NUS CS-3235: Computer Security March 16, 2020

Assignment 2 Report

Lecturer: Reza Shokri *Student: Pedro Teixeira A0208300Y*

Notes

- Python scripts were used to generate the payloads required to perform the attacks.
- An effort was mad to document well the source code used to build the exploits and so it may help clarify anything omitted in this document for the sake of brevity.
- Each exercise's folder includes a script (`exploit.sh`) that automates the sequence of commands needed to perform the attack, and verifies that ASLR is turned off.
- For instructions to manually run the attacks, please refer to the README file included.
- In exercises 1 and 2, the buffer and global variable addresses are determined at the time the payload is built by the Python scripts, which execute the programs capturing the standard output and extracting the addresses.
- All exploits were built and executed inside the VM provided for tutorial 2.
- The exploits in exercises 1 and 2 work well outside `gdb`. However, the exploit for exercise 3 can only be performed inside `gdb`.

1 Buffer Overflow

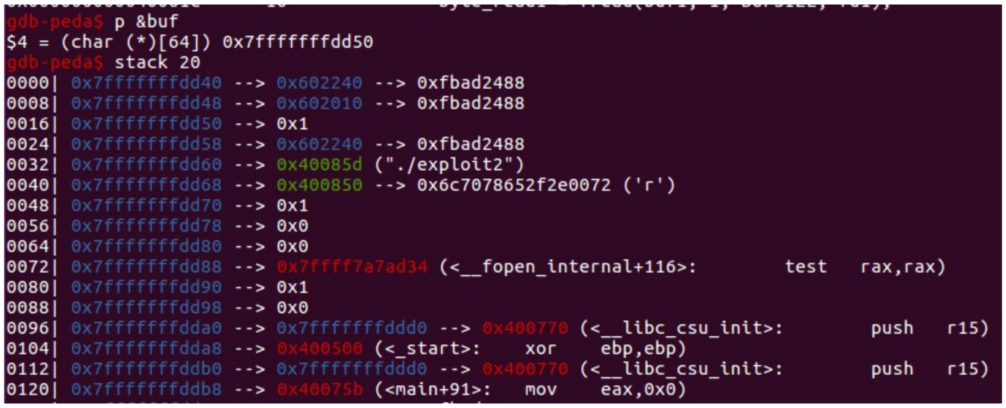
To cause a buffer overflow, all we need to ensure is that the same number of bytes is read from both the `exploit1` and the `exploit2` files, so they must have the same length to prevent the execution being stopped earlier.

We observe that the way the two initial buffers (`buf1` and `buf2`) are copied into the main buffer (`buf`) is that the bytes are interleaved, starting with a byte from `buf1`. Therefore, to write our payload to the files, we created a Python function that performs the inerse operation (given a string of bytes, splits it into two such that the buffer copying operation reconstructs the initial string).

We must also note that due to lines 19 and 20, the 64^{th} byte in each of the initial buffers gets overwritten with a NULL byte (0x0). Therefore, a limitation on the distance between the return address and the start of the main buffer is imposed on our traditional shellcode injection.

Since the start of the main buffer is printed by the program, we choose to place the shellcode at that address, for simplicity.

By running the program inside `gdb` and pausing it right after the local variables in the `bof` function variables are initialized, we observe that the return address is on the stack at address `0x7fffffffddb8`, 104 bytes away from the initial buffer address:

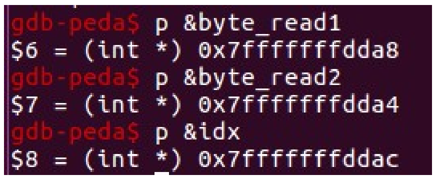


```
gdb-peda$ p &buf
$4 = (char (*)[64]) 0x7fffffffdd50
gdb-peda$ stack 20
0000| 0x7fffffffdd40 --> 0x602240 --> 0xfbad2488
0008| 0x7fffffffdd48 --> 0x602010 --> 0xfbad2488
0016| 0x7fffffffdd50 --> 0x1
0024| 0x7fffffffdd58 --> 0x602240 --> 0xfbad2488
0032| 0x7fffffffdd60 --> 0x40085d ("./exploit2")
0040| 0x7fffffffdd68 --> 0x400850 --> 0x6c7078652f2e0072 ('r')
0048| 0x7fffffffdd70 --> 0x1
0056| 0x7fffffffdd78 --> 0x0
0064| 0x7fffffffdd80 --> 0x0
0072| 0x7fffffffdd88 --> 0x7ffff7a7ad34 (<__fopen_internal+116>:        test   rax,rax)
0080| 0x7fffffffdd90 --> 0x1
0088| 0x7fffffffdd98 --> 0x0
0096| 0x7fffffffdda0 --> 0x7fffffffddd0 --> 0x400770 (<__libc_csu_init>:        push   r15)
0104| 0x7fffffffdda8 --> 0x400500 (<_start>:    xor    ebp,ebp)
0112| 0x7fffffffddb0 --> 0x7fffffffddd0 --> 0x400770 (<__libc_csu_init>:        push   r15)
0120| 0x7fffffffddb8 --> 0x40075b (<main+91>:   mov    eax,0x0)
```

Therefore to overflow the stack up to and including the return address we need $104 + 8 = 112$ bytes in our payload, which fits within the limitation earlier identified.

To preserve the behaviour of the copying loop and not risk it iterating forever, we must preserve the values of `byte_read1` and `byte_read2` and to make sure the copying is done in correct order we must also preserve `idx`.

We know that `byte_read1` and `byte_read2` will both take the value 56, because for a payload of 112 bytes, that is the length of each file. To determine where to put those values we can once again use `gdb`:



```
gdb-peda$ p &byte_read1
$6 = (int *) 0x7fffffffdda8
gdb-peda$ p &byte_read2
$7 = (int *) 0x7fffffffdda4
gdb-peda$ p &idx
$8 = (int *) 0x7fffffffddac
```

The value we want to replace in `idx` is not as trivial to calculate, but basically we need to know the loop iteration in which the variable is overwritten. Since each loop iteration writes one byte we calculate the distance from the start of the buffer to the `idx` variable: `07fffffffddac - 0x7fffffffdd50 = 0x5c (92)`. Therefore we must write the value 92 on `idx`.

The final structure of our payload is as follows, totalling 112 bytes:

- 27 bytes of shellcode
- 57 bytes of dummy data
- 4 bytes expressing the value 56 to overwrite `byte_read1`
- 4 bytes expressing the value 56 to overwrite `byte_read2`
- 4 bytes expressing the value 92 to overwrite `idx`
- 8 more bytes of dummy data
- 8 bytes of the address of the beginning of the buffer which contains the shellcode

The first 57 bytes of dummy data will cover the distance between the end of our shellcode and the variables we wish to overwrite with specific data. The 8 bytes of dummy data just before the return address cover the remaining distance.

2 Format String Attack

For this attack we started by running the program to identify the address of the global variable `jackpot`, which we wish to overwrites: `0x601074`.

Then we aim to construct a string that can write to that memory location when `printf()` is called with it as argument. For this we use the `%n` format specifier, in a similar way to what is explained in the instructions for tutorial 2.

The memory location to write to can be specified as one of the argument slots of the `printf()` call. Since the first 5 arguments are passed in registers, to pass the address on the stack we must specify argument 6 or larger. Since the address needs to be aligned to 8 byte words, and the string will already contain some other bytes before the address we can specify argument 7 and then pad until its location.

Our payload so far is: `%7$nAAAA\x00\x00\x00\x00\x00\x74\x10\x60`.

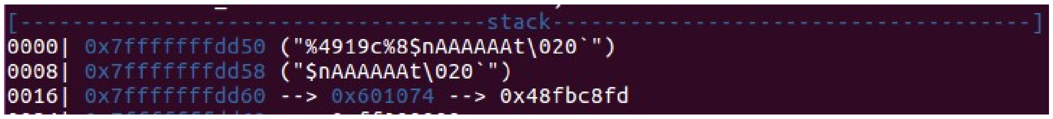
Note that we has to pad the address with 0s to extend it to 8 bytes.

Since all we can write is the number of characters printed before the format specifier is reached, this will write the value 0 into `jackpot`. To achieve our goal we must build our payload with 4919 characters followed by the `%n` format specifier. To make this possible with a buffer of only 128 bytes, we use `%4919c`, which causes 4919 spaces to be printed as padding.

Our payload so far is: `%4919c%7$nAAAA\x00\x00\x00\x00\x00\x74\x10\x60`

However, now the seventh argument is no longer the correct address, but rather the substring `$nAAAA\x00\x00`. To fix this, we specify the address to be the eight argument instead and add 2 more "A"s to align it, resulting in the final payload: `%4919c%8$nAAAAAA\x00\x00\x00\x00\x00\x74\x10\x60`.

This causes the address (`0x601074`) to be the third word on the stack, which corresponds to the eight argument:



```
[-----------------------------------stack-------------------------------------]
0000| 0x7fffffffdd50 ("%4919c%8$nAAAAAAt\020`")
0008| 0x7fffffffdd58 ("$nAAAAAAt\020`")
0016| 0x7fffffffdd60 --> 0x601074 --> 0x48fbc8fd
```

3 Return-oriented Programming

3.1 Creating a bufffer overflow

The variables `i` and `fsize` are compared on line 21 to determine how many bytes are going to be read from the `exploit` file. To create a buffer overflow we just need to cause `read_size = fsize`, which happens when `(size_t) i < (size_t) fsize` evaluates to false.

`i` must be at most 24 for execution to even reach this point, but it is not an unsigned variable, so it can be negative causing the type casting to `size_t` (which is unsigned) to transform it to a very large number. If `i` is -1, the conversion will transform it to the maximum value of a long and so it must be the case that `(size_t) i > (size_t) fsize`.

3.2 Determining arguments to the functions

The `read` function needs the file descriptor returned by open. Since file descriptors 0, 1 and 2 are standard input, standard output and standard error, and the `exploit` file should already be closed at the time we call `open`, the file descriptor returned must always be 3, as they are allocated in sequence.

The buffer for the file data can be any writable memory location so we opted for a segment in `libc` that spans addresses `0x00007ffff7dd1000` to `0x00007ffff7dd7000` which is 24576 bytes long, and was identified using the command `vmmap` in `gdb`.

The filename can be stored at the beginning of the buffer `buf`, which we identified in `gdb` as `0x7fffffffddc0`. This will be hard-coded as ASLR is supposed to be off. We considered the possibility of using an environment variable, but in that case the address would change in each new terminal session.

3.3 Passing arguments to the functions

According to the observations above, we would like our gadgets to set up the following sequence of function calls:

1. `open(filename, 0)`
2. `read(3, 0x00007ffff7dd1000, 24576)`
3. `write(1, 0x00007ffff7dd1000, 24576)`

For this, we will need to set two or three arguments per call. Those arguments must go into registers `rdi`, `rsi`, `rdx` in this order.

To be able to populate the registers with the values we intended, the following gadgets were identified, using the `asmsearch` command on the code and `libc` sections of memory:

- `0x00000000004008c3 :  (5fc3) pop rdi; ret`
- `0x00007ffff7a2d2e8 :  (5ec3) pop rsi; ret`
- `0x00007ffff7a0eb92 :  (5ac3) pop rdx; ret`

To find the addresses of the needed functions we used the `p <funcname>` in `gdb`:

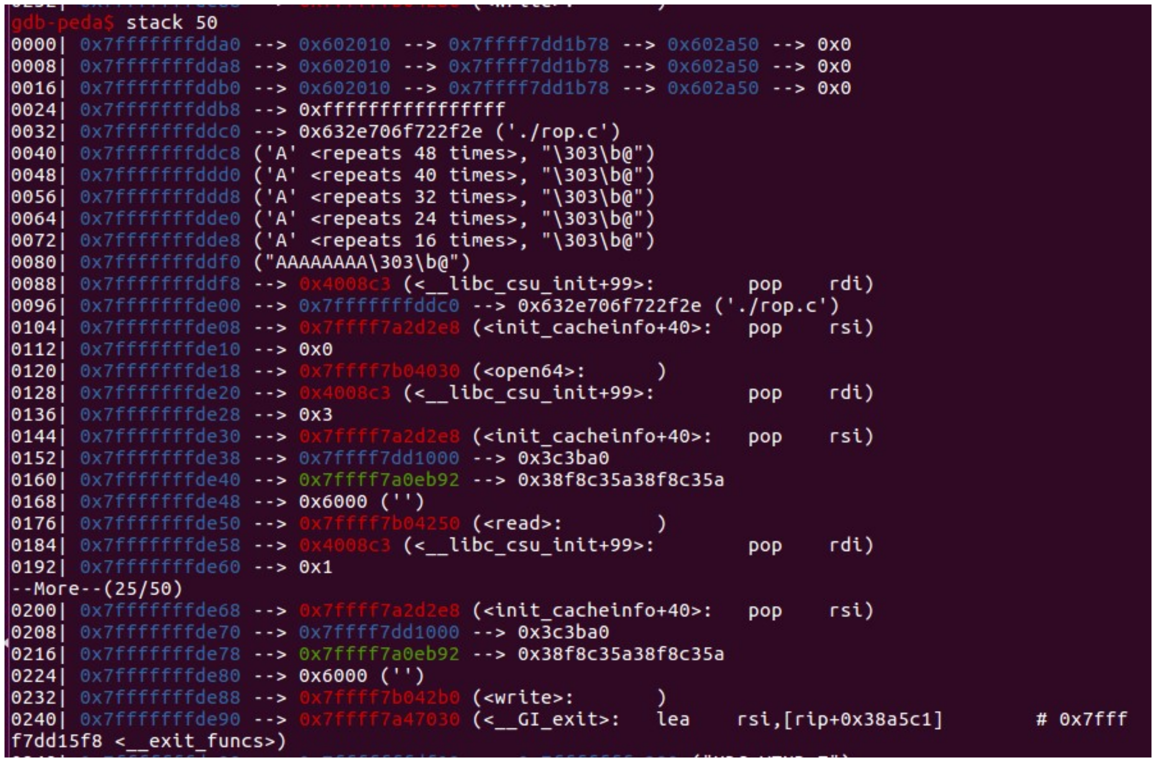
- `open:  0x7ffff7b04030`
- `read:  0x7ffff7b04250`
- `write:  0x7ffff7b042b0`

3.4 Locating the return address

By inspecting the stack inside `gdb` we noted that the return address is 56 bytes after the beginning of the buffer we are overflowing. This means that our choice to write the filename on it will limit us to filenames of up to 55 characters long (+1 byte for null termination), but we consider this to be a reasonable limitation for this exercise. Bytes not used will be populated with “A”s.

3.5 Building the payload

To execute our attack we overwrite the return address with the address of the first gadget we wish to execute. For the gadgets that pop values from the stack, we include the value to be popped right “below” (higher address as stack is descending) the gadget’s address. Chaining all our gadgets together we end up with the following state in the stack just before the `rop` function returns (the `exit` function was added at the end for “graceful” termination):



```
gdb-peda$ stack 50
0000| 0x7fffffffdda0 --> 0x602010 --> 0x7ffff7dd1b78 --> 0x602a50 --> 0x0
0008| 0x7fffffffdda8 --> 0x602010 --> 0x7ffff7dd1b78 --> 0x602a50 --> 0x0
0016| 0x7fffffffddb0 --> 0x602010 --> 0x7ffff7dd1b78 --> 0x602a50 --> 0x0
0024| 0x7fffffffddb8 --> 0xffffffffffffffff
0032| 0x7fffffffddc0 --> 0x632e706f722f2e ('./rop.c')
0040| 0x7fffffffddc8 ('A' <repeats 48 times>, "\303\b@")
0048| 0x7fffffffddd0 ('A' <repeats 40 times>, "\303\b@")
0056| 0x7fffffffddd8 ('A' <repeats 32 times>, "\303\b@")
0064| 0x7fffffffdde0 ('A' <repeats 24 times>, "\303\b@")
0072| 0x7fffffffdde8 ('A' <repeats 16 times>, "\303\b@")
0080| 0x7fffffffddf0 ("AAAAAAAA\303\b@")
0088| 0x7fffffffddf8 --> 0x4008c3 (<__libc_csu_init+99>:        pop    rdi)
0096| 0x7fffffffde00 --> 0x7fffffffddc0 --> 0x632e706f722f2e ('./rop.c')
0104| 0x7fffffffde08 --> 0x7ffff7a2d2e8 (<init_cacheinfo+40>:   pop    rsi)
0112| 0x7fffffffde10 --> 0x0
0120| 0x7fffffffde18 --> 0x7ffff7b04030 (<open64>:       )
0128| 0x7fffffffde20 --> 0x4008c3 (<__libc_csu_init+99>:        pop    rdi)
0136| 0x7fffffffde28 --> 0x3
0144| 0x7fffffffde30 --> 0x7ffff7a2d2e8 (<init_cacheinfo+40>:   pop    rsi)
0152| 0x7fffffffde38 --> 0x7ffff7dd1000 --> 0x3c3ba0
0160| 0x7fffffffde40 --> 0x7ffff7a0eb92 --> 0x38f8c35a38f8c35a
0168| 0x7fffffffde48 --> 0x6000 ('')
0176| 0x7fffffffde50 --> 0x7ffff7b04250 (<read>:         )
0184| 0x7fffffffde58 --> 0x4008c3 (<__libc_csu_init+99>:        pop    rdi)
0192| 0x7fffffffde60 --> 0x1
--More--(25/50)
0200| 0x7fffffffde68 --> 0x7ffff7a2d2e8 (<init_cacheinfo+40>:   pop    rsi)
0208| 0x7fffffffde70 --> 0x7ffff7dd1000 --> 0x3c3ba0
0216| 0x7fffffffde78 --> 0x7ffff7a0eb92 --> 0x38f8c35a38f8c35a
0224| 0x7fffffffde80 --> 0x6000 ('')
0232| 0x7fffffffde88 --> 0x7ffff7b042b0 (<write>:        )
0240| 0x7fffffffde90 --> 0x7ffff7a47030 (<__GI_exit>:   lea    rsi,[rip+0x38a5c1]        # 0x7fff
f7dd15f8 <__exit_funcs>)
```

3.6 File length limitation

This exploit can only print complete files if their size is up to 24576 bytes due to the size of the buffer chosen. For larger files, one could use the buffer iteratively and print one chunk of up to 24576 bytes at a time. For the purpose of this exercise we considered this a reasonable limitation and did not attempt to implement such a workaround.