Buffer Lab

Dongsu Zhang, CSE, 2015-11763

1. Buffer Lab and Overall Stack Image (except Nitroglycerin)

The main idea of the whole buffer overflow is to change the return address and return to another memory of the code, so that the instruction pointer (`%eip`) points to the other instruction when returning.

exploit line	stack	(pointing) register after return
15	argument	<--- `%esp`
14	return address	
13	old `%ebp`	
12	old `%ebx`	
11	Buffer	
:	(44 bytes)	
1	Buffer	

The exploit line is the line that will be written to (exploit) text. For reading convenience, each line has 4 bytes. When we write from line 1 to 11, the buffer will not overflow. However, when we write beyond that (from line 12 to 15), the buffer overflow happens.

The above table can be created, by disassembling the object file with `objdump -d buffbomb` command and searching for `getbuf` function inside. The dumped file in this report will be named `disas`

2. Implementation

2.0 Smoke

The objective of smoke is to call `smoke` function. To call smoke function, the return statement must redirect to `smoke` instead of going back to `test` . We do this by buffer overflowing the return address, which is equivalent as making the exploit line 14 the address of the symbol `smoke` . The detailed objectives are stated below.

1. Find the address of the symbol `smoke`

Go to `disas` and search `smoke` symbol. The address on the left side is the address of the symbol, which in my case is 0x0848be8.

2. Create exploit text

   Write whatever text until line 13. Then on the 14'th line, the text should be 0848be8 in little endian, which is `e8 8b 48 08`.

2.1 Sparkler

The objective of sparkler is to call function `fizz` and set the `val1` and `val2` so that `((~val1 << 8) & cookie) == val2`. Since `cookie` is 8 bytes, it is obvious that if we set the `val1` to 0xFFFFFFFF and `val2` to 0x0, the equation holds.

Calling `fizz` is done same as in 2.0 smoke. The address of symbol `fizz` is at 0x08048c12.

The other objective is to set `val1` and `val2` some other value. If we look at `fizz` part in `disas`, we can assume that `%eax` corresponds to `val1` and `%edx` corresponds to `val2`. Since the `%esp` is at line 15 after return from `getbuf` (look at table above), and `%ebp` is pushed down once after `fizz` is called, `%ebp` is at line 14. We can also check that `%eax` is `0x8(%ebp)` and `%edx` is `0xc(%ebp)`, which indicates line 16 and 17 . Summing up, `val1` and `val2` should be located at line 16 and 17. Placing 0xFFFFFFFF and 0x0 at line 16 and 17 finishes the job.

2.2 Firecracker

The objective of firecracker is to 1. change the global value `global_value` to `coookie & 0x0F0F` (which for me is 0x0407) and 2. call function `bang` afterwards. All of these cannot be done without putting exploited machine instructions.

We can put exploited machine instruction by injecting it in the stack and changing the return value to the stack that has machine instruction. Since the assembly `ret` is just changing the `%eip` register to the return value and popping the stack, it does not matter if the machine instruction is in the .text section or the stack section(?????????????) as long as the machine instruction is valid.

Changing machine instruction consists of 2 steps. 1. Changing the return value to address in stack where the machine instruction is held, 2. Injecting the machine instruction sequentially and returning to desired function (`bang`).

1. Changing the return value

   By running `gdb` and taking breakpoint at break with command `break getbuf`, we can figure out the stack's address. After running, by figuring out the location of the `%ebp`, we know that the line 13 corresponds to 0x55683410 and therefore line 1, where the return address should point corresponds to 0x556834e0 (by subtracting 48=0x30).

2. Injecting the machine instruction

By looking at `bang` section of the `disas` , we are able to find where the the `global_variable` is. It is located at 0x804d120. The `bang` symbol can also be found at 0x08048c81. So we push it to the stack and return to that address.

By assembling the following code to objective file and dumping it with `objdump -d`, we are able to find out the machine code and put it at first of the `exploit`. The machine code and assembly code is following.

```
    0:   c7 05 20 d1 04 08 07    movl   $0x407,0x804d120
    7:   04 00 00
    a:   68 81 8c 04 08          push   $0x8048c81
    f:   c3                      ret
```

2.3 Dynamite

The objective of dynamite is to change the local variable of test `val` and go back to test. To achieve this, we should 1) get machine code onto the stack, 2) set the return pointer to the start of this code, and 3) undo any corruptions made to the state.

Since 1), 2) were dealt in firecracker, same method is used likewise. However, the only difference in executing machine code is that the target variable we are about to change is local variable of `test`. So to figure out the address of the local variable, we must figure out it's address in stack. This can be done using `gdb`. Putting breakpoint on `getbuf` and running command `disas` we know that the `(%ebx)` has the location of the val. Since `%ebx` is `0x8(%ebp)` and `%ebp` is 0x55683410 and the value of stack at 0x55683418 is 0x55683434, the local variable `val` is at stack 0x55683434.

Like firecracker, we move and push (the next instruction of `getbuf` at `test`, which is 0x8048e65) the next instruction and return. The assembly code and machine code is following.

```
    0:   c7 05 34 34 68 55 67    movl   $0x26f00467,0x55683434
    7:   04 f0 26
    a:   ff 35 65 8e 04 08       pushl  0x8048e65
   10:   c3                      ret
```

(The pushl instruction was compiled on my compiler????, but I used push instruction instead, due to malfunctioning, which machine code is 68)

To do 3), undoing the corruptions, we must insert the old `%ebx` and old `%ebp` when the pop instruction is called. So, we figure out this by running `gdb` and running `x/2x $esp` command (breakpoint at `getbuf`) which is command for watching the value of the stack. The value of old `%ebx` and `%ebp` was 0x0 and 0x55683440.

2.4 Nitroglycerin

The objective of this project is same as dynamite, but with buffer size of 512 and using random stack address. Again, like before, we must 1) get machine code onto the stack, 2) set the return pointer to the start of this code, and 3) undo any corruptions made to the state.

Since 1) and 3) are correlated, they will be explained together. The machine code to be injected must (1) move cookie to the `%eax` (unlike before, since `getbufn` returns integer), (2) return to `testn` code after calling `getbufn` which address is 0x8048e65 (check by looking at `testn` in `disas`) , and lastly (3) fix the corrupt state. In dynamite (3) was done in exploit code, but this cannot be done like before since the address of stack is not fixed. Instead, since the relative address of the old `'%ebp` is relatively fixed compare to `%esp`, by looking at `testn` at `diasa` (by 40), we can inject this code to the machine code. Therefore, the machine code looks as following.

```
0:   8d 6c 24 28             lea    0x28(%esp),%ebp
4:   b8 67 04 f0 26          mov    $0x26f00467,%eax
9:   68 65 8e 04 08          push  0x8048e65
f:   c3                      ret
```

For 2), the buffer size is 512, by looking at `disas` and looking at the address of `$esp`, the return address is from the 525th byte to 528th byte. Since only 15 bytes of code is machine code. The tricky part is to estimating where the return address should point to. Since the value of `%ebp` can be relatively +-240, and the space we need below the `%ebp` is 11 bytes (machine code is 15 bytes and `old %ebp` takes up 4 bytes of 15-4=11), if we have 256 bytes below our sampled `%ebp`, it is ensured that our machine code will be operated (256 > 240 + 11). The below image will explain aid understanding.

```
------------------ <--- maximum %ebp (0x55684310 + 240)
       240
------------------ <--- sampled %ebp (0x55683410)
       240
------------------ <--- minimum %ebp (0x55683410 - 240 = 0x55683320)
------------------ <--- machine code of minimum %ebp (0x55683320 - 11)
------------------ <--- my ensuring range (0x55683320 - 16 = 0x55683300)
distance from maximum %ebp and my ensuring range = 480 + 16 = 496 < 520, which makes sure that
my ensuring range operates at any given range.
```

Briefly, the return address should be 0x5568330.

3. Conclusion

The buffer lab was very difficult for me, since this was my first time using the `gdb`, and the `gdb` did not execute as I expected. I'm still not sure why it does not work with raw input. I'm still curious why the `si` command for single instruction does not work. However, doing this lab, I completely understood stack and that instruction is nothing but long bytes, nothing fancy.