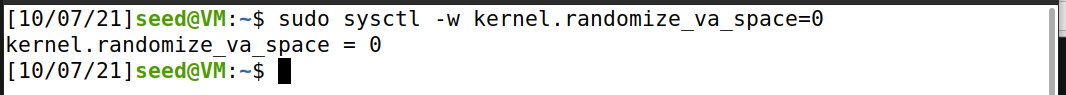
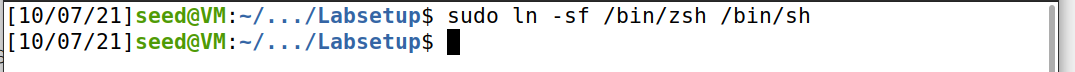
Task1:



Turning off address space randomization



Linking /bin/sh to another shell that does not have such a countermeasure which drop privileges on execution

Graphical user interface, text, Word

Description automatically generated

We use the make command to compile the retlib program that is already available to us. The make command uses Makefile which is also already given to us

Text

Description automatically generated

We then create a badfile in which the payload for buffer overflow will be created in.

And then we use gdb on retlib to get the required adressess

Text

Description automatically generated

Graphical user interface, text, application

Description automatically generated

After breaking at main and running the program we print out the address of “system” and “exit” and then save these address at another location which will be later used in exploit

Task2:

A picture containing graphical user interface

Description automatically generated

In this task we use export command to create an environment variable that points to shell location in the directory

And check if the variable is exported into the environment variables using the env command

Graphical user interface, text, application

Description automatically generated

The above is the code for getting the address of myshell environment variable that we created that is on stack

Text

Description automatically generated

Then we compile the program using gcc and execute it to get the address of the variable.

When the code is integrated into retlib.c

Graphical user interface, text, application

Description automatically generated

Text

Description automatically generated

We are able to see the same address for the myshell environment variable we exported

Task3:

A picture containing text

Description automatically generated

using frame pointer as ebp and buffer address we calculate the value for system which will be ebp - buffer



Using gdb we calculate the ebp - buffer and get 132 for which we add 4 and as it is 32 bit and use it as value for system that is Y in exploit.py

Graphical user interface, text, application

Description automatically generated

And then we get the value of z which is +4 of y and then x = z+4. And we also use the address we saved before here in the exploit code. The system address, exit address and the shell address.

Text

Description automatically generated

After creating the badfile using the explot.py we are able to exploit the buffer overflow vulnerability that exits in the vulnerable program and are able to invoke root shell

Attack v1:

Graphical user interface, text

Description automatically generatedText

Description automatically generated

We are still able to invoke root shell after commenting out the exit part of the program but we segmentation fault when we exit the shell as the exit address is missing

Attack v2:

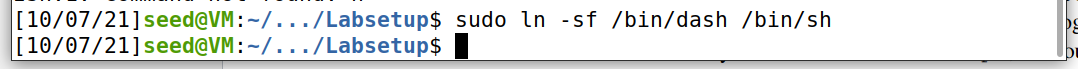
Text

Description automatically generated

From the observation our attack is not successful after changing the name for the program to newretlib as the address of the “myshell” changes with the number on characters in the program name.

And we also get an error saying zsh:1: command not found: h

Task4:



Linking the shell to dash shell

Graphical user interface, text, application

Description automatically generated

Using the same method as before we the addresses of system, exit and execv in this task and save them for later use

Text

Description automatically generated

For the exploit in this task we change the system address to “execv” address that we have found using gdb and also use address if input inside main that we get from “./retlib” which is used as buffer for this program and we get ar value by using trial and error method until we get the output and have found that ar accepts the values which are greater then 145

Here we also write the bit level code for execv arguments that are “/bin/sh -p” and null and add them into the content

Text

Description automatically generated

here we are able to create the badfile using the updated exploit and execute retlib to invoke root shell by exploiting buffer overflow vulnerability that exists in the program

Task5:

Text

Description automatically generated

For this task we find the address of foo in the retlib program using gdb like we have done in previous tasks and update the exploit.py program with address of foo.

And the value of foo uses the offset that is 136 and since the program has to run for 10 times before it can give us a root shell the final pointer is increased by 40 as each iteration takes 4 bits. Meaning the initial pointer points at offset and the final one points to offset + 40 which calls the function foo 10 times and the foo address is multiplied by 10 for it to run 10 times

And the values of X,Y and Z are also simplified using offset and are also increase by 40 with respect to their values.

Text

Description automatically generated

Once the exploit is executed and the badfile is created, retlib is executed and we are able to invoke function foo() 10 times before gaining access to root shell when the vulnerability is exploted.