
- New 64 bit instructions: **cdqe, lodsq, stosq**, etc
- Ability to reference data relative to instruction pointer (**rip**)

* Limited by processor implementation, most only support 48 bits now...

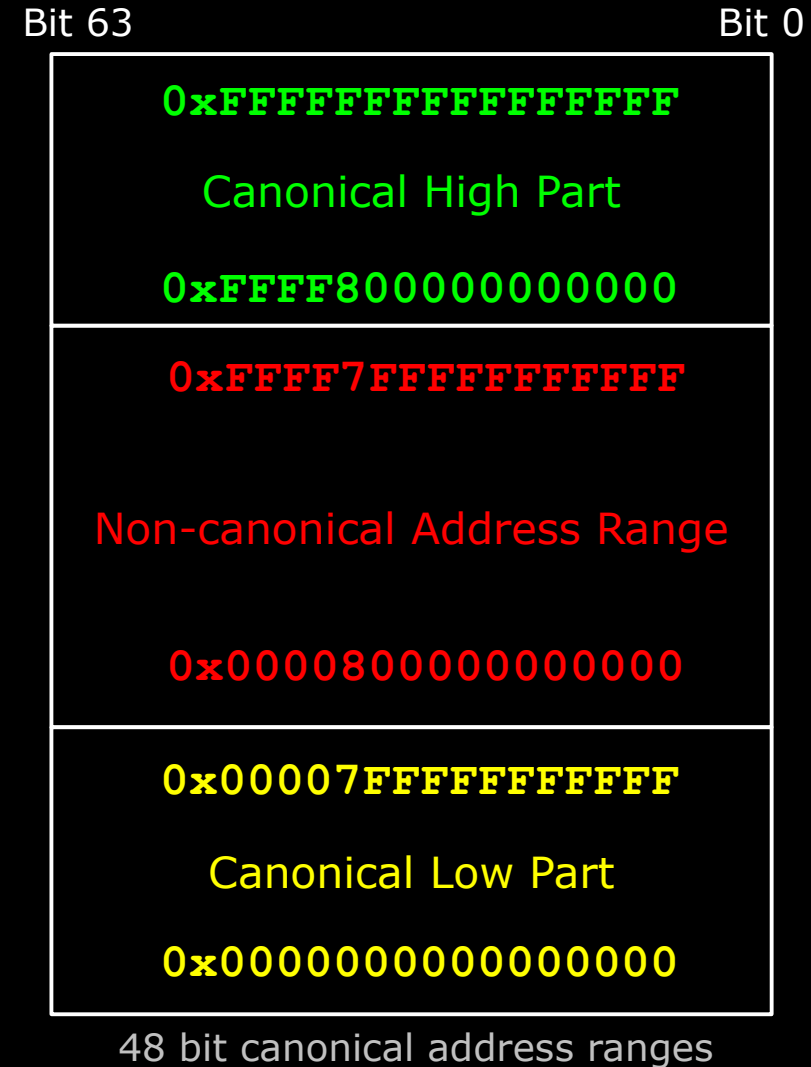
** Intel currently supports 40 bits of physical memory

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

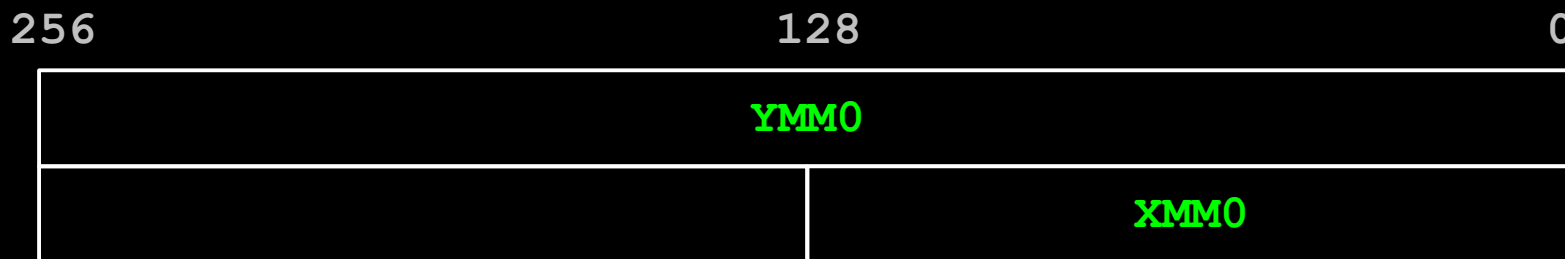


x64 registers

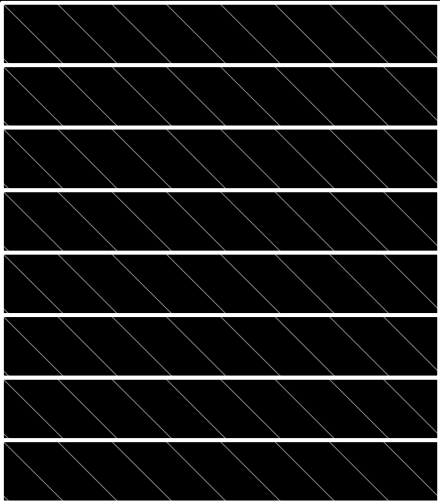

- 32 bit registers extended to 64 bits
 - `eax` → `rax`
 - `ebx` → `rbx`
 - `esp` → `rsp`
- 8 additional 64 bit registers
 - `r8`, `r9`, `r10`, ... `r15`
- 8 additional 128 bit XMM (SSE) registers
 - `xmm8`, `xmm9`, ... `xmm15`
 - Used for vector and floating point arithmetic


Intel/AMD AVX

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



x64 Registers

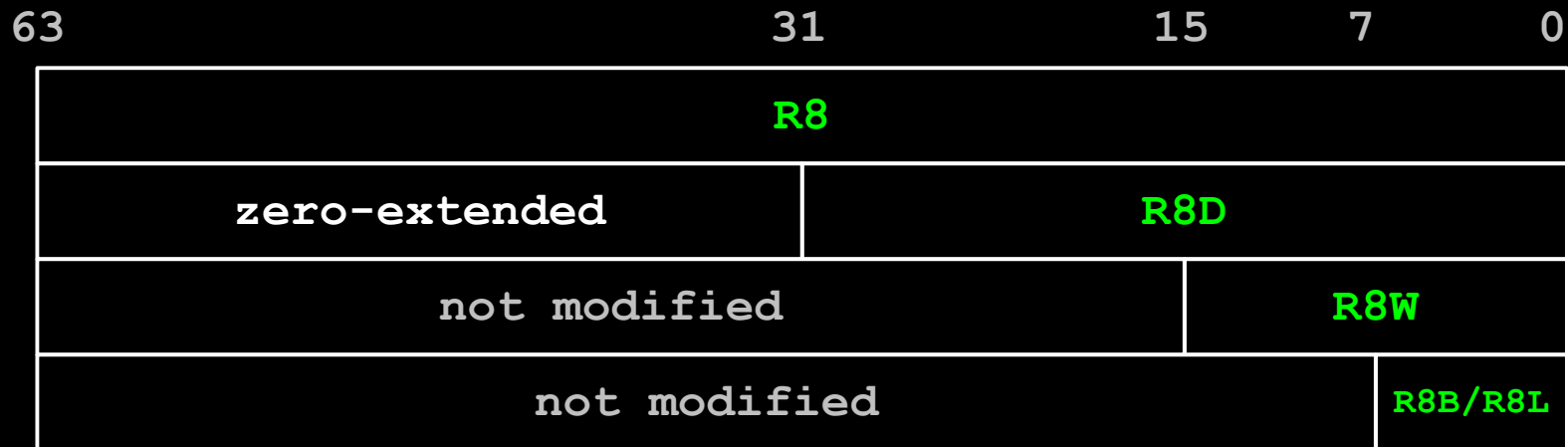
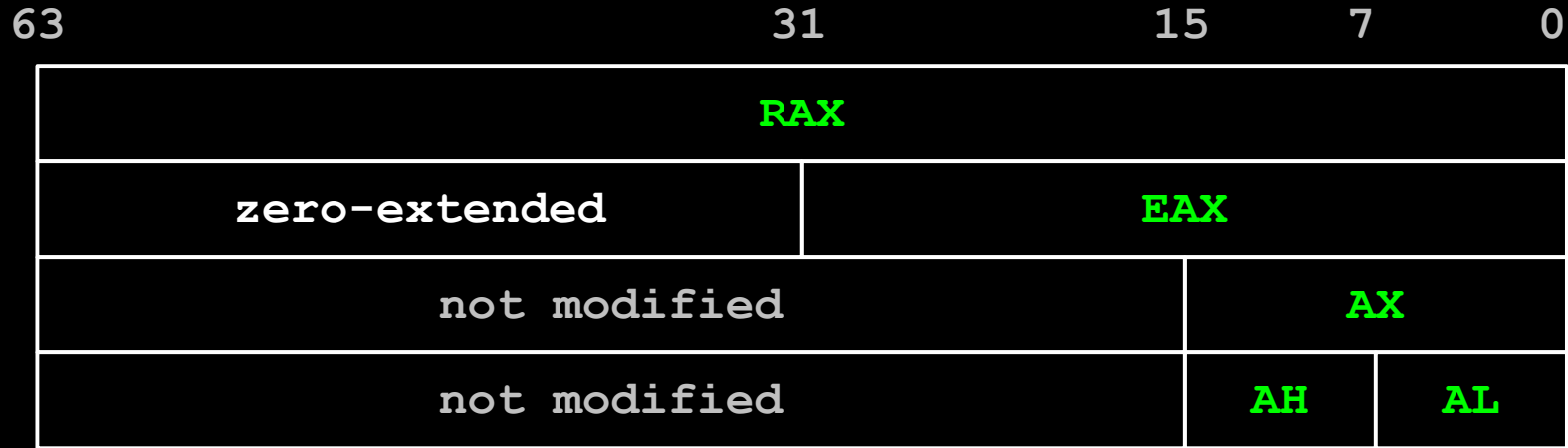
	63	31	0
RAX			EAX
RBX			EBX
RCX			ECX
RDX			EDX
RBP			EBP
RSI			ESI
RDI			EDI
RSP			ESP
R8			
R9			
R10			
R11			
R12			
R13			
R14			
R15			

	63	31	0
RIP			EIP
RFLAGS			EFLAGS

NOTE: Top half of RFLAGS is reserved, always 0

 = new in x64

Register operation in x64 mode



POP QUIZ #1!

- How many bits is **R9D**?
- 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**?
- 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, 11111111111111111111h
```

```
INC AL
```

```
INC AX
```

```
INC EAX
```

POP QUIZ #2!

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

```
MOV RAX, 11111111111111111111h
```

```
    RAX = 0x11111111111111111111
```

```
INC AL
```

```
INC AX
```

```
INC EAX
```

POP QUIZ #2!

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

```
MOV RAX, 11111111111111111111h
```

```
    RAX = 0x11111111111111111111
```

```
INC AL
```

```
    RAX = 0x11111111111111111112
```

```
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
```

```
    RAX = 0x1111111111111113
```

```
INC EAX
```


POP QUIZ #2!

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

```
MOV RAX, 1111111111111111h
```

```
    RAX = 0x1111111111111111
```

```
INC AL
```

```
    RAX = 0x1111111111111112
```

```
INC AX
```

```
    RAX = 0x1111111111111113
```

```
INC EAX
```

```
    RAX = 0x0000000011111114
```

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:

```
00000000140001000 ; int __stdcall WinMain(HINSTANCE hInstan
00000000140001000 WinMain proc near
00000000140001000
00000000140001000 arg_0= qword ptr 8
00000000140001000 arg_8= qword ptr 10h
00000000140001000 arg_10= qword ptr 18h
00000000140001000 arg_18= dword ptr 20h
00000000140001000
00000000140001000 mov [rsp+arg_18], r9d
00000000140001005 mov [rsp+arg_10], r8
0000000014000100A mov [rsp+arg_8], rdx
0000000014000100F mov [rsp+arg_0], rcx
00000000140001014 sub rsp, 28h
00000000140001018 xor r9d, r9d ; uType
0000000014000101B lea r8, Caption ; "Hi!"
00000000140001022 lea rdx, Text ; "Hello there!"
00000000140001029 xor ecx, ecx ; hWnd
0000000014000102B call cs:MessageBoxA
00000000140001031 xor eax, eax
00000000140001033 add rsp, 28h
00000000140001037 retn
00000000140001037 WinMain endp
00000000140001037
```

```
00000000140001000 ; int __stdcall WinMain(HINSTANCE hInstan
00000000140001000 WinMain proc near
00000000140001000
00000000140001000 arg_0= qword ptr 8
00000000140001000 arg_8= qword ptr 10h
00000000140001000 arg_10= qword ptr 18h
00000000140001000 arg_18= dword ptr 20h
00000000140001000
00000000140001000 mov [rsp+arg_18], r9d
00000000140001005 mov [rsp+arg_10], r8
0000000014000100A mov [rsp+arg_8], rdx
0000000014000100F mov [rsp+arg_0], rcx
00000000140001014 sub rsp, 28h
00000000140001018 xor r9d, r9d ; uType
0000000014000101B lea r8, [rip+7FDEh] ; "Hi!"
00000000140001022 lea rdx, [rip+7FDFh] ; "Hello there!"
00000000140001029 xor ecx, ecx ; hWnd
0000000014000102B call qword ptr [rip+5197h]
00000000140001031 xor eax, eax
00000000140001033 add rsp, 28h
00000000140001037 retn
00000000140001037 WinMain endp
00000000140001037
```

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/stdcall/fastcall)
- Calling convention modeled after fastcall
 - 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**, **RDY**, **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 **RDY** 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 XMM0 , XMM1 , XMM2 , XMM3 , [rsp+20h] , [rsp+28h] , . . . ) ;
```

Yellow = integers

Green = floats

Cyan = both

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]
```

Usually only see first four parameters stored in home space in debug code

MS x64 ABI: Params example

```
printf("%i %f %i %i %f\r\n", 1, 2.0, -4, 60, 5.5);
```

```
.text:00000000140001000 ; int __cdecl main(int argc, const char **argv, const char **envp)
.text:00000000140001000 main          proc near          ; CODE XREF: __tmainCRTStartup+136↓p
.text:00000000140001000                                     ; DATA XREF: .pdata:ExceptionDir↓o
.text:00000000140001000
.text:00000000140001000 var_18      = dword ptr -18h
.text:00000000140001000 var_10      = qword ptr -10h
.text:00000000140001000 arg_0       = dword ptr 8
.text:00000000140001000 arg_8       = qword ptr 10h
.text:00000000140001000
.text:00000000140001000          mov     [rsp+arg_8], rdx
.text:00000000140001005          mov     [rsp+arg_0], ecx
.text:00000000140001009          sub     rsp, 38h
.text:0000000014000100D          movsd  xmm0, cs:five_point_five
.text:00000000140001015          movsd  [rsp+38h+var_10], xmm0 ; 6th param is float 5.5, stored on the stack
.text:0000000014000101B          mov     [rsp+38h+var_18], 3Ch ; 5th param is the integer 60, stored on the stack
.text:00000000140001023          mov     r9d, 0FFFFFFFCh ; 4th param is the integer -4, stored in r9d
.text:00000000140001029          movsd  xmm2, cs:two_point_oh ; 3rd param is the float -2.0, stored both in xmm2 AND r8
.text:00000000140001031          movq   r8, xmm2
.text:00000000140001036          mov     edx, 1 ; 2nd param is the integer 1, stored in edx
.text:00000000140001038          lea   rcx, format_str ; "%i %f %i %i %f\r\n"
.text:00000000140001042          call  printf
.text:00000000140001047          xor   eax, eax
.text:00000000140001049          add   rsp, 38h
.text:0000000014000104D          retn
.text:0000000014000104D main          endp
```

MS x64 ABI: struct param example

```
typedef struct {
    __int64 a;
    __int64 b;
} abstruct;

int func(abstruct ab) {
    printf("%i %i\n", ab.a, ab.b);
    ab.a = 9;
    return 0;
}

int main(int argc, char **argv) {
    abstruct ab;

    ab.a = 3;
    ab.b = 5;

    func(ab);
    printf("%i %i\n", ab.a, ab.b);

    return 0;
}
```

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

```
mov     [rsp+78h+ab.a], 3 ; set local variable ab.a to 3
mov     [rsp+78h+ab.b], 5 ; set local variable ab.b to 5
lea     rax, [rsp+78h+new_ab] ; get offset to area on stack for a copy
                                ; of local variable ab
lea     rcx, [rsp+78h+ab] ; load ecx with address of local variable ab
mov     rdi, rax ; put stack address of ab copy in edi (destination)
mov     rsi, rcx ; put stack address of ab in esi (source)
mov     ecx, 10h ; set ecx (count) to 16, the size of the struct
rep movsb ; perform copy operation to move 16 bytes
                                ; from edi to esi
lea     rcx, [rsp+78h+new_ab] ; load ecx (first parameter) with address
                                ; of copy of ab struct
call    func ; call function with single parameter in ecx
```

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?

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., **r8** and **r9**)

POP QUIZ #3

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

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., **r8** and **r9**)

- **FALSE!**

MS x64 ABI: Volatile registers

- 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 **callees**
- For non-leaf functions, space for four arguments is always allocated ($4 * 8 \text{ bytes} = 32 = 0x20$)
 - `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

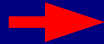
```
13FDB1080 ; int __stdcall WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nShowCmd)
13FDB1080 WinMain      proc near                                ; CODE XREF: __tmainCRTStartup+14F↓p
13FDB1080                                         ; DATA XREF: .pdata:000000013FDC000C↓o
13FDB1080
13FDB1080 lpText        = qword ptr -18h
13FDB1080 hInstance    = qword ptr 8
13FDB1080 hPrevInstance = qword ptr 10h
13FDB1080 lpCmdLine    = qword ptr 18h
13FDB1080 nShowCmd     = dword ptr 20h
13FDB1080
13FDB1080          mov     [rsp+nShowCmd], r9d
13FDB1085          mov     [rsp+lpCmdLine], r8 ; save caller's parameter registers even
13FDB1085                                         ; though it's not totally necessary here,
13FDB1085                                         ; but it can help with debugging...
13FDB108A          mov     [rsp+hPrevInstance], rdx
13FDB108F          mov     [rsp+hInstance], rcx
```

1008	return addr	(rsp)
1010	rcx	(rsp+08) hInstance
1018	rdx	(rsp+10) hPrevInstance
1020	r8	(rsp+18) lpCmdLine
1028	r9	(rsp+20) nShowCmd

```


13FDB1080
13FDB1080      mov     [rsp+nShowCmd], r9d
13FDB1085      mov     [rsp+lpCmdLine], r8 ; save caller's parameter registers even
13FDB1085                          ; though it's not totally necessary here,
13FDB1085                          ; but it can help with debugging...
13FDB108A      mov     [rsp+hPrevInstance], rdx
13FDB108F      mov     [rsp+hInstance], rcx
13FDB1094      sub     rsp, 38h ; save 0x38 bytes on stack for callee args,
13FDB1094                          ; local var, and alignment
13FDB1098      mov     r9d, 4 ; 4th argument in r9
13FDB109E      mov     r8d, 3 ; 3rd argument in r8
13FDB10A4      mov     edx, 2 ; second argument in edx
13FDB10A9      mov     ecx, 1 ; first argument in ecx
13FDB10AE      call   get_string
13FDB10B3      mov     [rsp+38h+lpText], rax ; return value from get_string is in rax,
13FDB10B3                          ; save it as local variable lpText

```



0FD0	ecx home	(rsp)	home space
0FD8	edx home	(rsp+08)	home space
0FE0	r8 home	(rsp+10)	home space
0FE8	r9 home	(rsp+18)	home space
0FF0	lpText	(rsp+20)	lpCmdLine
0FF8	???	(rsp+28)	???
1000	???	(rsp+30)	???
1008	return addr	(rsp+38)	(return to _tmainCRT..)
1010	rcx	(rsp+40)	hInstance
1018	rdx	(rsp+48)	hPrevInstance
1020	r8	(rsp+50)	lpCmdLine
1028	r9	(rsp+58)	nShowCmd

MS x64 ABI: Stack diagram

```
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                                     ; DATA XREF: .pdata:ExceptionDir↓o
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

0FD8	return addr	(rsp)	home space
0FE0	ecx	(rsp+08)	arg_a
0FD8	edx	(rsp+10)	arg_b
0FE0	r8	(rsp+18)	arg_c
0FE8	r9	(rsp+20)	arg_d
0FF0	lpText	(rsp+28)	lpCmdLine
0FF8	???	(rsp+30)	???
1000	???	(rsp+38)	???
1008	return addr	(rsp+40)	(return to _tmainCRT..)
1010	rcx	(rsp+48)	hInstance
1018	rdx	(rsp+50)	hPrevInstance
1020	r8	(rsp+58)	lpCmdLine
1028	r9	(rsp+60)	nShowCmd

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

```
.text:0000000140001000 ; int __stdcall WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine,  
.text:0000000140001000 WinMain      proc near          ; CODE XREF: __tmainCRTStartup+14F↓p  
.text:0000000140001000                                     ; DATA XREF: .pdata:ExceptionDir↓o  
.text:0000000140001000                                     sub     rsp, 28h  
.text:0000000140001004                                     lea    r8, Caption      ; "Hi!"  
.text:0000000140001008                                     lea    rdx, Text        ; "Hello there!"  
.text:0000000140001012                                     xor    r9d, r9d         ; uType  
.text:0000000140001015                                     xor    ecx, ecx         ; hWnd  
.text:0000000140001017                                     call   cs:MessageBoxA  
.text:000000014000101D                                     xor    eax, eax  
.text:000000014000101F                                     add    rsp, 28h  
.text:0000000140001023                                     retn  
.text:0000000140001023 WinMain      endp
```

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
 - **RDI, RSI, RDX, RCX, R8, R9**
- **8** registers for float/double/vector parameters
 - **XMM0-XMM7**
- No overlap, so you could have **14** parameters stored in registers
- **struct** params can be split between registers
- Everything else is on the stack
- **RAX** holds number of vector registers (**XMM_x**)

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);
```

```
.text:00000000004004F4 ; int __cdecl main(int argc, char **argv)
.text:00000000004004F4          public main
.text:00000000004004F4 main          proc near
.text:00000000004004F4
.text:00000000004004F4 var_10      = qword ptr -10h
.text:00000000004004F4 var_4      = dword ptr -4
.text:00000000004004F4
.text:00000000004004F4          push    rbp                ; save rbp
.text:00000000004004F5          mov     rbp, rsp           ; make rbp stack frame pointer
.text:00000000004004F8          sub     rsp, 10h           ; clear space for saving 2 volatile registers
.text:00000000004004FC          mov     [rbp+var_4], edi   ; save 1st parameter on the stack
.text:00000000004004FF          mov     [rbp+var_10], rsi  ; save 2nd parameter on the stack
.text:0000000000400503          mov     eax, offset format ; "%i %i %f %i %f %i\n"
.text:0000000000400508          movsd  xmm1, cs:five_point_oh ; put 6th argument (5.0) in xmm1
.text:0000000000400510          movsd  xmm0, cs:three_point_oh ; put 4th argument (3.0) in xmm0
.text:0000000000400518          mov     r8d, 6             ; put 7th argument (6) in r8d
.text:000000000040051E          mov     ecx, 4             ; put 5th argument (4) in ecx
.text:0000000000400523          mov     edx, 2             ; put 3rd argument (2) in edx
.text:0000000000400528          mov     esi, 1             ; put 2nd argument (1) in esi
.text:000000000040052D          mov     rdi, rax           ; put 1st argument (format string) in rdi
.text:0000000000400530          mov     eax, 2             ; eax holds number of SSE registers used (2)
.text:0000000000400535          call   _printf
.text:000000000040053A          leave
.text:000000000040053B          retn
.text:000000000040053B main          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);
```

RDI: e	XMM0: s.d	[RSP+0]: ld
RSI: f	XMM1: m	[RSP+16]: j
RDY: s.a,s.b	YMM2: y	[RSP+24]: k
RCX: g	XMM3: n	
R8: h		
R9: i		

(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 **RSP+256**
 - Beyond that is the **RED ZONE**

x64 Reversing Tools

Tools for x64 Reversing: IDA

The screenshot displays the IDA Pro interface for a 64-bit executable. The main window shows assembly code for the `WinMain` function. The instruction `sub rsp, 28h` is highlighted in red, indicating the current instruction pointer (RIP). The assembly code includes comments for `__stdcall` and `__tmainCRTStartup`. The right-hand pane shows the state of general registers, with `RAX` and `RIP` highlighted. The `RIP` register points to `WinMain+B`. The bottom-left pane shows the hex view of the current instruction, and the bottom-right pane shows the stack view, with `Stack[00000380]:00000000015F7D0` highlighted. The output window at the bottom shows system messages related to DLL loading.

```
.text:00000013F6F1000
.text:00000013F6F1000
.text:00000013F6F1000 ; int __stdcall WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nShowCmd)
.text:00000013F6F1000 WinMain proc near
.text:00000013F6F1000 ; CODE XREF: __tmainCRTStartup+14F↓
.text:00000013F6F1000 ; DATA XREF: .pdata:ExceptionDir↓
.text:00000013F6F1000 sub    rsp, 28h
.text:00000013F6F1004 lea   r8, Caption                ; "Hi!"
.text:00000013F6F1008 lea   rdx, Text                ; "Hello there!"
.text:00000013F6F1012 xor   r9d, r9d                  ; uType
.text:00000013F6F1015 xor   ecx, ecx                  ; hWnd
.text:00000013F6F1017 call  cs:MessageBoxA
.text:00000013F6F101D xor   eax, eax
.text:00000013F6F101F add   rsp, 28h
.text:00000013F6F1023 retn
.text:00000013F6F1023 WinMain endp
.text:00000013F6F1023
.text:00000013F6F1024 ; [00000199 BYTES: COLLAPSED FUNCTION __tmainCRTStartup. PRESS KEYPAD "+" TO EXPAND]
.text:00000013F6F11BD algn_13F6F11BD:                ; DATA XREF: .pdata:000000013F6FC00C↓
.text:00000013F6F11BD align 20h
.text:00000013F6F11C0 ; [00000012 BYTES: COLLAPSED FUNCTION start. PRESS KEYPAD "+" TO EXPAND]
.text:00000013F6F11D2 algn_13F6F11D2:                ; DATA XREF: .pdata:000000013F6FC018↓

0000040B 000000013F6F100B: WinMain+B
```

General registers:

RAX	0000000002931E6	debug014:0000000002931E6	OF 0
RBX	0000000000000000		DF 0
RCX	0000000013F6F000	test64.exe:hInstance	IF 1
RDY	0000000000000000		TF 0
RSI	0000000000000000		SF 0
RDI	0000000013F6F000	test64.exe:hInstance	ZF 0
RBP	0000000000000000		AF 0
RSP	0000000000015F7D0	Stack[00000380]:00000000015F7D0	PF 0
RIP	0000000013F6F100B	WinMain+B	CF 0
R8	0000000013F6F6240	.rdata:Caption	
R9	0000000000000000		
R10	0000000000000000		
R11	0000000000000022		
R12	0000000000000000		
R13	0000000000000000		
R14	0000000000000000		
R15	0000000000000000		
EFL	00000202		

Hex View-1:

```
00000013F6FDF0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
000007FEFCF30000 4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00 MZÉ.....
000007FEFCF30010 B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00 +.....@.....
000007FEFCF30020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
000007FEFCF30030 00 00 00 00 00 00 00 00 00 00 00 00 F0 00 00 00 .....
000007FEFCF30040 0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68 ..!..-!+.L-!Th
000007FEFCF30050 69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F is program canno
000007FEFCF30060 74 20 62 65 20 72 75 7E 20 69 6E 20 44 4F 53 20 t be run in DOS
000007FEFCF30070 6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00 mode...$.
000007FEFCF30080 A0 DA 7E 93 E4 BB 10 C0 E4 BB 10 C0 E4 BB 10 C0 ä+~ôS+.+S+.+S+.+
000007FEFCF30090 E4 BB 11 C0 7E BA 10 C0 ED C3 83 C0 E7 BB 10 C0 S+.+~|.f+â+tt+.+
000007FEFCF300A0 ED C3 93 C0 E1 BB 10 C0 ED C3 82 C0 E5 BB 10 C0 f+ô+ß+.f+é+st+.+
```

Stack view:

```
000000000015F7D0 0000000002931E5 debug014:0000000002931E5
000000000015F7D8 0000000000000000
000000000015F7E0 000000000015F808 Stack[00000380]:00000000015F808
000000000015F7E8 0000000000000000
000000000015F7F0 0000000013F6F000 test64.exe:hInstance
000000000015F7F8 0000000013F6F1178 __tmainCRTStartup+154
000000000015F800 0000000000000000
000000000015F808 0000000000000000
000000000015F810 0000000000000000
000000000015F818 0000000000000000
000000000015F820 0000000000000000
000000000015F828 0000000000000000
```

Output window:

```
7FEFD60000: loaded C:\Windows\system32\msvcrt.dll
7FEFD66000: loaded C:\Windows\system32\imm32.dll
7FEFD350000: loaded C:\Windows\system32\mactf.dll
```

Python

AU: idle Down Disk: 25GB

Tools for x64 Reversing: Windbg

The screenshot displays the WinDbg interface for debugging a 64-bit application. The main window shows the disassembly of the current instruction stream, with the following assembly code visible:

```
Offset: @$scopeip
00000000`76e8f950 4c8bd1      mov     r10,rcx
00000000`76e8f953 b827000000    mov     eax,27h
00000000`76e8f958 0f05        syscall
00000000`76e8f95a c3         ret
00000000`76e8f95b 0f1f440000    nop     dword ptr [rax+rax]
ntdll!ZwReplyWaitReceivePortEx:
00000000`76e8f960 4c8bd1      mov     r10,rcx
00000000`76e8f963 b828000000    mov     eax,28h
00000000`76e8f968 0f05        syscall
00000000`76e8f96a c3         ret
00000000`76e8f96b 0f1f440000    nop     dword ptr [rax+rax]
ntdll!ZwTerminateProcess:
00000000`76e8f970 4c8bd1      mov     r10,rcx
00000000`76e8f973 b829000000    mov     eax,29h
00000000`76e8f978 0f05        syscall
00000000`76e8f97a c3         ret
00000000`76e8f97b 0f1f440000    nop     dword ptr [rax+rax]
ntdll!NtSetEventBoostPriority:
00000000`76e8f980 4c8bd1      mov     r10,rcx
00000000`76e8f983 b82a000000    mov     eax,2Ah
00000000`76e8f988 0f05        syscall
00000000`76e8f98a c3         ret
00000000`76e8f98b 0f1f440000    nop     dword ptr [rax+rax]
ntdll!NtReadFileScatter:
00000000`76e8f990 4c8bd1      mov     r10,rcx
00000000`76e8f993 b82b000000    mov     eax,2Bh
00000000`76e8f998 0f05        syscall
00000000`76e8f99a c3         ret
00000000`76e8f99b 0f1f440000    nop     dword ptr [rax+rax]
ntdll!NtOpenThreadTokenEx:
```

The Registers window on the right shows the current state of the CPU registers:

Reg	Value
rax	0
rcx	0
rdx	7feff052a80
rbx	0
rsp	17f718
rbp	0
rsi	1ce3a30
rdi	0
r8	17f548
r9	0
r10	0
r11	286
r12	0
r13	1ce3a38
r14	1ce3a30
r15	0
rip	76e8f97a

The Memory window shows the virtual address @\$scopeip with the following hex dump:

```
Virtual: @$scopeip
00000000`76e8f97a c3 0f 1f 44 00 00 4c 8b d1 b8 2a 00 00 00 0f 05 c3 0f 1f 44 00 ...D...L...*...
00000000`76e8f98f 00 4c 8b d1 b8 2b 00 00 00 0f 05 c3 0f 1f 44 00 00 4c 8b d1 b8 ...L...+...D...L...
00000000`76e8f9a4 2c 00 00 00 0f 05 c3 0f 1f 44 00 00 4c 8b d1 b8 2d 00 00 0f ...D...D...L...-...
00000000`76e8f9b9 05 c3 0f 1f 44 00 00 4c 8b d1 b8 2e 00 00 00 0f 05 c3 0f 1f 44 ...D...L...L...-...
00000000`76e8f9ce 00 00 4c 8b d1 b8 2f 00 00 00 0f 05 c3 0f 1f 44 00 00 4c 8b d1 ...L.../...D...L...
00000000`76e8f9e3 b8 30 00 00 00 0f 05 c3 0f 1f 44 00 00 4c 8b d1 b8 31 00 00 00 ...0...D...D...L...1...
00000000`76e8f9f8 0f 05 c3 0f 1f 44 00 00 4c 8b d1 b8 32 00 00 00 0f 05 c3 0f 1f ...D...L...2...
00000000`76e8fa0d 44 00 00 4c 8b d1 b8 33 00 00 00 0f 05 c3 0f 1f 44 00 00 4c 8b ...D...L...3...D...
00000000`76e8fa22 d1 b8 34 00 00 00 0f 05 c3 0f 1f 44 00 00 4c 8b d1 b8 35 00 00 ...4...D...D...L...5...
00000000`76e8fa37 00 0f 05 c3 0f 1f 44 00 00 4c 8b d1 b8 36 00 00 00 0f 05 c3 0f ...D...D...L...6...
00000000`76e8fa4c 1f 44 00 00 4c 8b d1 b8 37 00 00 00 0f 05 c3 0f 1f 44 00 00 4c ...D...L...7...D...D...
00000000`76e8fa61 8b d1 b8 38 00 00 00 0f 05 c3 0f 1f 44 00 00 4c 8b d1 b8 39 00 ...8...D...D...T...
```

The Command window shows the following commands and their results:

```
0:000> int 3
0:001> g 0x76e8e915
ntdll!NtTerminateProcess+0xa:
00000000`76e8f97a c3      ret
0:000>
```

The status bar at the bottom indicates the current state: Ln 0, Col 0 | Sys 0:<Local> | Proc 000:b4c | Thrd 000:a2c | ASM | OVR | CAPS | NUM

Tools for x64 Reversing: Visual DuxDebugger

The screenshot displays the Visual DuxDebugger interface for a 64-bit application. The main window shows the assembly code for the current instruction set, with the instruction at address 0x0000000076e8fae0 highlighted in green. The instruction is a MOV R10, RCX. The registers window on the right shows the current state of the registers, with RAX and RIP highlighted in red. The memory window at the bottom shows the current address and the data being read or written.

Visual DuxDebugger - [PID: 2196 - TID: 0x6ac - test64.exe - ntdll.dll+NtContinue]

File Debug Window Help

Modules

- gdi32.dll
- imm32.dll
- kernel32.dll
- kernelbase.dll
- lpk.dll
- msctf.dll
- msvcrt.dll
- ntdll.dll
- test64.exe
- user32.dll
- usp10.dll

Address	MachineCode	Mnemo...	Operands
0x0000000076e8fab8	0f05	SYSCALL	
0x0000000076e8faba	c3	RET	
0x0000000076e8fabb	0f1f4400 00	NOP	DWORD [RAX+RAX+0x0]
0x0000000076e8fac0	4c 8bd1	MOV	R10, RCX
0x0000000076e8fac3	b8 3e000000	MOV	EAX, 0x3e
0x0000000076e8fac8	0f05	SYSCALL	
0x0000000076e8faca	c3	RET	
0x0000000076e8facb	0f1f4400 00	NOP	DWORD [RAX+RAX+0x0]
0x0000000076e8fad0	4c 8bd1	MOV	R10, RCX
0x0000000076e8fad3	b8 3f000000	MOV	EAX, 0x3f
0x0000000076e8fad8	0f05	SYSCALL	
0x0000000076e8fada	c3	RET	
0x0000000076e8fadb	0f1f4400 00	NOP	DWORD [RAX+RAX+0x0]
0x0000000076e8fae0	4c 8bd1	MOV	R10, RCX
0x0000000076e8fae3	b8 40000000	MOV	EAX, 0x40
0x0000000076e8fae8	0f05	SYSCALL	
0x0000000076e8faea	c3	RET	
0x0000000076e8faeb	0f1f4400 00	NOP	DWORD [RAX+RAX+0x0]
0x0000000076e8faf0	4c 8bd1	MOV	R10, RCX
0x0000000076e8faf3	b8 41000000	MOV	EAX, 0x41
0x0000000076e8faf8	0f05	SYSCALL	
0x0000000076e8fafa	c3	RET	
0x0000000076e8fafb	0f1f4400 00	NOP	DWORD [RAX+RAX+0x0]
0x0000000076e8fb00	4c 8bd1	MOV	R10, RCX
0x0000000076e8fb03	b8 42000000	MOV	EAX, 0x42
0x0000000076e8fb08	0f05	SYSCALL	
0x0000000076e8fb0a	c3	RET	

Basic-Registers

Reg	Value
RAX	0x0000000000000000
RCX	0x0000000000002af760
RDY	0x0000000000000001
RBX	0x0000000000002af760
RSP	0x0000000000002af708
RBP	0x0000000000000000
RSI	0x0000000000000000
RDI	0x0000000000000000
R8	0x0000000000002af698
R9	0x0000000000000000
R10	0x0000000000000000
R11	0x00000000000000202
R12	0x0000000000000000
R13	0x0000000000000000
R14	0x0000000000000000
R15	0x0000000000000000
RIP	0x0000000076e8fae0
CS	0x0000000000000033
DS	0x000000000000002b
ES	0x000000000000002b
FS	0x0000000000000053
GS	0x000000000000002b
SS	0x000000000000002b
CF	0
PF	0
AF	0
ZF	0
SF	0
TF	0
IF	1
DF	0
OF	0
IOPL	0
NT	0
RF	0
VM	0
AC	0
VIF	0
VIP	0
ID	0

Memory

Address: [] BYTE - 8 BIT

Modules Threads Controls Memory Callstack Events

Basic-Registers Advanced-Registers Properties

CAP NUM SCRL

Tools for x64 Reversing: edb

The screenshot displays the edb debugger interface with the following components:

- Assembly View:** Shows assembly instructions with their addresses and hex values. The current instruction is `call 0x0000000000004004f4` at address `00000000:004005b8`.
- Registers:** A list of registers including RAX, RBX, RCX, RDX, RBP, RSP, RSI, RDI, R8, R9, R10, and R11. RDI is highlighted with the value `00000000000000c8`.
- Data Dump:** Shows a memory dump at address `00007f48:79f21200` with hex and ASCII values.
- Stack:** Shows the stack frame with addresses and values, including a return address `return to 00007f4879989eff`.

```
00000000:0040056a 55          push rbp
00000000:0040056b 48 89 e5    mov rbp, rsp
00000000:0040056e 48 83 ec 30 sub rsp, 48
00000000:00400572 89 7d dc    mov dword ptr [rbp-36], edi
00000000:00400575 48 89 75 d0 mov qword ptr [rbp-48], rsi
00000000:00400579 48 c7 45 e0 2c 01 00 00 mov qword ptr [rbp-32], 0x012c
00000000:00400581 48 b8 00 00 00 00 00 00 mov rax, 0x4079000000000000
00000000:0040058b 48 89 45 e8 mov qword ptr [rbp-24], rax
00000000:0040058f 48 8b 45 e0 mov rax, qword ptr [rbp-32]
00000000:00400593 f2 0f 10 4d e8 movsd xmm1, xmmword ptr [rbp-24]
00000000:00400598 f2 0f 10 05 58 01 00 00 movsd xmm0, xmmword ptr [rip+344]
00000000:004005a0 41 b8 bc 02 00 00 mov r8d, 0x02bc
00000000:004005a6 b9 58 02 00 00 mov ecx, 0x0258
00000000:004005ab ba 32 00 00 00 mov edx, 50
00000000:004005b0 48 89 c6    mov rsi, rax
00000000:004005b3 bf c8 00 00 00 mov edi, 0xc8
00000000:004005b8 9a 37 ff ff ff call 0x0000000000004004f4
00000000:004005bd f2 0f 11 45 f8 movsd xmmword ptr [rbp-8], xmm0
00000000:004005c2 b8 e9 06 40 00 mov eax, 0x004006e9
00000000:004005c7 f2 0f 10 45 f8 movsd xmm0, xmmword ptr [rbp-8]
00000000:004005cc 48 89 c7    mov rdi, rax
00000000:004005cf b8 01 00 00 00 mov eax, 1
00000000:004005d4 e8 17 fe ff ff call 0x0000000000004003f0
00000000:004005d9 b8 00 00 00 00 mov eax, 0
00000000:004005de c9         leave
00000000:004005df c3         ret
00000000:004005e0 48 89 6c 24 d8 mov qword ptr [rsp-40], rbp
00000000:004005e5 4c 89 64 24 e0 mov qword ptr [rsp-32], r12
00000000:004005ea 48 8d 2d 33 08 20 00 lea rbp, [rip+0x00200833]
00000000:004005f1 4c 8d 25 2c 08 20 00 lea r12, [rip+0x0020082c]
00000000:004005f8 4c 89 6c 24 e8 mov qword ptr [rsp-24], r13
```

0x00000000004004f4 = 00000000004004f4

Data Dump: 00007f48:79f21200

00007f48:79f212e8	00 00 00 00 00 00 00 00 db 9c d1 79 48 7f 00 00Ù.NyH...
00007f48:79f212f8	50 0e 60 00 00 00 00 00 78 18 f2 79 48 7f 00 00	P.....x.ÿyH...
00007f48:79f21308	00 00 00 00 00 00 00 00 e8 12 f2 79 48 7f 00 00è.ÿyH...
00007f48:79f21318	00 00 00 00 00 00 00 00 58 18 f2 79 48 7f 00 00X.ÿyH...
00007f48:79f21328	00 00 00 00 00 00 00 00 50 0e 60 00 00 00 00 00P.....
00007f48:79f21338	f0 0e 60 00 00 00 00 00 e0 0e 60 00 00 00 00 00	ð.....à.....

Stack:

00007fff:5b3b7130	00007fff5b3b7248	Hr:[ÿ...
00007fff:5b3b7138	00000001004005e0	à.@.....
00007fff:5b3b7140	000000000000012c	,.....
00007fff:5b3b7148	4079000000000000y@
00007fff:5b3b7150	00007fff5b3b7240	@r:[ÿ...
00007fff:5b3b7158	0000000000000000
00007fff:5b3b7160	0000000000000000
00007fff:5b3b7168	00007f4879989eff	ÿ..yH... return to 00007f4879989eff
00007fff:5b3b7170	0000000000000000
00007fff:5b3b7178	00007fff5b3b7248	Hr:[ÿ...

paused

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
 - Intel Architecture Software Development Manuals:
<http://www.intel.com/products/processor/manuals/>
 - AMD Architecture Programmer's Manuals:
<http://developer.amd.com/documentation/guides/pages/default.aspx>
- 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: jarimer@gmail.com
 - Twitter: [@shydemeanor](https://twitter.com/shydemeanor)
 - Reddit: [r0swell](https://www.reddit.com/user/r0swell)