**Task 1: Exploiting the Vulnerability**

The vulnerability we are exploiting is the lack of bounds checking in the C function strcpy(), which performs string copy operations. In this lab, we exploit this to get a root shell by stack smashing. However, in order to demonstrate this vulnerability, we have to work around three protections built into Ubuntu 9.11, namely address randomization, the symbolic link to /bin/bash by /bin/sh, and StackGuard.

By default, the starting memory addresses for the stack and heap are randomized, which makes guessing the address more difficult for our buffer overflow attack. /bin/bash drops privileges when it is invoked. We first disable the address randomization through the command “sysctl -w kernel.randomize\_va\_space=0” after logging in as root. We then change the symbolic link of /bin/sh to /bin/zsh to ensure that when our shell program is invoked, privileges are not dropped.

The third protection is StackGuard, which is implemented in GCC by default. StackGuard adds a special memory location, otherwise known as the canary, to our stack. This canary must be verified before any control is passed to an address on the stack. A buffer overflow attack would overwrite the canary, which will alert StackGuard and the system to not trust the stack values, often killing the program. To remove StackGuard, we have to add the argument “-fno-stack-protector” when compiling the vulnerable code with GCC.

The next steps are to compile the provided shellcode and vulnerable stack code. One thing to note about the stack code is that we have to compile it as a root user, so it runs as a root user no matter what user is actually calling the program.

The last step is to complete exploit.c to fill the buffer with the values needed to exploit our stack program’s vulnerability. First, we create a function to get the address on top of the stack:

unsigned long get\_sp(void)

{

\_\_asm\_\_("movl %esp,%eax"); // Get the address at the top of the stack

}

Then, we fill the buffer:

int i = 0;

char \*ptr; // Pointer to buffer

long \*addrptr;

// We use this address to try and get into the exploit and overwrite the return in stack.c

long retaddr;

int num = sizeof(buffer) - (sizeof(shellcode) + 1);

ptr = buffer; // Address at start of the buffer

addrptr = (long\*)(ptr);

retaddr = get\_sp() + 500; // Offset to get into the bof function in stack.c

// Start by putting in the address to get into our exploit

for (i = 0; i < 20; i++) {

\*(addrptr++) = retaddr;

}

// Fill the rest of the buffer with shellcode

for (i = 0; i < sizeof(shellcode); i++) {

buffer[num + i] = shellcode[i];

}

buffer[sizeof(buffer) - 1] = '\0'; // Null character at the end

We end our exploit program by writing the contents of our buffer into badfile.

To execute the attack, we create the badfile by running our exploit and then running our vulnerable stack code.

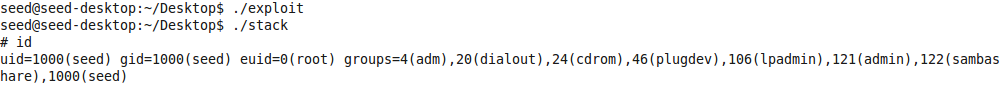


Figure 1: The attack is a success! We have a root shell.

**Task 2: Protection in /bin/bash**

After pointing /bin/sh back to /bin/bash, running the vulnerable program does not open a root shell. Running the id command reveals that we do not get a root shell, as euid=0(root) is not shown. This is because bash drops privileges when it is invoked, so we do not get a root shell.

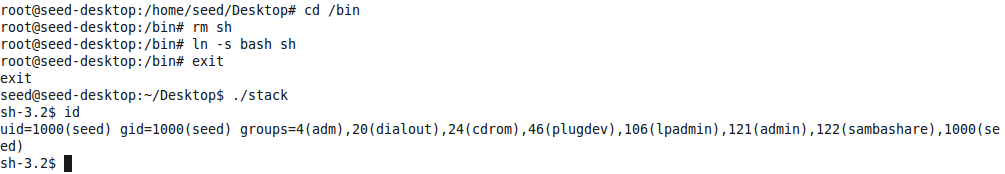


Figure 2: Attempting the attack after pointing /bin/sh back to /bin/bash gives us a non-root shell.

**Task 3: Address Randomization**

We get two different outcomes with re-enabling address randomization. First, if we let /bin/sh point to /bin/bash, then we will get predominantly segmentation faults with the rare non-root shell. This can be attributed to our exploit being unable to find the correct address, with the rare non-root shells coming from any time our exploit “guessed right” on the address. If we let /bin/sh point to /bin/zsh, then the program will segmentation fault as well unless the exploit gets the address right. However, one thing to note is that when trying to run the stack code as a loop with the command, ‘sh -c “while [ 1 ]; do ./stack; done;”,’ with zsh the program would just hang while it would give the expected segmentation fault if run with simply ./stack.

**Task 4: Stack Guard**

To show the effect of enabling StackGuard on our attack, I compiled stack.c without ‘-fno-stack-protector’ and named it ‘safestack.’ When I run this program, Ubuntu’s StackGuard terminates the program, stating that stack smashing was detected. StackGuard adds a canary to the stack layout, which is a special memory location. The buffer overflow attack we perform in this lab will **overwrite** the canary, triggering StackGuard, as the canary is no longer verifiable.

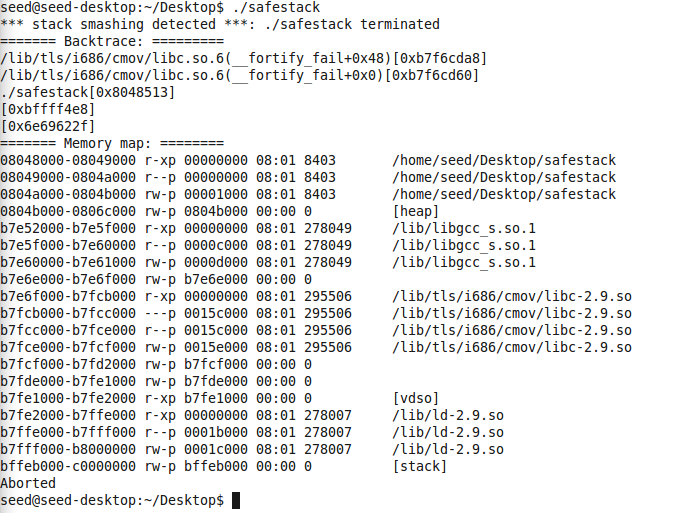


Figure 3: Trying the attack with StackGuard enabled. Notice that the program is killed due to the attack overwriting the canary.