Secure Systems Development

Stack smashing exploit is an attack that causes a stack buffer overflow. It is used to change the next program the system is going to execute or crash the system by writing more data to a buffer than it is capable of holding and overwriting return address of the current program.

Text

Description automatically generatedText

Description automatically generated

This is our example code that is vulnerable to stack smashing attack. Line 45 is where the vulnerability lies. The function strcpy copies the contents of string input provided to the outer function into the buffer. However, even though the buffer has a length of 80 bytes, strcpy will keep writing into the buffer if the string is long enough, eventually causing buffer overflow. When the program is executed, the buffer’s memory is filled with the string in file “hello”. The contents in buffer are then printed on the terminal. Afterwards, the function called func puts these contents into a buffer that is 80 bytes. Finally, the buffer is freed, and the program terminates.

There are three protection measures that stop our attack with the given example code. First of these is the stack protector. Stack protector itself is a stack that is placed before the return address. The same idea as checksum, if the stack protector is overwritten and changed, then it means the return address might be in danger. If that is the case, then the return address is not used and program terminates. To disable stack protector, we use “-fno-stack-protector” while compiling our vulnerable code. The next protection is non-executable stacks. This is a virtual protection to allow only writing on the stack without execution. Therefore, even if the return address is overwritten, the shell code cannot be injected. To disable this protection, we use “-zexecstack” while compiling our vulnerable code. The last protection is an operating system implementation instead of c language. Address Space Layout Randomization (ASLR) is a technique that increases the difficulty of performing a buffer overflow attack that requires knowing the location of an executable in memory by randomizing the memory location when each time the code is executed. We disable this in our virtual machines by executing the code with “env – setarch -R ./vulnerable” command. Also, we need some debugging information to carry out our attack. The python script that will create our shell code needs two addresses. The first one is the return address of the vulnerable code and the second one is the location of the buffer. To get these numbers, we need to start GBD and write “attach processID” to see the vulnerable code execution. Hence, “processID” is the process identification number of vulnerable code’s process. This is found by running “ps aux | grep vulnerable” on a different terminal. Once that is done, a breakpoint needs to be created before the strcpy and we do that with “b code.c:45” in GDB. Afterwards, “info frame” will give the information we are looking for. “rip at <address>” is the return address we are looking for. “p &buffer” will give us the second information, the memory location of buffer.

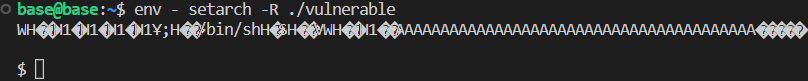


Buffer AddressText

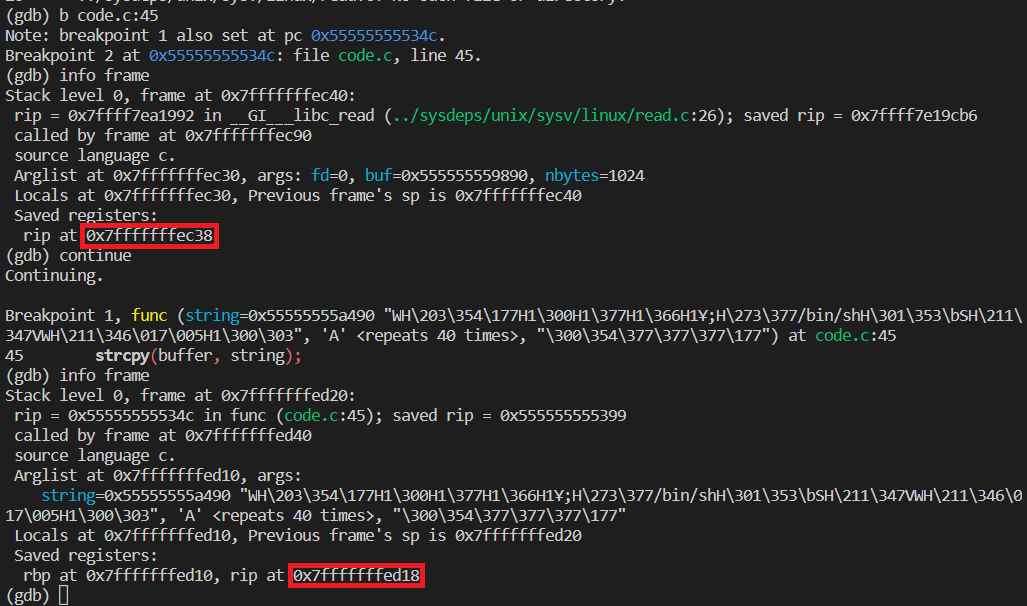
Description automatically generated

Return Address

The python code takes these numbers as an input and a string to write the shell code with that name. The shellcode we need to use is formed in the python script. The return address point and the buffer address point are validated in the python code and then used to calculate the space left in the stack for the created shell code. The python code will create a file with the shell code named as the third parameter, in our case it is called “hello”. Once the vulnerable code is executed again, now it will read the “hello” file as it is hard coded and will fill the buffer with our shell code. The string in “hello” will be printed on the terminal and after hitting the enter once, the shell terminal will open.



Exploit running the shell



The return address will be changed as can be seen in the figure.