COMPSCI 390R

Format String Exploits

The not fun ones

Topics to Cover:

1. Project 2 due 3/21 at midnight (no guarantees we answer campus wire during break!)

2. Project 1 Solutions coming out sometime tonight

3. Presentation Groups due tonight, need all members and 3 projects ideas
    - If you don’t have a group, submit yourself alone and with ideas so we can match you with others

Recap of what we've learned last class:

Return-Oriented-Programming

- Instead of writing our own shellcode, reuse existing code
- Can use return "gadgets" to jump around the program
- No longer need a large buffer for shellcode (just large overwrite)
- If we have a address leak to LIBC functions we can use gadgets and functions from the entire LIBC

Issue:

Finding memory leaks in the wild is hard

printf(“%p\n”, &gets) will never be seen in the real world unfortunately

Are there exploits we can find that will lead to memory leaks?

Let’s Look at Some Code

Echo Program:

```c
#include <stdio.h>

int main() {
        printf("I'll echo whatever you send me \n>>");
        char buf[256];
        fgets(buf, 255, stdin);
        printf(buf);
}
```

Echo Program (bug):

```c
#include <stdio.h>

int main() {
        printf("I'll echo whatever you send me \n>>");
        char buf[256];
        fgets(buf, 255, stdin);
        printf(buf);
}
```

Sending the buffer directly allows us to control the format string parameters

Printf In-depth (Format String Specifier):

The format string specifier has multiple possible flags

- %c - Character
- %s - String
- %i/d - Decimal integer
- %x - Hexadecimal integer
- %p - Pointer

h stands for half, while l stands for long, so you can do %ld and get a long integer or hhd for a single byte as an integer (these are promotions and can run into type conversion issues)

Printf In-depth (Parameters):

printf can take any number of variables

- First parameter will be interpreted as a format string specifier. This lets printf know how many parameters there should be
- RDI is used for the format string specifier, RSI, RDX, RCX, r8-r9 used for the rest of the parameters
- After those registers are used, if there are more variables then they are pushed onto the stack
- printf keeps an internal separate stack pointer to know where the next variable is

How printf keeps track of how many parameters there are in registers vs stack
if there’s more format specifiers than arguments isn’t known by us and

Printf In-depth (Leaking Stack Info):

With a format string exploit, we can leak stack variables

printf(“%p %p %p %p %p %p”) will pop values from the stack and print them out as pointers for use to see

Useful to leak unknown values on the stack, or the return address of our program if the memory layout is randomized

Printf In-depth (Leaking Stack Info):

This can take a lot of bytes if we want to leak something high on the stack

Luckily printf has a built in way to pop the nth value on the stack

The ‘$’ character allows us to get the nth pointer the stack by ‘$n%p’

ex: printf(“%5$p”) will print the 5th pointer on the stack

This will save roughly 2*n bytes in our payload where n is the location on the

Printf In-depth (The Dangerous Part):

Format Strings have another flag...

- %n - Prints nothing but writes the number of characters written so far into its parameter pointer (should be an integer pointer so it writes 4 bytes)
- We can add a pointer to our payload, which will then be on the stack and we can reference with the %(stack location)$n
- With hhn and hn, we can write 1 or 2 bytes accordingly (this overflows)

ex: printf(“AAAA%n”, 0x1337000) writes 4 into 0x1337000

ex: payload = “AA%2$n” + p64(0x1337) if the location of 0x1337 is 2nd

Printf In-depth (Overview):

Format String bugs can do the following:

- Leak values on the stack.
  - Note this doesn’t just have to be the current function’s stack, since the call stack is one block if we leak enough we can see data from previous functions
- Write value to memory
  - Given a pointer to the desired memory region we can write bytes and change values accordingly

How Does This Help?

Lets Learn More About ELFs:

A program can be compiled statically or dynamically

- If it is compiled statically all used LIBC functions are stored directly into the ELF file
- If it is compiled dynamically we will load LIBC functions in memory during execution from the one LIBC file on your system
  - This makes it so you don’t need to have the same repeated code thousands/millions of times on your hardware

Statically Linked Programs:

Statically linked programs are much simpler

- The compiler can hardcode jmp/call addresses for LIBC functions as though it’s a normal function that we wrote

Statically linked programs are much simpler to exploit

- If a function such as fgets is used in a program, we know the offset it is at and thus can use gadgets and ROP from that function to write exploits
- No memory leaks are generally needed unless we need to access something that is dynamically generated by the program such as

Dynamically Linked Programs:

Dynamically compiled programs are much more complex

- The location in which LIBC is loaded into memory is randomized by the operating system during process setup.
- The ELF must have a consistent way to call these functions without editing the code of the program for every instance (kind of). All our call/ret/jmps should have consistent values from each execution

But how?

Introducing the PLT and GOT:

PLT:

- PLT stands for the Procedure Linkage Table
- A table of offsets that maps a function to an offset in the GOT
  - ex: If we want to call fgets, we call fgets@plt, which redirects us to the fgets entry of the GOT

GOT:

- GOT stands for the Global Offset Table
- It’s a table of dynamically generated addresses to functions in LIBC, populated the first time a function is called in the program. LIBC global

GOT In-depth:

GOT Structure (Basically a big array of addresses):

- GOT[0], GOT[1] are information for the dynamic linker
- GOT[2] is a pointer to the dynamic linker code
- Remaining entries are function addresses that are generated during the first call to the function
- Function entries begin not with the function address but with the address of a PLT entry to call the dynamic linker

PLT In-depth:

PLT Entries/Function Setup:

- If we call a dynamically called function we actually call the PLT
  - ex: fgets@plt, printf@plt
- First call the GOT entry of the function we want
- If it’s the first time, the GOT redirects back to the PLT, which then calls the dynamic linker with the functions GOT offset it wants to setup

Abusing the PLT and GOT:

Leaking:

Since the PLT and GOT locations aren’t randomized in the program layout, if we can leak arbitrary memory locations with a format string exploit (think printf(“%n$p”) where it pointer to a value we put on the stack), we can find where LIBC is located

Writing:

Since the GOT is only updated during the first call of a function, if we call “puts” once, then overwrite it’s GOT value with the location of shellcode, the next time “puts” is called it will redirect execution to the shellcode