Week 4: Reverse Engineering Part 1

Mina Zhou & Roy Xu
OSIRIS Lab Hacknight

OSIRIS

What is Reverse Engineering?

- Understanding how things work

In CTF:

- taking a compiled (machine code, bytecode) program and converting it back into a more human readable format.

Overview - Intro to x86 and x86-64

- ASM Basics
  - Registers
  - Instructions
  - Conditionals & Branching
  - Function calls & calling conventions
  - Syscalls
- Stack
- Workshop

ASM Basics - Syntax

- The examples show on the slides are in Intel syntax
- AT&T syntax differs in appearance
  - Direction of operands is opposite
  - Registers are prefixed with ‘%’ and constants with ‘$’
  - Memory operands are enclosed in parenthesis instead of brackets
  - Form for complex operations is more complicated
    - [ebx + 20h] ⇒ 0x20(%ebx)
  - Some instructions have suffixes (movb, movw, movl)

ASM Basics - General Registers

- Data Registers: RAX, RBX, RCX, RDX

Register	Accumulator			Counter			Data			Base		
64-bit	RAX			RCX			RDX			RBX		
32-bit		EAX			ECX			EDX			EBX	
16-bit		AX			CX			DX			BX	
8-bit		AH	AL		CH	CL		DH	DL		BH	BL

ASM Basics - General Registers

Pointer Registers: **EIP**, ESP, EBP

Stack Pointer		Stack Base Pointer	
RSP		RBP	
	ESP		EBP
	SP		BP
	SPL		BPL

ASM Basics - Registers

64-bit register	Lower 32 bits	Lower 16 bits	Lower 8 bits
rax	eax	ax	al
rbx	ebx	bx	bl
rcx	ecx	cx	cl
rdx	edx	dx	dl
rsi	esi	si	sil
rdi	edi	di	dil
rbp	ebp	bp	bpl
rsp	esp	sp	spl
r8	r8d	r8w	r8b
r9	r9d	r9w	r9b
r10	r10d	r10w	r10b
r11	r11d	r11w	r11b
r12	r12d	r12w	r12b
r13	r13d	r13w	r13b
r14	r14d	r14w	r14b
r15	r15d	r15w	r15b

ASM Basics - Segment Registers

- **.code (CS)** - the program code
- **.data (DS)** - global data
- **.stack (SS)** - local variables, function arguments, etc.

- Old method of accessing memory regions
- Modern operating systems use paging

ASM Basics - Paging

- Splits RAM into equal sized chunks (pages)
- Simplified explanation:
  - OS provides a page table that maps virtual memory to physical memory
  - Virtual memory is what program uses
  - Physical memory is actual location in RAM
  - OS handles switching between page tables

ASM Basics - Instructions

- Loaded in memory (in its own page or pages)
- Instruction Register (EIP/RIP) points to memory location of instruction to be executed next

Instruction Cycle

- fetch-decode-execute cycle
  - CPU fetches from memory address in instruction register
  - The encoded instruction is decoded
  - Required data is fetched from main memory and placed in registers
  - CPU performs the actions required by the instruction

ARM Basics - Data Types (Integer)

Data type	Keyword	Bytes	C
byte	BYTE (b)	1	char
word	WORD (w)	2	short
double word	DWORD (l)	4	int
quad word	QWORD (q)	8	long int

ASM Basics - Data movement

mov

- mov <reg1>, <reg2>
- mov <reg1>,<reg2>
- mov <reg1>,<mem>
- mov <mem>,<reg2>
- mov <reg1>,<const>
- mov <mem>,<const>

copies data from first operand to second operand

ASM Basics - Data Movement

push

- push <reg>
- push <mem>
- push <const>

places its operand onto the top of the stack
decrements ESP by 4 then places operand into the address in ESP
operand has to be 32 bit or 64 bit depending on architecture

ASM Basics - Data Movement

pop

- pop <reg>
- pop <mem>

removes top of the stack into operand
increments ESP by 4
operand has to be 32 bit or 64 bit depending on architecture

ASM Basics - Data Movement

lea

- lea <reg>, <mem>

places the address specified by second operand into first operand

lea edi, [ebx + 4*esi] - the value ebx + 4 * esi is calculated and placed in edi
mov edi, [ebx + 4*esi] - the value in the address, ebx + 4 * esi, is placed in edi

ASM Basics - Arithmetic Instructions

- add dest, src
- sub dest, src
- div divisor → eax as dividend, edx as remainder
- mul value || mul dest, value, value || mul dest, value

ASM Basics - Logical Instructions

and, or, xor

- and <reg>, <reg>
- and <reg>, <mem>
- and <mem>, <reg>
- and <reg>, <const>
- and <mem>, <const>

Perform the operation on their operands

xor edx, edx - set contents of edx to zero

ASM Basics - Control Flow

- cmp
- jmp, je, jle, jnz, jz, jbe, jge...

Difference between JBE and JLE?

JBE for unsigned, JLE for signed

ASM Basics - Conditionals & Branching

- **Single-branch conditionals**

```
mov     eax, [var]
test    eax, eax
jnz     After condition
call    function
After condition...
```

```
if(var == 0){

        function();

}
// After condition...
```

ASM Basics - Conditionals & Branching

- **Two-way conditionals**

```
if (var == 7){
        functionOne();
}
else{
        functionTwo();
}

//After condition….
```

```
cmp         eax, 7
jz          else_statement
call        functionOne
jmp         After condition

else_statement:
call        functionTwo

After condition...
```

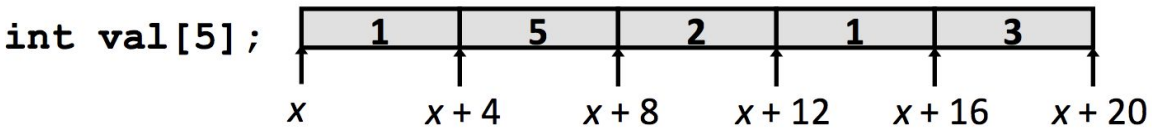
ASM Basics - Loops

```
// Assume the array size is alright

c = 0;
while (c < 1000) {
        array[c] = c;
        c++;
}
```

```
mov ecx, DWORD PTR [array]
xor eax, eax
LoopStart:
mov DWORD PTR [ecx+eax*4], eax
add eax, 1
cmp eax, 1000
jl LoopStart
```

ASM Basics - Array Accessing



- Register edx contains the starting address of the array
- Register eax contains array index
- Desired digit at 4*eax + edx
- Use memory reference

```
mov     eax, [edx + eax * 4]
```

ASM Basics - Function Calls

function(2, 10, 5)

```
push    5
push    10
push    2
call    function
```

- call <function>

- Differences between CALL and JMP?

JMP cannot store the current EIP on stack, it only loads

So CALL is basically push, jmp

ASM Basics - Function Calls

- ret
- ret num

- When callee is finished, caller's EBP is popped back into EBP
- RET remove the stack frame of the callee, incrementing ESP, popped the old EIP into EIP, and continue
- (depends on calling conventions)

So RET is basically pop, jmp

ASM Basics - Calling Conventions

- SystemV AMD64 ABI arguments passed in the registers
  - The first 6 integer arguments are passed in registers RDI, RSI, RDX, RCX, R8, R9
  - Arguments after that are pushed into stack

```
func(int a, int b, int c, int d, int e);
// a in RDI, b in RSI, c in RDX, d in RCX, e in R8
```

ASM Basics - Calling Conventions

- x32 arguments passed in the stack

ASM Basics - Syscalls

- Linux x86 and x86_64 have different Syscall reference
- int 0x80
- syscall && sysenter

Syscall Num	Param 1	Param 2	Param 3	Param 4	Param 5	Param 6
EAX	EBX	ECX	EDX	ESI	EDI	EDP
RAX	RDI	RSI	RDX	R10	R8	R9

RETURN VALUE

ASM Basics - Syscalls

- 32-bit: sys_execve #11 0x0b
- 64-bit: execve #59 0x3b

```
xor     eax, eax
push    eax
push    0x68732f2f
push    0x6e69622f
mov     ebx, esp
mov     ecx, eax
mov     al, 0xb
int     0x80
```

Syscall Num	Param 1	Param 2
EAX	EBX	ECX
RAX	RDI	RSI

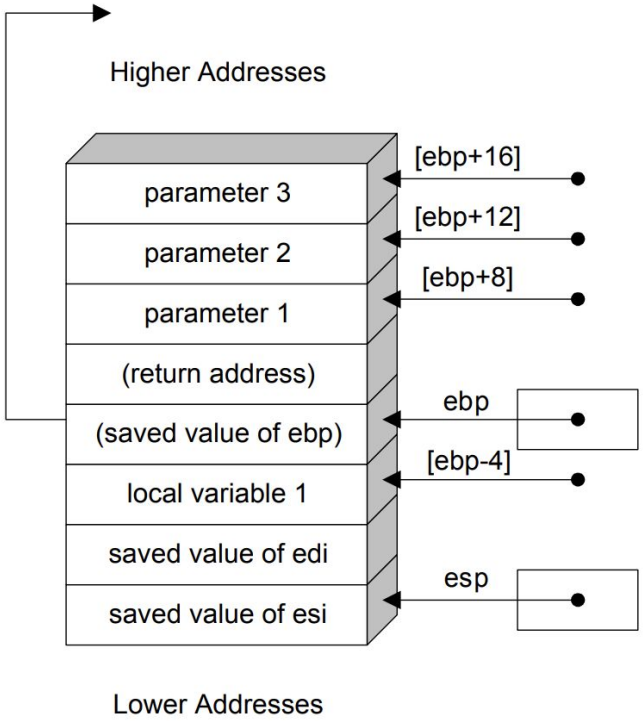
We will get into this more in shellcoding

Stack - grows down

- From the highest to the lowest
- RSP points to the top of the stack
- Add something → RSP **decrements**

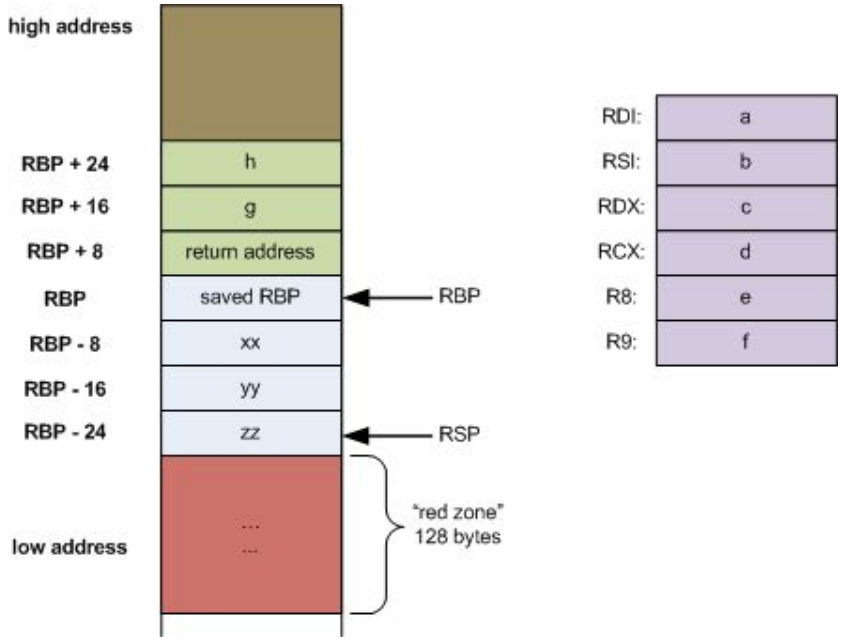
```
push  rax:
        sub   rsp, 8
        mov  [rsp], rax

pop   rax:
        mov  rax, [rsp]
        add   rsp, 8
```



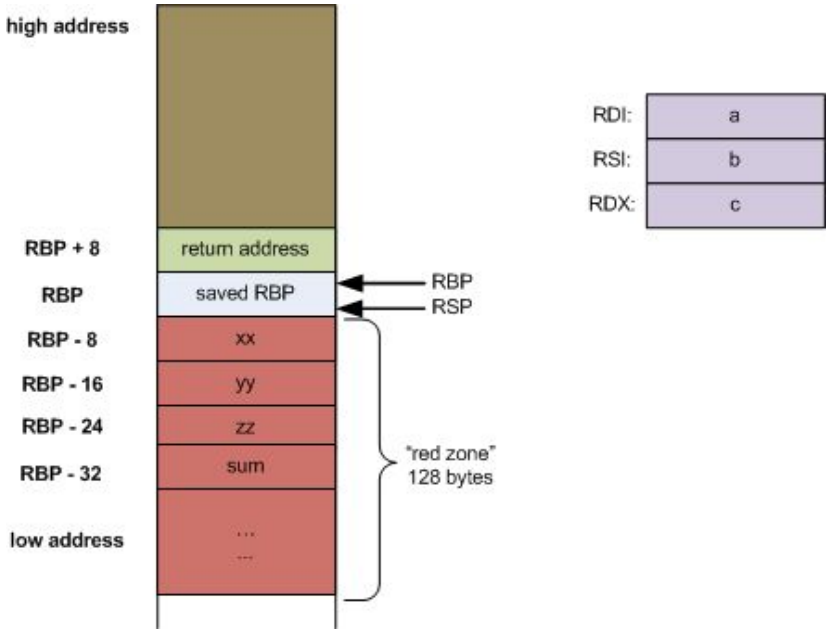
X86-64 Stack Frame Example

```
long myfunc(long a, long b, long c, long d, long e,
long f, long g, long h){
        long xx = a * b * c * d * e * f * g * h;
        long yy = a + b + c + d + e + f + g + h;
        long zz = utilfunc(xx, yy, xx % yy);
        return zz + 20;
}
```



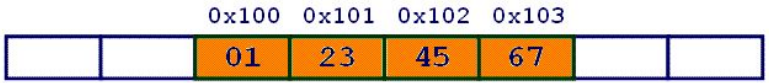
X86-64 Stack Frame Example 2

```
long utilfunc(long a, long b, long c){
        long xx = a + 2;
        long yy = b + 3;
        long zz = c + 4;
        long sum = xx + yy + zz;
        return xx * yy * zz + sum;
}
```



ASM Basics - Endianness

- The order that bytes are arranged when stored
- 0x01234567



Big Endian

Little Endian

Workshop

- https://github.com/osirislab/Hack-Night/tree/master/Rev

References

- http://www.cs.cmu.edu/afs/cs/academic/class/15213-s14/www/lectures/08-machine-data.pdf
- https://imada.sdu.dk/Employees/kslarsen-bak/Courses/dm18-2007-spring/Litteratur/IntelnATT.htm
- https://sensepost.com/blogstatic/2014/01/SensePost_crash_course_in_x86_assembly-.pdf
- https://en.wikibooks.org/wiki/X86_Assembly/Interfacing_with_Linux
- https://syscalls.kernelgrok.com/
- https://filippo.io/linux-syscall-table/
- https://aaronbloomfield.github.io/pdr/book/x86-32bit-ccc-chapter.pdf
- https://eli.thegreenplace.net/2011/09/06/stack-frame-layout-on-x86-64/