Task1:

Text

Description automatically generated

Here we tur off address space randomization bu setting it to “0” so that we are able to guess the exact address of the code

And then we set change the shell from dash to zsh because dash/bash shell has a counter measure that drops privileges and doesn’t allow a user to get root privileges when the vulnerability is executed

Graphical user interface, text, application

Description automatically generated

Here we use make command to compile the already given call\_shellcode.c program in both 32 bit and 64 bit. The make command uses makefile script to compile the c program

Graphical user interface, text, application

Description automatically generated

Here we can see that we are able to invoke the shell when we execute the program and also see that he shell is invoked on seed user and not on root user

Graphical user interface, text, application

Description automatically generated

Here we can see that we are able to invoke the shell when we execute the program and also see that he shell is invoked on seed user and not on root user

Task2:

Makefile for task 2

Text

Description automatically generated

Here we edit the L!,L2,L3,L4 of the makefile script as mention by the professor

This is the stack.c vulnerable program

Text

Description automatically generated

Text, letter

Description automatically generated

To compile the stack.c program we use the already available makefile and use make command to compile the program. The make command compiles the program at L1,L2,L3,L4 buffer size and sets al the programs to setuid

A picture containing text

Description automatically generated

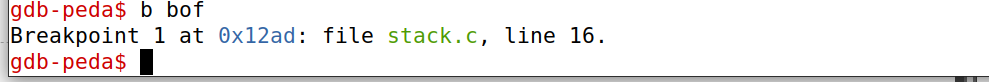
Touch command is used to create a badfile

Using gdb to access stack-l1-dgb in debugger mode

Text

Description automatically generated

Set a break point at bof function



Using run command to execute the program until we reach the break point

Text

Description automatically generated

Text

Description automatically generated with low confidence

After we reach the break point we use the next command to reach the next point where vulnarability exists that is the strcpy command

Text

Description automatically generated

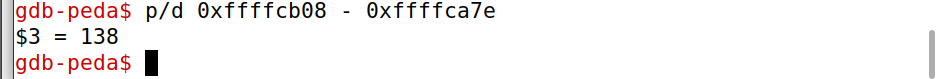
Table

Description automatically generated

Graphical user interface, application

Description automatically generated

In this part we print out the assembly level codes for the break pointer what is pointing to the buffer and the address of the buffer



Next we do ebp -buffer to get the offset value which is further incremented by 4 because it’s a 32 bit program

The ret value is going to be the buffer address + any random number >250 because it has to be something that overflows the available buffer and should not end with a double zero in the hexdump when the exploit is created in the badfile

The start is going to be 517 – len(shellcode) which is already given to us

Contents of exploit.py after updating the start, ret, and offset

Graphical user interface, text, application

Description automatically generated

Text, table

Description automatically generated with medium confidence

Once we execute the exploit and a badfile is updated with the payload which is sent into the buffer overflow vulnerable program and execute it we can see that we are able to get seed shell

Task4:

Text

Description automatically generated

Table

Description automatically generated with medium confidence

Graphical user interface, text, application

Description automatically generated

Doing the same as task3 but for stack-l2-dgb

The start is going to be the same value but ret and offset change based on the new address found

We further increment the offset value by another 200 for this program

Contents of exploit.py after updating the code for L2

Text

Description automatically generated

Updated exploit.py for level 2

Table

Description automatically generated

Here we execute the exploit again to make another badfile that is compatible for the stack-L2 and the execute the stack-L2. Then we are able to gain access to the seed shell using the buffer overflow exploit that exists in the stack.c program at L = 150

Task5:

Text

Description automatically generated

Opening stack-l3 using gdb debugger

Text

Description automatically generated with medium confidence

Running all the commands and getting the required address that will be used in the exploit

A picture containing diagram

Description automatically generated

In this we take the rbp value and buffer address values and also calculate the offset number which is rbp-buffer address that is 176

Graphical user interface, text, application

Description automatically generated

The exploit.py program is updated with the results we got in the debugger and the address are used in ret, offset. The offset is also incremented by 8 as this is a 64 bit and the shell code is also updated to 64 but shell code (already given in the pdf and task 1) and the value of L is also change to 8 as this is running for a 64 bit

Once all the values are updated exploit.py is executed to create the bad file

Text

Description automatically generated

When we execute the vulnerable program using the payload(badfile) created by the exploit and we are able to get seed shell after the vulnerability is exploited causing buffer overflow

Task6:

A picture containing text

Description automatically generated

We do the all the same thing we have done in the previous task and get the addresses for rbp and buffer and calculate the offset

Graphical user interface, text, application

Description automatically generated

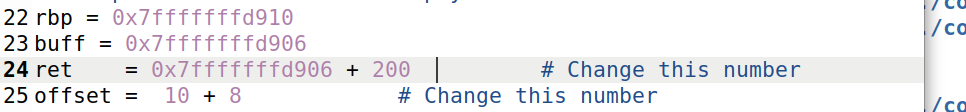
The exploit.py has been updated according to the values that we have calculated and above image and update ret , offset which ius incrfemented by 8 as this if for a 64-bit and L = 8 the same as before.

The ret is increased by 1600 as the given buffer is too small and smaller values give out segmentation faults

Text

Description automatically generated

Once the vulnerable program is executed using the badfile created by the exploit, a buffer overflow occurs and we are able to gain access to the sed shell.



Graphical user interface

Description automatically generated with low confidence

Here we can see that when that when smaller buffer is used we get segmentation fault error

Task7:



using Commented setuid(0) binary,i.e. the setuid binary code is comment out in the code

Graphical user interface, text, application

Description automatically generated

We observe that we are able to again access to the shell but its not root shell as we are not able to bypass the counter measures set by dash shell

using setudi(0) binary which added to the top of shellcode

Text, letter

Description automatically generated

Graphical user interface, text

Description automatically generated

Text

Description automatically generated

Once we put the dash counter measure code in the program and execute it we are able to get the shell with root privileges when the program is made a setuid program and executed

Text

Description automatically generated

Graphical user interface, text, application

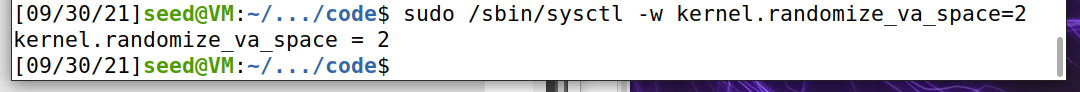
Description automatically generated

Text

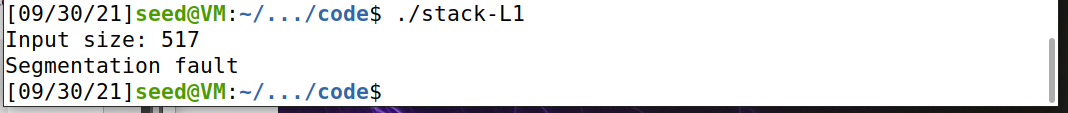
Description automatically generated with medium confidence

Using the dash countermeasure binary code we are able to get root shell while not using the counter measure gives us only seed shell as the privileges are dropped on execution when the vulnerability is sensed

Task8:



Turning on address randomization using the above command



Once the address randomization is turned back on the code stop working and gives out a segmentation fault error

A picture containing shape

Description automatically generated

The bruteforce.sh program that is used on stackL1 so that it can run multiple times hoping that the address in the exploit program are correct at some point and we get a root shell

Application

Description automatically generated with low confidence

Using the brute-force method we were able to gain access to seed shell in 1minute 51 seconds and it took 105460 attempts to gain access to the shell. Once we have gained access to the shell the brute-force loop stops

Task9:

Task9a:



Address randomization set to off

Text, letter

Description automatically generated

Exploit still working before stack guard

Text

Description automatically generated with medium confidence

Compiling stack.c without -fno-stack-protector

Graphical user interface, text

Description automatically generated

We observe that the stack guard that is enabled by default, doesn’t allow any buffer overflow, but instead it terminates the execution as soon as the vulnerability comes into play

Task9b

Graphical user interface, text, application

Description automatically generated

Compiling the program without “-z execstack” and executing it

We observe that when we try to exploit the vulnerability we get error saying segmentation fault, this is because of non-executable stack countermeasure