NUS CS-3235: Computer Security March 31, 2020

Assignment 2 Report

Lecturer: Reza Shokri *Student: Luca KRUEGER A0209176B*

1 Buffer Overflow

The upper limit of the `for` loop in function `bof` of `buffer_overflow.c` can be set to a maximum value of 128 if the program was able to read $64B$ from each of the two input files. The iterator of the loop is used to access the buffer `buf`, which is defined as char array of size 64. Hence, the loop can be exploited to overflow the buffer and to overwrite the stack. This will be used to change the functions return pointer to an executable shellcode, which will be inserted into the buffer.

When overwriting the stack, other variables such as the iterator `idx` and the upper limit of the loop `byte_read1 + byte_read2` will also be overwritten. To avoid a unwanted termination of the program due to changed iterators of the loop, one need to know precisely where these variable are stored in the stack and which values they hold at the moment of overwriting. For this exercise, the address randomization ASLR is deactivated.

I used the tool `gdb` to observe the variable's addresses and its relative positions to the buffer `buf`. The command `p &byte_read2` for example returns the address of the variable `byte_read2` on the stack after entering the gdb environment through `gdb ./buffer_overflow exploit1 exploit2`.

The following mapping can be observed: (Table 1)

Variable	Address	Length	Index *(rel. to buf)*	Preferred value
`buf[0]`	`0x7fffffffe190`	1B	0	shellcode
`buf[63]`	`0x7fffffffe1cf`	1B	63	
`byte_read2`	`0x7fffffffe1e4`	4B	$84 - 87$	$40_h = 64$
`byte_read1`	`0x7fffffffe1e8`	4B	$88 - 91$	$40_h = 64$
`idx`	`0x7fffffffe1ec`	4B	$92 - 95$	92
`idx2`	`0x7fffffffe1dc`	4B	$76 - 79$	don't care
`idx1`	`0x7fffffffe1e0`	4B	$80 - 83$	don't care
`return`	`0x7fffffffe1f8`	8B	$104 - 111$	`0x7fffffffe210`

Table 1: Variable locations on the stack

This knowledge can be used to generate the input files accordingly to overwrite the variables and the return pointer correctly. Unfortunately the input files are not directly fed into the buffer. The program alternates between the two inputs such that for even `idx` input file 1 is used and for uneven `idx` input file 2. I came up with a short python script `buf_gen.py`, which generates a string, a concatination of the preferred variable values, and which then writes this string in the same alternating manner into the two files.

2 Format String Attack

In the program `format_string.c` line 14 an unfiltered input string (stored in the buffer `buf`) is used as first argument for the format string function `printf()`. Therefore, if the buffer respectively the input string contains of format string modifiers such as `%n, %p, ...` the function will search for more arguments stored in the registers (argument $0 - 7$) and on the stack (argument $7 - n$) and format them as specified.

The input string will be generated and stored in a file by a single command such as:

```
echo -ne '%4919c%8$nAAAAAA\x1c\x50\x75\x55\x55\x55\x00\x00' > payload
```

The string above will be interpreted as follows:

`%4919c%` : prints $4919 = 0x1337$ characters to `stdout`, used to set the internal output counter to this value.

`%8$n` : accesses the 8th argument with respect to the printf and write the number of bytes already printed to the adress provided.

`AAAAAA` : padding A's to align the following pointer address to a 8B stack entry

Rest: address of `jackpot` in little endian byte order

3 Return-oriented Programming

The function `rop` inside `rop.c` checks input values to be smaller than 24 to avoid a buffer overflow. The user input is written into a `signed long`, checked for the condition and later casted to a `unsigned long` namely `size_t`. The casted value `read_size` tells the `read` the amount of characters to be read. Therefore, a user can enter a value $x < 0$ to pass the condition $x < 24$, but cause the function to read $|x| >> 24$ characters which will overflow the buffer. By overflowing the buffer, return addresses can be overwritten to point to other instructions within the execution space.

I used a python script `sample.py` to generate the input file `exploit`. The first 24 characters are needed to fill the buffer, after this the function return pointer can be overwritten. Starting from this return call, other instractions can be chained together by adding pointers to them on the stack. The only constraint to these instructions is, that they have to return to the next entry on the stack to infer the next instruction. The program which will be executed after entering the return, reads a filename from `stdin`, write to the file and closes it. The user input must not exceed a total length of 8B and must be terminated with a zero byte. This can be generated depending on the terminal used, using the shortcut `Ctrl+@`.