Introduction to x86 Reverse Engineering

Melbourne Information Security Club

March 16, 2020

What is Reverse Engineering?

If we consider the ordinary process of *engineering* in the context of software.

$$\begin{array}{c}\text{Conceptual Idea or Design} \\ \big\downarrow \text{Development} \\ \text{High Level Code, (Python, Java, C etc.)} \\ \big\downarrow \text{Compiler} \\ \text{Machine Code / Assembly}\end{array}$$

What is Reverse Engineering?

Reverse Engineering is the opposite process.

$$\begin{array}{c}\text{Conceptual Idea or Design}\\ \Big\uparrow \\ \text{Machine Code/ Assembly}\end{array}$$

- That is, the end goal is to work backwards from a non human-friendly form of a program, in order to gain a conceptual understanding of what the program is doing.
- This workshop will teach you how to read (32-bit) x86 assembly in order to reverse engineer what a given program is doing.

How to disassemble Linux ELF Binaries

We first need to understand how to execute ELF binaries.

Example

Given an ELF Binary,

```
$ file behemoth0
behemoth0: setuid ELF 32-bit LSB executable, Intel
    80386, version 1 (SYSV),
```

We first make sure it is executable

```
$ chmod +x behemoth0
$ ls -l behemoth0
-r-sr-x--x 1 darrenx darrenx 5900 Aug  8  2019
   behemoth0
```

and then we can run it using ./

```
$ ./behemoth0
```

How to disassemble Linux ELF Binaries

Now that we know how to execute binaries, we will now use a debugger to look at the underlying assembly that we will need to reverse engineer.

Using gdb to disassemble a binary

Basic gdb commands summary sheet:

- View functions: `(gdb) info functions`
- Change disassembly from AT&T syntax to Intel: `(gdb) set disasembly-flavor intel`
- Disassemble function (e.g main): `(gdb) disass main`

Demo

Use gdb to disassemble the main function of a `hello_world` program.

Introduction to x86 Assembly

There are two fundamental concepts which you'll need to know.

Definition

A **register** is a quickly accessible storage location, located directly on the processor itself. We can think of them as essentially *untyped variables*

- The 32-bit general purpose registers we generally deal with are given the names EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP.
- The EIP register points to the next instruction to be executed.
- The ESP and EBP registers point to the top of the stack and the bottom of the current stack frame respectively.
- Although it is technically possible for a general purpose register to hold any 32-bit piece of data, registers are typically associated with a special purpose.

Example

For example,EAX typically stores the result of arithmetic operations.

Introduction to x86 Assembly

Definition

Instructions are the basic statements that tell the processor what to do.

Examples

- `mov eax, ecx` moves (or more accurately copies) data from the ECX register to the EAX register.
- `push eax` or `pop eax` places the value of EAX onto the stack, or removes the value at the top of the stack and places it into EAX respectively.
- `add eax, ebx` adds the values of the EAX and EBX registers, and stores the result in the EAX register. Other arithmetic operations are available, and in Intel syntax, the result is stored in the left operand.

Recognising Code Patterns

So technically, we now possess the requisite knowledge to reverse engineer binaries. Given any disassembled function we simply need to

- Read through every single instruction one at a time,
- figure out what effect the instruction has on each register,
- and we can just look up any instructions we don't know in the (2000 page long) Intel x86 Software Developer's Manual.
- .... right??

Main takeaway from this workshop

Trying to reverse engineer a program by reading through and interpreting every single instruction is ridiculous. What is more important to gaining a conceptual understanding of the program is to **recognise code patterns**.

The Simplest Example

The simplest program we can do is an empty function.

C code

```c
void func(){
   ;
}
```

When we look at the compiled binary in a disassembler, we get

Assembly

```
(fcn) sym.func 2
  0x000004f0      f3c3              ret
```

When a function is called, the return address is pushed onto the stack. At the end of a function, the `ret` instruction causes the program to jump to the address located at top of the stack.

Returning values

Lets try returning some values now.

C Code

```c
int func(){
  return 1337;
}
```

Assembly

```
(fcn) sym.func 6
  0x000004ed      b839050000      mov eax, 0x539
  0x000004f2      c3              ret
```

Note: 0x539 = 1337

From this, we can deduce that the general convention is to store function return values in the EAX register.

The Stack

- The stack is a region of memory used by the program to store local variables, function arguments, and any other data not immediately required.
- By convention, the bottom of the stack begins at 0xffffff, and grows backwards towards lower addresses.
- Recall that the ESP register points to the top of the stack, and the EBP register points to the bottom of the current stack frame. Hence the function's stack frame lies between these two pointers.

$$\text{ESP} \longmapsfrom \text{Function Stack Frame} \longmapsto \text{EBP}$$

- Establishing the stack frame (i.e moving ESP and EBP to their correct values) is achieved in the *function prologue* and the *function epilogue*.

Function prologue and epilogue

Let's try a function with a local variable.

C Code

```c
void func(){
    int a = 0;
}
```

Assembly

```
(fcn) sym.func 15
  /* Establish func()'s stack frame */
  0x000004ed      55              push ebp
  0x000004ee      89e5            mov ebp, esp
  0x000004f0      83ec10          sub esp, 0x10
  /* Assign 0 to the local variable 'a' stored on the
     stack */
  0x000004fd      c745fc000000.   mov dword [ebp-0x4], 0
  0x00000504      90              nop
  /* Clean up func()'s stack frame */
  0x00000505      c9              leave
  0x00000506      c3              ret
```

Note that `leave` is a short hand for `mov esp, ebp; pop ebp`.

Passing function arguments

Let's try a function that takes in arguments, firstly from the callers' side

C Code

```
int main(){
  return func(1,2);
}
```

Assembly

```
(fcn) main 17
  0x00000504      55                 push ebp
  0x00000505      89e5               mov ebp, esp
  /* Push function arguments onto stack in reverse
     order */
  0x00000507      6a02               push 2
  0x00000509      6a01               push 1
  0x0000050e      e8d3ffffff         call sym.func
  /* Clean up the stack space used by the arguments
     from before */
  0x00000511      83c408             add esp, 8
  0x00000512      c9                 leave
  0x00000513      c3                 ret
```

Passing function arguments

Now let's see what happens from the function's perspective.

C Code

```c
int func(int a, int b){
  return a + b;
}
```

Assembly

```
(fcn) sym.func 23
  0x000004ed      55              push ebp
  0x000004ee      89e5            mov ebp, esp
  /* Function arguments are accessed from the stack,
     relative to func()'s stack frame. */
  0x000004fa      8b5508          mov edx, dword [ebp+0x8]
  0x000004fd      8b450c          mov eax, dword [ebp+0xc]
  0x00000500      01d0            add eax, edx
  0x00000502      5d              pop ebp
  0x00000503      c3              ret
```

Control Flow: Selection

The last two basic code patterns deal with control flow.

C Code

```c
int func(int a){
  if(a > 3)
    return 1;
  else
    return 0;
}
```

Assembly

```
(fcn) sym.func 33
      0x000004ed      55            push ebp
      0x000004ee      89e5          mov ebp, esp
      0x000004fa      837d0803      cmp dword [ebp+0x8], 3
 ,=< 0x000004fe      7e07          jle 0x507
 |   0x00000500      b801000000    mov eax, 1
,==< 0x00000505      eb05          jmp 0x50c
||    ; CODE XREF from sym.func (0x4fe)
|`-> 0x00000507      b800000000    mov eax, 0
|     ; CODE XREF from sym.func (0x505)
`--> 0x0000050c      5d            pop ebp
      0x0000050d      c3            ret
```

Control Flow Graphs

- While it was very nice of `radare2` to include arrows in the disassembly of the previous slide to indicate jumps, in more complex functions those arrows get cluttered easily.
- For this reason, most reverse engineering platforms will possess the capacity to produce CFGs.

Demo

Use `radare2` to view a CFG of the function on the previous slide.

Control Flow: Repetition

The last code pattern we will explore is actually best represented in a CFG.

C Code

```
int func(int a){
    for(int i = 0; i < a; i++)
        a += i;
    return a;
}
```

Demo

Use `radare2` to view the CFG representation of a for loop. Identify the loop initialisation statement, the loop index variable, the guard condition, and the increment statement.

Where to from here?

- We have covered the fundamental code patterns that make up programs, and explored how they appear in assembly.
- Build up your knowledge of code patterns by writing your own code, and examining the corresponding disassembly like we have done today.
- What we have covered today forms the basis of **static analysis**. There is also **dynamic analysis** where one reverse engineers a program by executing it in a debugging environment and tracing the flow of data as the program is running.
- There are also many other architectures to explore beyond 32-bit x86.
- Practice by doing lots of CTF challenges! We have curated some for this workshop which you can play at `https://workshop-ctf.umisc.info/`.