Introduction to Reverse Engineering

Zack Orndorff

Why might I want to learn reversing?

- It’s fun
- You want to know how a program does something
    - I’ve (tried to) reverse the Binary Ninja UI module to know what APIs it uses
- *You do not reverse code purely for the heck of it.*

What can I do with RE?

- Vulnerability research
- Malware analysis
- Application interoperability
- Game hacking
- DRM cracking
- CTFs!

You should probably check with your lawyer before doing some of these!

What is RE?

- Looking at a finished system to learn about how it was engineered
- Or: to quote Chris, “RE is the art of NOT reading assembly”
- More than just binary reversing, but we will focus on binary reversing.

Quick intro to assembly

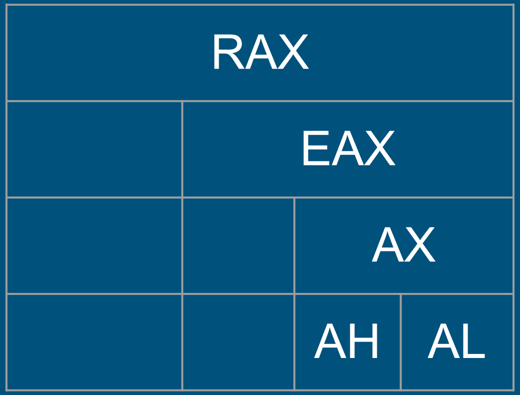
Assembly isn't hard because of the terse abbreviations

It's hard because:

1. Registers are harder to follow than variables
2. You have to know how the compiler used memory
3. You have to follow algorithms at a lower level than normal.
4. Processors each have their own quirks

You don't have to be able to write assembly! Just read small bits of it

Registers



Small 64-bit sized chunks of data

Used to hold data the processor is actively using

General purpose vs Special purpose

Names like rdi, rsi, r8, r9, rcx, rdx, rax, rbx

Special registers: XMM, YMM, ZMM; RIP, RFLAGS (ignore these mostly)

mov Instruction / Addressing Modes

Simple instruction: mov rax, rbx

mov rax, rbx	rax = rbx
mov rax, [rbx]	rax = *rbx
mov rax, [rbx+rcx*8]	rax = rbx[rcx] (where rbx is array of ints)
mov rax, 0xDEADBEEF	rax = 0xDEADBEEF

lea: Load Effective Address - an exception

The lea instruction is the ONLY case in x86_64 assembly where [brackets] does not dereference memory

Calculates what address WOULD have been accessed and stores that

```
mov rbx, 4
mov rcx, 2
lea rax, [rcx+8*rbx-3]
```

What will be in rax after this code runs?

Arithmetic

add, sub, mul, imul, div, idiv

Generally OP, destination, other_source, but varies

Look it up. I have no idea what imul does off the top of my head

Simple case: add rax, 1

xor

Fun little operation. Can be used for lame encryption, you'll see this in CTFs and malware

xor rax, rax sets rax to 0. Used often by compilers

Here's why:

```
0:   48 31 c0                    xor    rax,rax
3:   48 c7 c0 00 00 00 00        mov    rax,0x0
```

Conditionals

Usually consist of a cmp then a jCC instruction, where CC is a *condition code*

Options include e (equal), ne (not equal), le (less than or equal), etc

Signedness sometimes matters, but less than you'd think.

```
mov rax, <some_num>
cmp rax, 10
jle small

call print_large_msg
jmp end

small:
call print_small_msg

end:
```

Flags

What's happening here?

cmp instruction sets flags

Most arithmetic also sets flags

Zero, Signed, Overflow, Carry, etc

Conditional jumps use flags

I ignore all but the Zero flag, the rest make sense

```
sub rax, 10
jle small

call print_large_msg
jmp end

small:
call print_small_msg

end:
```

Intel String Instructions

Nobody knows what “repne movsq” means off the top of their head

Instructions that start with “repne” tend to mean “keep doing the thing to the right until some condition is met”

Make doing certain operations on strings faster

Just look them up… no shame in that

Quick exercise

```
mov rdi, <some_number>
lea rdi, [rdi+rdi]
add rdi, 1
cmp rdi, 50
jge err

cmp rdi, 30
jle err
```

```
sub rdi, 30
cmp rdi, 2
jge err
call print_success
jmp done

err:
call print_error
done:
call exit
```

The Stack

We have a stack

rsp points to the top item on the stack

Confusingly, the stack grows down the address space

But that’s ok!

push and pop let you, well, push and pop things

“bottom of stack”	0xFFF0	“bottom \0”
	0xFFE8	0xDEADBEEF
	0xFFE0	0xF00DBEEF
	0xFFD8	“hithere\0”
	0xFFD0	0x1337
rsp -> “top of stack”	0xFFC8	4

push, pop instructions

push decrements rsp by 8 and places its operand at [rsp]

(Roughly) equivalent to:

sub rsp, 8
mov [rsp], <pushed_value>

pop does the inverse:
mov <some_reg>, [rsp]
add rsp, 8

call, ret instructions

call pushes the address of the next instruction and jumps where you tell it

So call some_func is roughly equivalent to:
push <next_instruction_addr>
jmp some_func

ret pops an address off the stack and jumps there

Equivalent to
pop rip

The Stack Frame

```c
int main() {
    func1();
}
void func1() {
    func2();
}
void func2() {
   int my_var = 0xDEADBEEF
}
```

```
main:
        call func1
func1:
        push rbp
        mov rbp, rsp
        call func2
        pop rbp
        ret
```

```
func2:
        push rbp
        mov rbp, rsp
        sub rsp, 8
        mov [rbp-8], 0xDEADBEEF
        mov rsp, rbp
        pop rbp
        ret
```

The Stack Frame

Suppose this C code:

```
int main() {
    func1();
}
void func1() {
    func2();
}
void func2() {
   int my_var = 0xDEADBEEF
}
```

```
main:
        call func1
func1:
        push rbp
        mov rbp, rsp
        call func2
        pop rbp
        ret
```

"bottom of stack"	0xFFF0	ret addr in main
	0xFFE8	rbp from main, saved by func1
rsp -> "top of stack"	0xFFE0	ret addr in func1

The Stack Frame

Suppose this C code:

```
int main() {
    func1();
}
void func1() {
    func2();
}
void func2() {
   int my_var = 0xDEADBEEF
}
```

```
func2:
        push rbp
        mov rbp, rsp
        sub rsp, 8
        mov [rbp-8],
0xDEADBEEF
        mov rsp, rbp
        pop rbp
        ret
```

"bottom of stack"	0xFFF0	ret addr in main
	0xFFE8	rbp from main, saved by func1
	0xFFE0	ret addr in func1
rbp ->	0xFFD8	rbp from func1, saved by func2
Local variable rsp -> "top of stack"	0xFFD0	0xDEADBEEF local variable

Passing parameters

What if we want to call a function, say

```
int say_hi(char *greeting, char *name);
```

How do we pass those parameters?

Depends on OS

We're focusing on Linux x86_64, known as the System V AMD64 calling convention

Passing parameters

int say_hi(char *greeting, char *name);

Registers: rdi, rsi, rdx, rcx, r8, r9

```
greeting:
    db “Hey, “, 0
name:
    db “Ben”, 0

mov rdi, greeting
mov rsi, name
call say_hi
// return value in rax
```

Other calling conventions

Different on Windows, they don’t use rdi/rsi

Different for 32-bit, they push all parameters on the stack

Other arches tend to use registers but they’re named differently

Calling conventions are actually really complicated and have lots more details, but in general you won’t need to care unless you’re writing a compiler

Tools!

- Disassemblers
  - IDA, Binary Ninja, Ghidra, radare2, Hopper
  - objdump works in a pinch
- Debuggers
  - Windows: WinDbg, x64dbg, OllyDbg
  - Linux: gdb, gdb, or gdb.
- Decompilers
  - Hex-Rays decompiler ($$$$), Ghidra, Retdec, Hopper, Snowman (pls no)
- Magic
  - Symbolic execution (angr, manticore), fuzzing (afl, libfuzzer)

Intro to IDA

Industry standard tool

The Pro version costs ~$1000 for a personal license, higher for companies

Disassembles most architectures known to man

There is a free version for noncommercial use that will disassemble x86_64, which is a gift from Hex-Rays to the RE community.

IDA Cheat Sheet (for the lab)

When loading files, just use the defaults

Double click things to navigate into them

N to rename things, Y to change type

X to look at XREFs

Right click on numbers to change how they are displayed

View -> Open subviews -> Strings or Shift-F10 for the Strings view