Intro to Reverse Engineering
Day 1

Rowan Hart

September 5, 2020

General Announcements & Introduction

- I assume but don't require previous familiarity with some programming concepts.
- Caveat: you don't **have** to know anything to get something out of this.
- I will answer questions from Twitch, Youtube, and Discord.
- Docker container/source!

Insert a bit about me here . . .

What is Reverse Engineering? - Demo 0: Demo CTF Challenge

- **Official Definition:** "...the process by which a man-made object is deconstructed to reveal its designs, architecture, code or to extract knowledge from the object" [1]
- **Layperson's Terms:** "Doing whatever you need to do to figure out how something works, to whatever level of understanding you need, with or without documentation."
- So what counts?
  - Figuring out a co-worker's code after they get fired.
  - Taking apart tickle me elmo to identify the PCB components.
  - **(The one we care about today)** figuring out how a compiled program works.

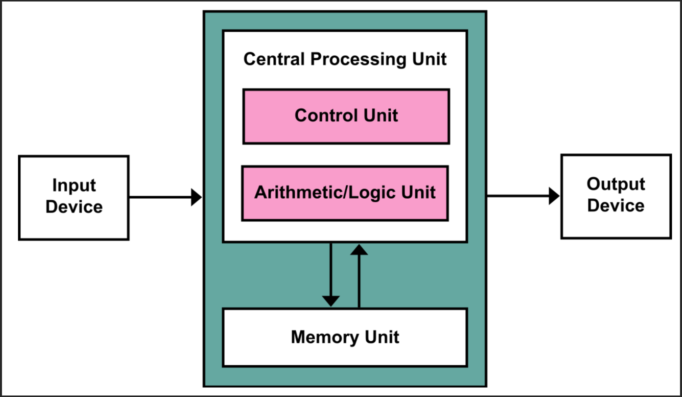
What's in a computer?

I'll just be talking about x86_64 machines throughout for consistency. They are actually pretty simple. I am ignoring a LOT of parts here, but the basics are:

- Memory (or RAM).
- Cache (related to RAM).
- Registers:
  - General purpose: rax, rbx, rcx, rdx, rsi, rdi, r8-15.
  - Base pointer rbp, stack pointer rsp, instruction pointer rip.
  - Segment registers cs, ss, es, ds, fs, gs.
  - Flag register RFLAGS/EFLAGS/FLAGS. Contains:
    - Carry (CF)
    - Parity (PF)
    - Adjust (AF)
    - Zero (ZF)
    - Sign (SF)
    - Trap (TF)
    - Some more things. Look it up!
  - 8 Floating point registers.
- A significant number of other hardware, software, and IO are also parts of the machine context but we don't need to worry about them except in specific cases. Also, we'll come back to those registers later.

Some guy named Von Neumann

Computers nowadays diagram-wise look like this (more or less):



Data Representations

In this presentation I'll be using several different data representations:

- Decimal (looks like you're used to)
- Hexadecimal (looks like 0x0000)
- Binary (looks like 0b00000000)

Just so you know ;)

Main Takeaway

If you learn nothing else about what a computer is, make sure to understand this:

1. Everything in a computer is either consuming, producing, modifying, or storing data.
2. Different parts of a computer represent data differently, but it's all just bits.

Language Types - Demo 1: Python vs Java vs C

(Generally) two kinds of programming languages:

- Compiled (C, C++, Rust, Java, Haskell, Go, . . .)
- Intepreted (Ruby, Lua, Python, Bash, PHP, Javascript)

```c
// Some Example C Code
#include <stdio.h>

int bar(int n) {
    return n + 20;
}

int foo(int n) {
    return bar(n) * 2;
}

int main(int argc, char ** argv) {
    printf("%d\n", foo(10));
}
```

```python
# Some Example Python Code

def bar(n):
    return n + 20

def foo(n):
    return bar(n) * 2

print(foo(10))
```

Compiled Languages

Our focus is on reverse engineering **compiled** code.

- Non-compiled programs can be reverse understood by most anyone with language skill.
- Compiled languages require different skills and tools to understand.
- For the rest of this session, we are going to talk about the C language on the 64-bit x86 Platform because it is one of the most familiar and common compiled languages.
- The same principles apply to other languages and compilers as well.
- Let's talk about what compilation is...

How a C compiler works - Demo 2: Processing + Compilation Steps

The C compiler (we will specifically be looking at GCC) takes source code as input and outputs an executable, or **binary**. This happens in several steps: [2]

- Preprocessing (gcc -E)
- Compiling (gcc -S -masm=intel)
- Assembly (gcc -c)
- Linking (gcc)

We are here only considering the ELF (Executable and Linkable Format). The three most common executable formats are:

- ELF (Linux/Unix)
- Mach-O (OSX/MacOS)
- PE (Portable Executable) (Windows)
- DOS (Really old Windows executables)

The ELF Format - Demo 3: Readelf, Sections

The basics of the ELF format aren't terribly complicated. ELF files, however, can be very complicated.

- Introducing our first tool: *readelf*!
- *readelf* parses an ELF file and outputs information about it in human-readable format.

The key parts of the ELF file are:

- ELF Header
- Program Headers
- Section Headers
- Symbol Table

There's more to it, but we're going to focus on the basics.

ELF Header

The ELF header is a structure of the following form: [5]

```
typedef struct elf64_hdr {
      unsigned char e_ident[0x9];   /* ELF Type identifier:
                                    - 4 byte magic number (0x7f E L F)
                                    - 1 byte either 32 (0x1) or 64 (0x2) bit
                                    - 1 byte either big (0x1) or little (0x2) endian
                                    - 1 byte always (0x1), reserved
                                    - 1 byte identifier of OS ABI (more on ABI later)
                                    - 1 byte ABI version identifier (ignored usually)
                                    - 1 byte unusued zero padding */
      Elf64_Half e_type;      /* ELF Object file type (executable, shared obj, etc) */
      Elf64_Half e_machine; /* ISA Specification */
      Elf64_Word e_version; /* Always set to 1 */
      Elf64_Addr e_entry;   /* Entry point virtual address */
      Elf64_Off e_phoff;    /* Program header table file offset (offset in this file) */
      Elf64_Off e_shoff;    /* Section header table file offset (offset in this file) */
      Elf64_Word e_flags;   /* None defined yet for Linux x86/x86_64 */
      Elf64_Half e_ehsize;  /* The size of the ELF header (this header) */
      Elf64_Half e_phentsize; /* The size of a program header */
      Elf64_Half e_phnum;     /* The number of program headers in this file */
      Elf64_Half e_shentsize; /* The size of a section header */
      Elf64_Half e_shnum;     /* The number of section headers in this file */
      Elf64_Half e_shstrndx;  /* The index in the list of section headers
                              for the header that points to the string table */
} Elf64_Ehdr;
```

The ELF header tells the loader (we'll talk about this later) information about the image. What version it is, what type of computer this is for, and where in this file to find the code, data, and other info needed to run the program.

Program Header

The ELF Program Header is a structure of the following form: [5]

```
typedef struct elf64_phdr {
    Elf64_Word p_type;      /* Header Segment Type
                            (dynamic linking, loadable, relro, \ldots) */
    Elf64_Word p_flags;     /* Segment-dependent flags
                            (position for 64-bit structure). */
    Elf64_Off p_offset;     /* Segment file offset */
    Elf64_Addr p_vaddr;     /* Segment virtual address */
    Elf64_Addr p_paddr;     /* Segment physical address */
    Elf64_Xword p_filesz;   /* Segment size in file */
    Elf64_Xword p_memsz;    /* Segment size in memory */
    Elf64_Xword p_align;    /* Segment alignment, file & memory */
} Elf64_Phdr;
```

Each program header tells the loader what type of data is contained in a part of the executable file, where in memory the part of the file should be located, and more.

Section Headers

The ELF format consists of **sections**, each of which contains some type of data the program uses. There are a lot of different sections, but the ones we really care about are:

- .text: (R/X) Contains the actual instructions the program executes.
- .data: (R/W) Contains initialized data (non-const variables with a value).
- .rodata: (R) Contains initialized read-only data (const variables, defines).
- .bss: (R/W) Contains uninitialized data (non-const variables with no initial value).
- .got: (R/W) Global Offset Table, important in pwning, holds addresses of dynamic functions.
- .got.plt: (R) Procedure Linkage Table, stub functions called to either jump to a GOT address or trigger the loader to look up a dynamic function.

For an exhaustive description, read [4].

How Programs Are Run - Demo 4: Fork / Exec and System Calls

In Linux, running a program consists of a couple steps. We'll walk through those steps in a code example:

```c
int main(int argc, char ** argv) {
    pid_t pid = fork();
    char * const args[] = {"ls", "-lah", "/bin/"};
    switch(pid) {
        case 0:     /* We are in the child process, see `man 2 fork`. */
            /* Execute `ls -lah /bin` */
            printf("Hello, I am the child process! My pid is %d\n", getpid());
            execvp(args[0], args);
        case -1:    /* Something went wrong and the fork() call failed. */
            /* Print an error */
            perror("fork");
            exit(1);
        default:    /* We are in the parent process and pid == the child pid */
            /* Print out something and exit */
            printf("I have run the ls command from PID %d\n", pid);
            exit(0);
    }

}
```

Debugging

Debugging! We don't have to follow the rules the people who wrote a program set for us!
We'll use gdb (GNU Debugger) for this training, but there are others. gdb can do a ton, but here are the critical commands:

1. run/continue: start over / continue to the next breakpoint
2. break *address: set a break (stop) point at a particular place in the binary
3. step: step one instruction
4. next: forward one line of code (similar to step . . . unless you have source)
5. set: assign a variable to a value ex. set $rax = 10
6. x/ : examine a location of a particular format
7. print: print a variable (similar to x)

The Dynamic Loader/Intepreter - Demo 5: Stepping in GDB

We've been discussing loading up to this point, so lets talk about what the loader is and how it works. The loader is a program that works with the kernel to:

- Kernel: processes the ELF header to figure out what intepreter (*ld*) to use.
- Kernel: create a processes image (we're going to ignore virtual memory for now [3]) which includes:
    - Create a buffer to hold the program image (literally the total size of all segments, plus some other stuff).
    - Map process segments into memory with correct permissions.
    - Perform virtual address randomization (we're ignoring how though).
    - Set up the stack with information the loader needs to start the program.
    - Zero all registers, erasing previous program (remember *(*exec())?)
- Kernel: call start_thread() which includes (Assuming default RTLD_LAZY):
    - Relocate the dynamic linker itself (initialize start entries in GOT).
    - Take the previously stacked information the loader needs from the stack.
    - Set up environment variables.
    - Get information about each dynamic section.
    - Get information from the PHT (program header table).
    - Load shared libraries.
    - Relocate shared libraries.
    - Start the program (usually from main).

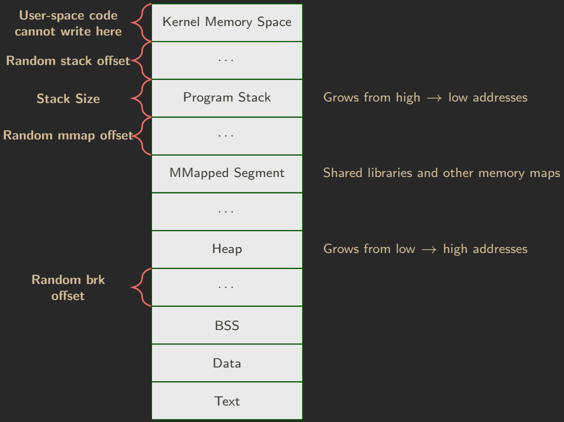
The Dynamic Loader/Interpreter pt. 2

Lets talk about the **main** job of the dynamic loader...to dynamically load things! Whenever a shared library function is called for the first time...

- The program jumps to the PLT entry for the shared library function (like printf).
- The loader function *fixup()* finds the actual address in memory of the function we want.
- The loader replaces the GOT entry for that function with its actual address.
- The program continues executing that function as if nothing strange happened.
- Each subsequent call just jumps to the (now filled in) GOT entry address instead of doing a fixup.

Program Images

We previously mentioned that the kernel creates a program image when it runs a new program. This is what we really care about from the last few pieces of info. Lets look at what that image looks like:



Note that as before, this is somewhat of an oversimplification. Take this as the basics, not an end-all-be-all.

Assembly Language

Assembly language is the human-readable representation of the language computers understand (machine code).

If you remember when we compiled with -S, we got something I called *assembly*. Like any programming language, assembly has syntax. We'll consider two main syntaxes:

- Intel Syntax. (Used by nasm + me)
- AT&T Syntax. (Used by gcc + gas)

An example of both:

- mov rax, [rbx + 0x10];
- mov 0x10(%rbx), rax;

Both of these instructions move a value at *(rbx + 16) into rax.
Notice that rax is on a different side of the comma. In Intel syntax, the syntax is *destination, source*, while AT&T is *source, destination*. Google is your friend!

Some Important Instructions - pt. 1

Most instructions you shouldn't bother memorizing. However, there are a few instructions you should definitely know.

- **push** [arg]: push something to the stack, decrement rsp.
- **pop** [arg]: pop something from the stack into arg, increment rsp.
- **mov** [dest], [src]: Copy a value from one location to another.
- **call** [arg]: push RIP to the stack and jump to a function.
- **add** [arg1], [arg2]: arg1 = arg1 + arg2. Other arithmetic instructions include:
  - sub (subtract).
  - mul (multiply).
  - div (divide).
  - xor (xor).

Some Important Instructions - pt. 2

Perhaps the most important thing we'll look at is control flow.
We'll look at some control flow graphs to get an idea of how these work, but the primary instructions below are:

- **jmp** [arg]: Jump to another location in code. There are other jump instruction types:
  - jle (jump if less than or equal)
  - jge (jump if greater than or equal)
  - jne (jump if not equal)
  - je (jump if equal)

  These don't actually mean equal though, they check something called FLAGS. How are FLAGS set? With these, mostly:
- **test** [arg1], [arg2]: Performs a bitwise and on the two operands. Used often to check if something is zero. Sets ZF accordingly.
- **cmp** [arg1], [arg2]: Compares two values and sets FLAGS accordingly:
  - If $[arg1] == [arg2]$: Set ZF (Zero Flag) to 1, Clear (set to 0) CF
  - If $[arg1] < [arg2]$: Clear ZF, Set CF
  - If $[arg1] > [arg2]$: Clear ZF, Clear CF

Stack and Heap

Remember that we talked about a process image with a stack and heap? There are two places we can put things in a program:

- The stack: each function has what is called a stack frame (detail in next slide). This is where local variables go like:

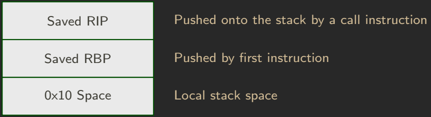
  ```
  foo (int b) {
      int x = 1; // This is a variable in foo()'s stack frame
      int y = 2; // This is another
  }
  ```

- The heap: heap data is allocated by requesting it with malloc and deallocated by using free. Heap data is user-managed.

Stack Frames

Every time a function is called, something called a stack frame is created. Lets go through an example function:

```
/* This is the program Prologue */
0x000000000040113a <+0>:        push   rbp
0x000000000040113b <+1>:        mov    rbp,rsp
0x000000000040113e <+4>:        sub    rsp,0x10
```


Saved RIP
Pushed onto the stack by a call instruction
Saved RBP
Pushed by first instruction
0x10 Space
Local stack space

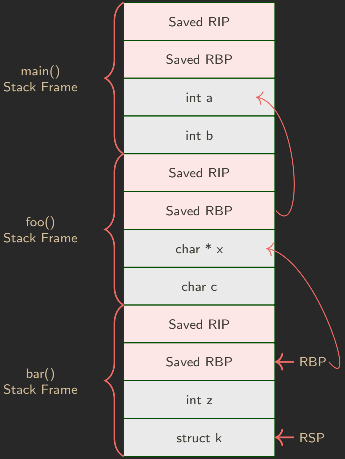
The program body would go in here . . .

```
/* This is the program Epilogue */
0x0000000000401169 <+47>:        leave
0x000000000040116a <+48>:        ret
```

These instructions pop the saved RBP from the prologue back into RBP to return the stack frame's context back to the caller of the current function.

Function Calls

Remember when we said *every* time a function is called?
Functions that call other functions will create a program stack like this:



Calling Convention/ABI

Calling convention is the way functions are called and argument passed. In x86_64 on linux, the calling convention is called System V.

- Arguments are passed in registers rdi, rsi, rdx, rcx, r8, r9, in order.
- Additional arguments are passed on the stack.
- Return values are returned in rax.
- Functions must preserve rbx, rsp, rbp, r12-15.

Reverse Engineering - Demo 6: No Source

We finally have enough base knowledge about programs, calling convention, and what have you.

Let's reverse engineer!
The primary steps of reverse engineering are:

- Figure out what you're dealing with.
- Survey the target for key information to speed analysis.
- Identify key locations in the program.
- Determine control flow through key locations.
- Dive deep on an important part until you get what you want or realize it isn't there.
- Rinse and repeat the last four.

Previous Slide in Practical Terms

That was very idealistic. In more practical terms, what we need to do is:

- Run diagnostic tools. file, binwalk, readelf. If it's a weird format, find a way to load it into a RE tool or disassemble it.
- If strings are in the program, *read them*.
- Disassemble or decompile the program (you will likely want both easily accessible).
- Using strings as signposts (or descending from main), label functions with likely names.
- Once low hanging, easily identifiable functions have been labeled, explore more in depth in specific areas.
- Use data initializers to determine structure formats, sizes, and uses.
- Obtain a view into the program sufficient to get the required info. (Yes yes, rest of the fucking owl)

That last one is VERY different from CTF to industry VR (Vulnerability Research).

RE Construct 1: Selection - Demo 7

Lets look at an example of selection.

Questions?

I know you have them. . . ask away!

References

[1] W. Commons. Reverse engineering. URL https://en.wikipedia.org/wiki/Reverse_engineering.

[2] I. Free Software Foundation. Using the gnu compiler collection (gcc). URL https://gcc.gnu.org/onlinedocs/gcc-10.2.0/gcc/.

[3] M. Gorman. Understanding the linux virtual memory manager. URL https://pdos.csail.mit.edu/~sbw/links/gorman_book.pdf.

[4] T. I. S. (TIS). Executable and linkable format (elf). URL http://www.skyfree.org/linux/references/ELF_Format.pdf.

[5] L. Torvalds. elf.h. URL https://github.com/torvalds/linux/blob/master/include/uapi/linux/elf.h.