**TASK1**

A screenshot of a computer

Description automatically generated with medium confidence

**TASK 3**

Address of system() = 0xb7e42da0

Address of exit() = 0xb7e369d0

At first, we used trial and error to see that return pointer on the stack was starting at 32, we also confirmed this by gdb, because before 32 the return pointer was same but after 32 the return pointer began to get overwrite partially with ‘aa’ and then fully at 36. Then we found the system() and exit() address which came out to be same as given in the manual. Now since, the parameters are pushed first, then the return address and then the function address therefore, we first put system() address in place of return address (32-35). Then we put exit() address in place of return of system function (36-39). Then we put /bin/sh address at the place of parameters of system function (40-43).

After making the environment variable, we ran the program to find out its address. The address that came out to be off by 2 as it was giving only “in/sh” in parameters of system() therefore we tweaked it a bit by subtracting 2 from the address after which we got the whole /bin/sh and hence, root shell opened.

A screenshot of a computer

Description automatically generated with medium confidence

**TASK 4**

The exit function is not necessary, the attack works even without the exit function because the address of exit function will be used when the system function finishes, and system function ends when the root shell is exited. So, the only difference now is that when the root shell and then system function finishes, the program won’t have any address to return to and will crash.

A screenshot of a computer

Description automatically generated with medium confidence

**TASK 5**

The attack was not successful because system() got the wrong address for MYSHELL variable because the address of variable changed. As written in the manual, this changing of address of environment variable was caused by the changing in the name of the program in terms of the number of characters). Since, newretlib had more characters than retlib, the address of MYSHELL changed and the attack failed.

A screenshot of a computer

Description automatically generated with medium confidence

**TASK 6**

1) Data Execution Prevention (DEP) which is always enabled in 64-bit programs which marked the pages in memory as non-executable. Hence, even if the instruction pointer is pointed to this section of memory, the program would not execute those commands

Another way to secure the system from buffer overflow vulnerability is ASLR is the Address Space Layout Randomization which randomizes the location of the processes. It doesn’t prevent buffer overflow attacks but makes these attacks much harder. Address space in 64-bit architecture is larger where around 48 bits are randomized, which makes it difficult to attack by brute force.

2) A buffer overflow occurs when the data exceeds the memory buffer size. Frameworks such as Vue.js provide strong mechanisms against this vulnerability, explained below:

Stack Canaries: In this canary value is placed at the end of the buffer which is just before the return address. When buffer overflow occurs which replaces the original canary value display the buffer overflow message or terminate the program.

Bound Checking: It this buffer size and the value of the input is checked which means the value of input should not exceed buffer size otherwise display error without saving the value

Tagging: In this same area of memory is marked as non-allocated which means it is not allocated to the buffer. It is used for sensitive data and cannot be affected by the buffer overflow.

3) We can use a system-wide defense where when the function is called, a copy of the return address is stored in the safe area of memory just like the tag area by frameworks. As function return, the return address is compared with the saved value and allow if same otherwise display an error or terminates the program.