Binary Reverse Engineering And Analysis

Course 2: Assembly

Caragea Radu

February 23, 2021

Recap

- Black-box analysis: figure out only from external interactions

Recap

- Black-box analysis: figure out only from external interactions
- White-box analysis: exhaustively cover the binary
- Gray-box analysis: middle ground
- Today we start learning concepts for white-box analysis

Executables

- Most executables (ELF/SO, PE/DLL, WASM) have structure
- Based on generic computer science concepts
- Multiple sections/segments:
  - Text section (text == readable by the CPU)
  - Read-only Data section/Read-Write Data Section
  - Relocations/Compiler Stubs
- However...

Executables

CPU functionality

- The CPU consumes code and produces effects
- The consumed code is in binary form (machine code)

```
000006d0  85 c0 74 0c 5d ff e0 66  0f 1f 84 00 00 00 00 00  |..t.]..f........|
000006e0  5d c3 0f 1f 40 00 66 2e  0f 1f 84 00 00 00 00 00  |]...@.f.........|
000006f0  80 3d 49 09 20 00 00 75  27 48 83 3d f7 08 20 00  |.=I. ..u'H.=.. .|
00000700  00 55 48 89 e5 74 0c 48  8b 3d 2a 09 20 00 e8 0d  |.UH..t.H.=*. ...|
00000710  ff ff ff e8 48 ff ff ff  5d c6 05 20 09 20 00 01  |....H...].. . ..|
00000720  f3 c3 0f 1f 40 00 66 2e  0f 1f 84 00 00 00 00 00  |....@.f.........|
00000730  48 8d 3d b1 06 20 00 48  83 3f 00 75 0b e9 5e ff  |H.=.. .H.?.u..^.|
00000740  ff ff 66 0f 1f 44 00 00  48 8b 05 99 08 20 00 48  |..f..D..H.... .H|
00000750  85 c0 74 e9 55 48 89 e5  ff d0 5d e9 40 ff ff ff  |..t.UH....].@...|
00000760  55 48 89 e5 48 83 ec 50  c6 45 e0 41 c6 45 e1 47  |UH..H..P.E.A.E.G|
00000770  c6 45 e2 42 c6 45 e3 57  c6 45 e4 40 c6 45 e5 6d  |.E.B.E.W.E.@.E.m|
00000780  c6 45 e6 41 c6 45 e7 57  c6 45 e8 51 c6 45 e9 40  |.E.A.E.W.E.Q.E.@|
00000790  c6 45 ea 57 c6 45 eb 46  c6 45 ec 6d c6 45 ed 42  |.E.W.E.F.E.m.E.B|
000007a0  c6 45 ee 53 c6 45 ef 41  c6 45 f0 41 c6 45 f1 45  |.E.S.E.A.E.A.E.E|
000007b0  c6 45 f2 5d c6 45 f3 40  c6 45 f4 56 c6 45 f5 03  |.E.].E.@.E.V.E..|
000007c0  c6 45 f6 00 c6 45 f7 01  c6 45 f8 01 c6 45 f9 00  |.E...E...E...E..|
000007d0  c6 45 fa 07 c6 45 fb 32  48 8d 45 b0 48 89 c6 48  |.E...E.2H.E.H..H|
000007e0  8d 3d ee 00 00 00 b8 00  00 00 00 e8 20 fe ff ff  |.=.......... ...|
000007f0  c7 45 fc 00 00 00 00 eb  1c 8b 45 fc 48 98 0f b6  |.E........E.H...|
00000800  44 05 e0 83 f0 32 89 c2  8b 45 fc 48 98 88 54 05  |D....2...E.H..T.|
00000810  e0 83 45 fc 01 8b 45 fc  83 f8 1b 76 dc 48 8d 55  |..E...E....v.H.U|
00000820  e0 48 8d 45 b0 48 89 d6  48 89 c7 e8 d0 fd ff ff  |.H.E.H..H.......|
00000830  85 c0 75 0e 48 8d 3d 9e  00 00 00 e8 b0 fd ff ff  |..u.H.=.........|
00000840  eb 0c 48 8d 3d 99 00 00  00 e8 a2 fd ff ff b8 00  |..H.=...........|
00000850  00 00 00 c9 c3 66 2e 0f  1f 84 00 00 00 00 00 90  |.....f..........|
00000860  41 57 49 89 d7 41 56 49  89 f6 41 55 41 89 fd 41  |AWI..AVI..AUA..A|
00000870  54 4c 8d 25 60 05 20 00  55 48 8d 2d 60 05 20 00  |TL.%`. .UH.-`. .|
00000880  53 4c 29 e5 48 83 ec 08  e8 3b fd ff ff 48 c1 fd  |SL).H....;...H..|
00000890  03 74 1b 31 db 0f 1f 00  4c 89 fa 4c 89 f6 44 89  |.t.1....L..L..D.|
000008a0  ef 41 ff 14 dc 48 83 c3  01 48 39 dd 75 ea 48 83  |.A...H...H9.u.H.|
000008b0  c4 08 5b 5d 41 5c 41 5d  41 5e 41 5f c3 0f 1f 00  |..[]A\A]A^A_....|
000008c0  c3 00 00 00 48 83 ec 08  48 83 c4 08 c3 00 00 00  |....H...H.......|
000008d0  01 00 02 00 25 34 30 73  00 43 6f 72 72 65 63 74  |....%40s.Correct|
000008e0  21 00 57 72 6f 6e 67 00  01 1b 03 3b 3c 00 00 00  |!.Wrong....;<...|
000008f0  06 00 00 00 f8 fc ff ff  88 00 00 00 38 fd ff ff  |............8...|
00000900  b0 00 00 00 48 fd ff ff  58 00 00 00 78 fe ff ff  |....H...X...x...|
00000910  c8 00 00 00 78 ff ff ff  e8 00 00 00 d8 ff ff ff  |....x...........|
00000920  30 01 00 00 00 00 00 00  14 00 00 00 00 00 00 00  |0...............|
00000930  01 7a 52 00 01 78 10 01  1b 0c 07 08 90 01 07 10  |.zR..x..........|
```

CPU functionality

- Machine code can be unequivocally translated to readable assembly code
- In assembly form, it can be "interpreted" by the human brain
- For efficiency, it is organized into blocks, subroutines, functions, libraries, etc

```
7a0:   c6 45 ee 53              mov    BYTE PTR [rbp-0x12],0x53
7a4:   c6 45 ef 41              mov    BYTE PTR [rbp-0x11],0x41
7a8:   c6 45 f0 41              mov    BYTE PTR [rbp-0x10],0x41
7ac:   c6 45 f1 45              mov    BYTE PTR [rbp-0xf],0x45
7b0:   c6 45 f2 5d              mov    BYTE PTR [rbp-0xe],0x5d
7b4:   c6 45 f3 40              mov    BYTE PTR [rbp-0xd],0x40
7b8:   c6 45 f4 56              mov    BYTE PTR [rbp-0xc],0x56
7bc:   c6 45 f5 03              mov    BYTE PTR [rbp-0xb],0x3
7c0:   c6 45 f6 00              mov    BYTE PTR [rbp-0xa],0x0
7c4:   c6 45 f7 01              mov    BYTE PTR [rbp-0x9],0x1
7c8:   c6 45 f8 01              mov    BYTE PTR [rbp-0x8],0x1
7cc:   c6 45 f9 00              mov    BYTE PTR [rbp-0x7],0x0
7d0:   c6 45 fa 07              mov    BYTE PTR [rbp-0x6],0x7
7d4:   c6 45 fb 32              mov    BYTE PTR [rbp-0x5],0x32
7d8:   48 8d 45 b0              lea    rax,[rbp-0x50]
7dc:   48 89 c6                 mov    rsi,rax
7df:   48 8d 3d ee 00 00 00     lea    rdi,[rip+0xee]        # 8d4 <_IO_stdin_used+0x4>
7e6:   b8 00 00 00 00           mov    eax,0x0
7eb:   e8 20 fe ff ff           call   610 <__isoc99_scanf@plt>
7f0:   c7 45 fc 00 00 00 00     mov    DWORD PTR [rbp-0x4],0x0
7f7:   eb 1c                    jmp    815 <main+0xb5>
7f9:   8b 45 fc                 mov    eax,DWORD PTR [rbp-0x4]
7fc:   48 98                    cdqe
7fe:   0f b6 44 05 e0           movzx  eax,BYTE PTR [rbp+rax*1-0x20]
803:   83 f0 32                 xor    eax,0x32
806:   89 c2                    mov    edx,eax
808:   8b 45 fc                 mov    eax,DWORD PTR [rbp-0x4]
80b:   48 98                    cdqe
80d:   88 54 05 e0              mov    BYTE PTR [rbp+rax*1-0x20],dl
811:   83 45 fc 01              add    DWORD PTR [rbp-0x4],0x1
```

Assembly pro/cons

- Pro: Produces faster code (in theory)

Assembly pro/cons

- Pro: Produces faster code (in theory)
- Pro: Very-fine grained control (a.k.a. "I know what I'm doing")

Assembly pro/cons

- Pro: Produces faster code (in theory)
- Pro: Very-fine grained control (a.k.a. "I know what I'm doing")
- Pro: Educational purpose (give the compiler more hints)

Assembly pro/cons

- Pro: Produces faster code (in theory)
- Pro: Very-fine grained control (a.k.a. "I know what I'm doing")
- Pro: Educational purpose (give the compiler more hints)

- Con: Takes more time, compiler usually knows better than you (in practice)

Assembly pro/cons

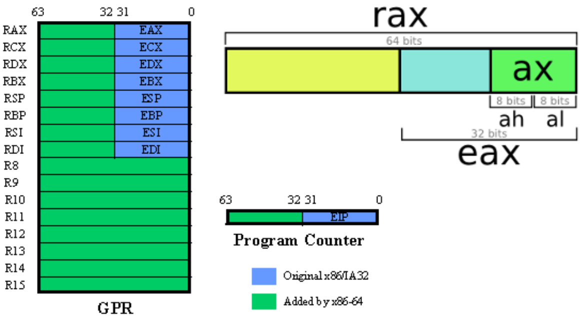
- Pro: Produces faster code (in theory)
- Pro: Very-fine grained control (a.k.a. "I know what I'm doing")
- Pro: Educational purpose (give the compiler more hints)

- Con: Takes more time, compiler usually knows better than you (in practice)
- Con: Need to be frugal w.r.t. variables (limited register count)
- Con: Easy to make non-maintainable spaghetti code

CPU registers

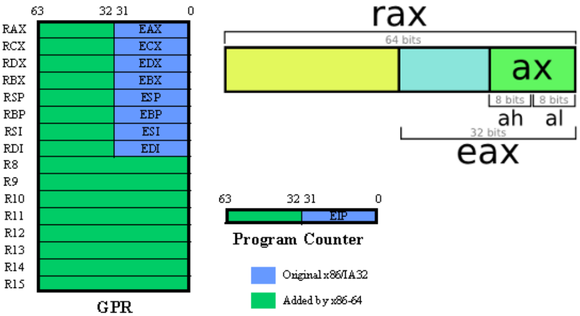
- A (finite) set of internal variables
- Some are general purpose (GP)
- Some are usually (but not always) used by the compiler in certain situations
- Some are always used for a specific purpose (instruction pointer, stack register)

x86-64 registers



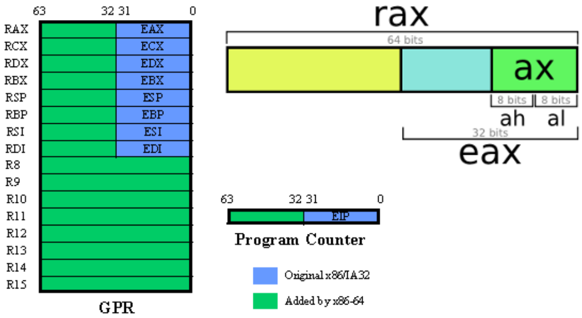
- RIP: Instruction Pointer

x86-64 registers



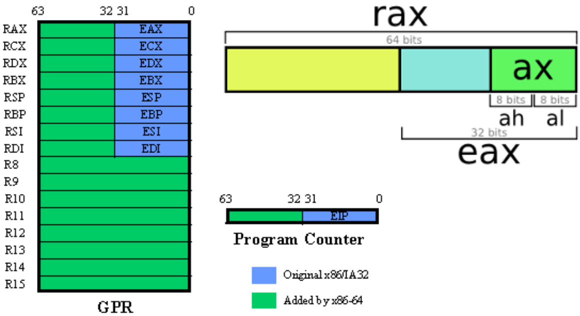
- RIP: Instruction Pointer
- RSP: Stack Pointer; RBP: Base Pointer (usually GP)

x86-64 registers



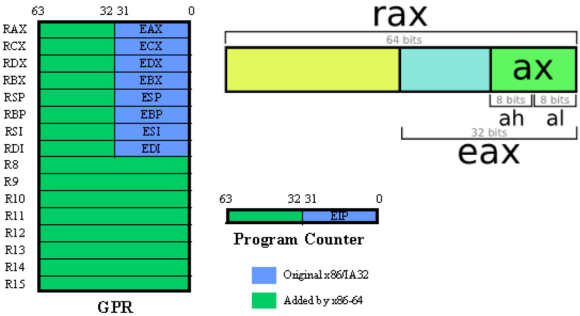
- RIP: Instruction Pointer
- RSP: Stack Pointer; RBP: Base Pointer (usually GP)
- R[A,B,C,D]X: GP; RCX (affinity in loop counters)

x86-64 registers



- RIP: Instruction Pointer
- RSP: Stack Pointer; RBP: Base Pointer (usually GP)
- R[A,B,C,D]X: GP; RCX (affinity in loop counters)
- RDI, RSI: usually GP (affinity in string ops, copy ops)

x86-64 registers



- RIP: Instruction Pointer
- RSP: Stack Pointer; RBP: Base Pointer (usually GP)
- R[A,B,C,D]X: GP; RCX (affinity in loop counters)
- RDI, RSI: usually GP (affinity in string ops, copy ops)
- Divisions: BYTE (AL), WORD (AX), DWORD (EAX), QWORD (RAX)

x86-64 instructions: arithmetic/logic

```
MOV RAX, 2021                    ; rax = 2021
```

x86-64 instructions: arithmetic/logic

```
MOV RAX, 2021                    ; rax = 2021
SUB RAX, RDX                     ; rax -= rdx
```

x86-64 instructions: arithmetic/logic

```
MOV RAX, 2021          ; rax = 2021
SUB RAX, RDX           ; rax -= rdx
AND RCX, RBX           ; rcx &= rbx
```

x86-64 instructions: arithmetic/logic

```
MOV RAX, 2021                  ; rax = 2021
SUB RAX, RDX                   ; rax -= rdx
AND RCX, RBX                   ; rcx &= rbx
SHL RAX, 10                    ; rax <<= 10
```

x86-64 instructions: arithmetic/logic

```
MOV RAX, 2021          ; rax = 2021
SUB RAX, RDX           ; rax -= rdx
AND RCX, RBX           ; rcx &= rbx
SHL RAX, 10            ; rax <<= 10
SHR RAX, 10            ; rax >>= 10 (sign bit not preserved)
```

x86-64 instructions: arithmetic/logic

```
MOV RAX, 2021            ; rax = 2021
SUB RAX, RDX             ; rax -= rdx
AND RCX, RBX             ; rcx &= rbx
SHL RAX, 10              ; rax <<= 10
SHR RAX, 10              ; rax >>= 10 (sign bit not preserved)
SAR RAX, 10              ; rax >>= 10 (sign bit preserved)
```

x86-64 instructions: arithmetic/logic

```
MOV RAX, 2021      ; rax = 2021
SUB RAX, RDX       ; rax -= rdx
AND RCX, RBX       ; rcx &= rbx
SHL RAX, 10        ; rax <<= 10
SHR RAX, 10        ; rax >>= 10 (sign bit not preserved)
SAR RAX, 10        ; rax >>= 10 (sign bit preserved)
IMUL RAX, RCX      ; rax = rax * rcx
```

x86-64 instructions: arithmetic/logic

```
MOV RAX, 2021          ; rax = 2021
SUB RAX, RDX           ; rax -= rdx
AND RCX, RBX           ; rcx &= rbx
SHL RAX, 10            ; rax <<= 10
SHR RAX, 10            ; rax >>= 10 (sign bit not preserved)
SAR RAX, 10            ; rax >>= 10 (sign bit preserved)
IMUL RAX, RCX          ; rax = rax * rcx
IMUL RCX               ; <rdx:rax> = rax * rcx (128 bit mul)
```

x86-64 instructions: arithmetic/logic

```
MOV RAX, 2021              ; rax = 2021
SUB RAX, RDX               ; rax -= rdx
AND RCX, RBX               ; rcx &= rbx
SHL RAX, 10                ; rax <<= 10
SHR RAX, 10                ; rax >>= 10 (sign bit not preserved)
SAR RAX, 10                ; rax >>= 10 (sign bit preserved)
IMUL RAX, RCX              ; rax = rax * rcx
IMUL RCX                   ; <rdx:rax> = rax * rcx (128 bit mul)
XOR RAX, RAX               ; rax ^= rax
```

x86-64 instructions: arithmetic/logic

```
MOV RAX, 2021              ; rax = 2021
SUB RAX, RDX               ; rax -= rdx
AND RCX, RBX               ; rcx &= rbx
SHL RAX, 10                ; rax <<= 10
SHR RAX, 10                ; rax >>= 10 (sign bit not preserved)
SAR RAX, 10                ; rax >>= 10 (sign bit preserved)
IMUL RAX, RCX              ; rax = rax * rcx
IMUL RCX                   ; <rdx:rax> = rax * rcx (128 bit mul)
XOR RAX, RAX               ; rax ^= rax
LEA RCX, [RAX * 8 + RBX]   ; rcx = rax * 8 + rbx
```

x86-64 instructions: memory

```
MOV RAX, QWORD PTR [0x123456]    ; rax = *(int64_t*) 0x123456
MOV QWORD PTR [0x123456], RAX    ; *(int64_t*) 0x123456 = rax
```

x86-64 instructions: memory

```
MOV RAX, QWORD PTR [0x123456]    ; rax = *(int64_t*) 0x123456
MOV QWORD PTR [0x123456], RAX    ; *(int64_t*) 0x123456 = rax
MOV EAX, DWORD PTR [0x123456]    ; rax = *(int32_t*) 0x123456
MOV AL, BYTE PTR [0x123456]      ; al = *(int8_t*) 0x123456
```

x86-64 instructions: control flow

```
JMP 0x1234                    ; rip = 0x1234
JMP [RAX]                     ; rip = *(int64_t) rax
```

x86-64 instructions: control flow

```
JMP 0x1234              ; rip = 0x1234
JMP [RAX]               ; rip = *(int64_t) rax
JZ/JE  0xABCD           ; if (zf) rip = 0xabcd
JNZ/JNE  0xABCD         ; if (!zf) rip = 0xabcd
```

x86-64 flag register

`EFLAGS`:

`(carry parity adjust zero sign trap interrupt direction overflow)`

- Carry flag: Addition, Subtraction
- Zero flag: Last operation result was 0
- Sign flag: Last operation result was < 0
- Overflow: Last operation result was $> 2^{register\_bitcount}$

x86-64 flag instructions

```
TEST RAX, RBX                     ; _ = rax & rbx; set SF, ZF, PF
                                  ; useful when checking for null vals
                                  ; and bit masks

CMP RAX, RBX                      ; _ = rax - rbx
                                  ; arithmetic comparisons
```

x86-64 instructions: stack raison d'etre

- In practice, we cannot use only 16 registers for all variables
- In practice, we cannot use only JMP for function calls

x86-64 instructions: stack raison d'etre

- In practice, we cannot use only 16 registers for all variables
- In practice, we cannot use only JMP for function calls
- To this end, each program is given a slab of blank memory called the stack
- How to use it efficiently?

x86-64 instructions: stack micro-operations

```
PUSH RAX                          ; rsp -= 8; *(int64_t*)rsp = rax;
```

x86-64 instructions: stack micro-operations

```
PUSH RAX                                ; rsp -= 8; *(int64_t*)rsp = rax;
POP RAX                                 ; rax = *(int64_t*)rsp; rsp += 8
```

x86-64 instructions: stack micro-operations

```
PUSH RAX          ; rsp -= 8; *(int64_t*)rsp = rax;
POP RAX           ; rax = *(int64_t*)rsp; rsp += 8

CALL 0x12345      ; PUSH RIP; JMP 0x12345

RET               ; POP RIP
```

x86-64 instructions: stack macro-operations

```
PUSH RBP                ; save previous frame base
MOV RBP, RSP            ; move frame base to current top
SUB RSP, 100            ; allocate 100 bytes on the stack
                        ; "push new stack frame"
```

x86-64 instructions: stack macro-operations

```
PUSH RBP                          ; save previous frame base
MOV RBP, RSP                      ; move frame base to current top
SUB RSP, 100                      ; allocate 100 bytes on the stack
                                  ; "push new stack frame"

MOV RBX, [RBP - 0x20]             ; rbx = *(int64_t*)(rbp-0x20)
                                  ; use the allocated space for storage
```

x86-64 instructions: stack macro-operations

```
PUSH RBP                        ; save previous frame base
MOV RBP, RSP                    ; move frame base to current top
SUB RSP, 100                    ; allocate 100 bytes on the stack
                                ; "push new stack frame"

MOV RBX, [RBP - 0x20]           ; rbx = *(int64_t*)(rbp-0x20)
                                ; use the allocated space for storage

LEAVE                           ; MOV RSP, RBP ; POP RBP
                                ; "pop current stack frame"
```

x86-64 instructions: conventions

- In order to use software modules (libraries, objects, etc) a standard must be set. Why?

x86-64 instructions: conventions

- In order to use software modules (libraries, objects, etc) a standard must be set. Why?
- How do you pass parameters to external functions? Memory? Stack? Registers?

x86-64 instructions: conventions

- In order to use software modules (libraries, objects, etc) a standard must be set. Why?
- How do you pass parameters to external functions? Memory? Stack? Registers?
- Calling conventions are used: cdecl, stdcall, fastcall.
- On 32 bit systems, parameters are passed on the stack, return in EAX
- On 64 bit systems, parameters are passed using registers, return in RAX
- Memorize this: 'RDI, RSI, RDX, RCX, R8, R9 (Linux)'

x86-64 instructions: syscalls

- In order to cross the application - OS limit, syscalls are needed
- File operations: read/write/close/open/create/remove
- Sleep, Select, Yield, Fork, Kill, GetTime
- Allocate/Release Memory
- Socket/Networking Operations
- IPC communication

```
MOV RAX, 0x2                ; Choose syscall number 2 (open)
MOV RDI, [RSP + 0x10]       ; Set first argument to some stack value
SYSCALL                     ; Invoke kernel functionality
```

What you need to know

- As a RE, writing ASM code by hand is not needed very often
- Reading ASM code is maybe 10% of the work
- However, knowing the basics is absolutely crucial and can be learned fast

Practice

- Any Questions?
- `http://pwnthybytes.ro/unibuc_re/02-lab.html`