PROGRAMMING AND SECURITY PROJECT



Buffer Overflow with Gets Function

in 64 Bit Linux System

1701240 - BARIS ATAK

1701236 - MUHAMMET MUCAHIT AKTEPE

[Buffer overflow with gets function]

Ubuntu 16.04, 64-Bit Operating System

# Overview

## What is Buffer Overflow? Why it is a problem?

Shortly the buffer overflow is an anomaly, where a program, while writing data to a buffer, overruns the buffer's boundary. Namely buffers are created by fixed size so if we pass more data than the buffer can store, buffer will overflow. When a buffer overflowed, the program can access other parts of memory which belong to other programs. As you think, this may cause really big problems.

Normally we shouldn't have permissions to access other parts of memory but in some cases it may be. Then we would have the vulnerable program and nothing is safe with that program. Here the Buffer Overflow is!

## Where can we face with Buffer Overflow?

We can face with the Buffer Overflow vulnerability in C/C++ technologies because those technologies have no built-in protection against accessing or overwriting data in any part of their memory about buffer limits and includes some vulnerable functions. But buffer overflows also can exist in any programming environment where direct memory manipulation is allowed.

Along with that, if a programmer who programmes with those technologies, uses the vulnerable functions without controlling limits of buffer, there is going to be a big problem for security. As you understand it from the project title, we are going to examine the gets function that is the one of vulnerable functions of C, in this project.

## What is the size of the danger of Buffer Overflow?

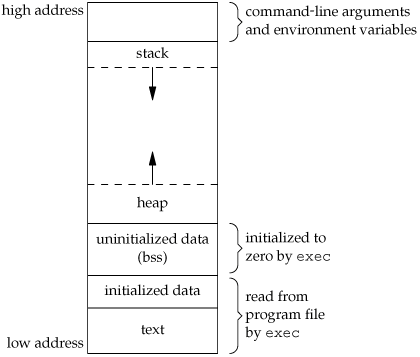
It completely depends on the imagination of the attacker. In fact, the attacker can do anything which wants, such as deleting files, stealing information, using the computer in other attacks.

## Important points to better understand Buffer Overflow

We should know how memory runs for better understanding the Buffer Overflow. That’s why we will dive deeper into memory and some registers.

By the way we are trying to explore buffer overflow in Stack but it may be also in Heap. We will just give some short informations about other parts of memory and then mostly talk about Stack. We can see whole memory structure in next page…

**MEMORY FOR A RUNNING PROCESS** [6]

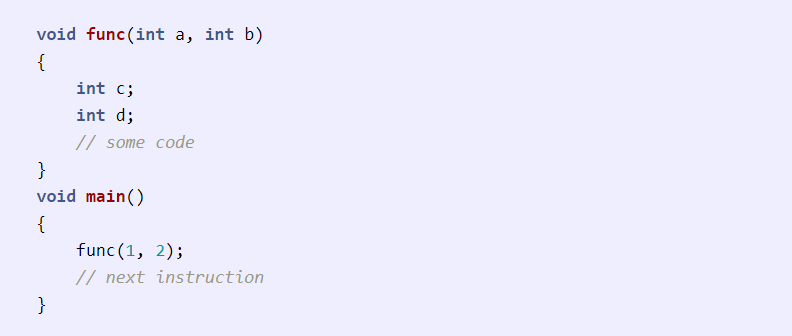
 Source: [*http://i.stack.imgur.com/1Yz9K.gif*](http://i.stack.imgur.com/1Yz9K.gif)

1. **Stack**: This is the place where all the function parameters, return addresses and the local variables of the function are stored. It’s a LIFO structure. It grows downward in memory(from higher address space to lower address space) as new function calls are made. We will examine the stack in more detail later. [6]
2. **Heap**: All the dynamically allocated memory resides here. Whenever we use malloc to get memory dynamically, it is allocated from the heap. The heap grows upwards in memory(from lower to higher memory addresses) as more and more memory is required. [6]
3. **Command line arguments and environment variables**: The arguments passed to a program before running and the environment variables are stored in this section. [6]
4. **Uninitialized data(Bss Segment)**: All the uninitialized data is stored here. This consists of all global and static variables which are not initialized by the programmer. The kernel initializes them to arithmetic 0 by default. [6]
5. **Initialized data(Data Segment)**: All the initialized data is stored here. This constists of all global and static variables which are initialised by the programmer. [6]
6. **Text**: This is the section where the executable code is stored. The loader loads instructions from here and executes them. It is often read only. [6]

**SOME IMPORTANT REGISTERS** [6]

1. **%rip**: The **Instruction pointer register**. It stores the address of the next instruction to be executed. After every instruction execution it’s value is incremented depending upon the size of an instrution. [6]
2. **%rsp**: The **Stack pointer register**. It stores the address of the top of the stack. This is the address of the last element on the stack. The stack grows downward in memory(from higher address values to lower address values). So the %rsp points to the value in stack at the lowest memory address. [6]
3. **%rbp**: The **Base pointer register**. The %rbp register usually set to %rsp at the start of the function. This is done to keep tab of function parameters and local variables. Local variables are accessed by subtracting offsets from %rbp and function parameters are accessed by adding offsets to it as you shall see in the next section. [6]

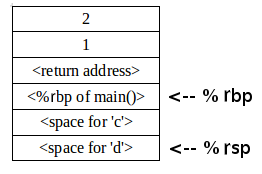
**MEMORY MANAGEMENT DURING FUNCTION CALLS** [6]



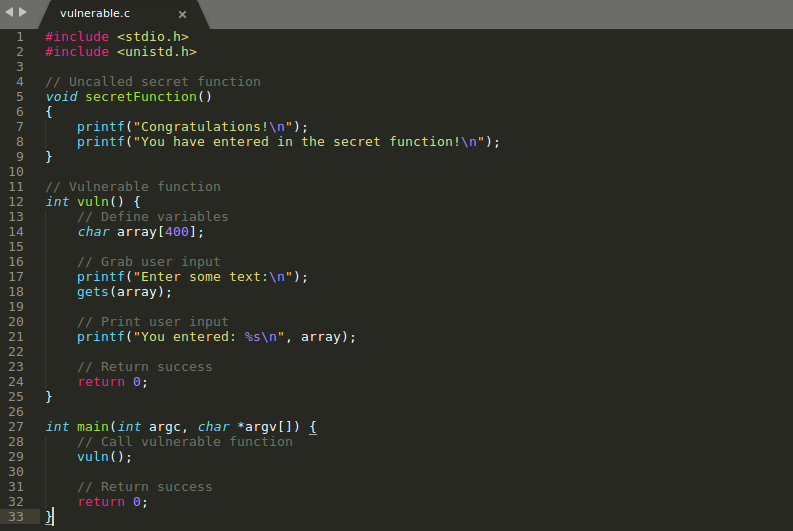
Assume our **%rip** is pointing to the **func** call in **main**. The following steps would be taken: [6]

1. A function call is found, push parameters on the stack from right to left(in reverse order). So **2** will be pushed first and then **1**. [6]
2. We need to know where to return after **func** is completed, so push the address of the next instruction on the stack. [6]
3. Find the address of **func** and set **%rip** to that value. The control has been transferred to **func()**.[6]
4. As we are in a new function we need to update **%rbp**. Before updating we save it on the stack so that we can return later back to **main**. So **%rbp** is pushed on the stack. [6]
5. Set **%rbp** to be equal to **%rsp**. **%rbp** now points to current stack pointer. [6]
6. Push local variables onto the stack/reserver space for them on stack. **%rsp** will be changed in this step. [6]
7. After **func** gets over we need to reset the previous stack frame. So set **%rsp** back to **%rbp**. Then pop the earlier **%rbp** from stack, store it back in **%rbp**. So the base pointer register points back to where it pointed in **main**. [6]
8. Pop the return address from stack and set **%rip** to it. The control flow comes back to **main**, just after the **func** function call. [6]

This is how the stack would look while in **func**. [6]



# EXPLOITING



Here is our code! It includes an uncalled function as you can see. It’s the **secretFunction()**. We will try to execute it even though it’s not called.

----------------------------------------------------------------------------------------------------------------

First of all we have to disable some operating system’s protection mechanishms to be able to exploit. We will talk about these mechanishms later.

----------------------------------------------------------------------------------------------------------------

* **sudo su**
* **echo 0 > /proc/sys/kernel/randomize\_va\_space 🡪 It will disable ASLR** [5]

----------------------------------------------------------------------------------------------------------------

When we compile our code, we should use this:

* **gcc –fno-stack-protector –z execstack –mpreferred-stack-boundary=4 –o bufferoverflow bufferoverflow.c** [5]

to disable stack protection.

Now we compiled our code without protection. Namely we will be able to exploit a buffer overflow vulnerability.

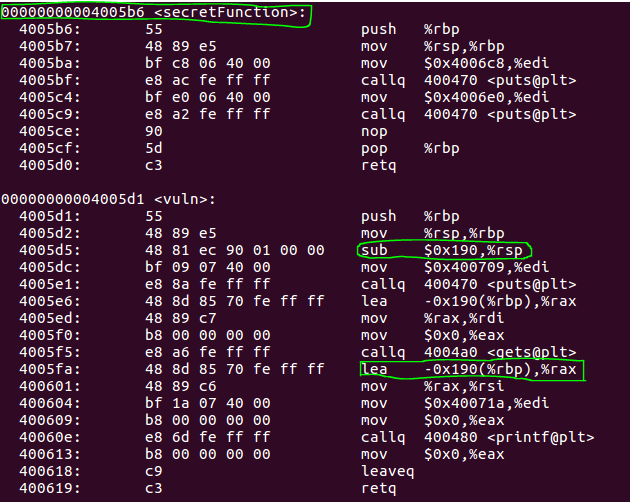
----------------------------------------------------------------------------------------------------------------

----------------------------------------------------------------------------------------------------------------

* **objdump –d vulnerable**

With this command we will get some information about program.

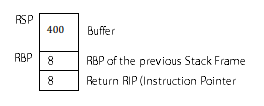
Result of this command:



As you can see above, the **secretFunction’s address** is **00000000004005b6**.

We will overflow our buffer till **Base Pointer** then place this address of **secretFunction** to the **Instruction Pointer** to call **secretFunction**.

**190 in hex or 400 in decimal** bytes are reserved for the local variables of **vuln** function. If you remember we asked also for **400** bytes in our code.



The **400** is our buffer size.

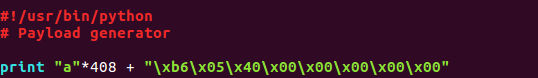
Our registers are 8 byte because our system is 64-bit.

----------------------------------------------------------------------------------------------------------------

----------------------------------------------------------------------------------------------------------------

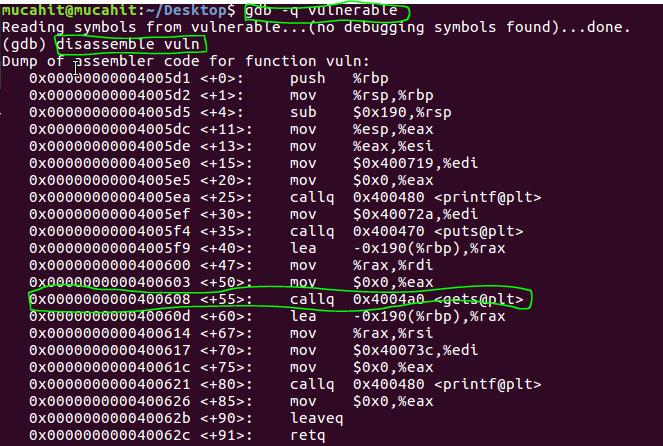
* **nano payloadGenerator.py**

After this command we will write codes below and save/close it.



It’s our payload generator. We write the ‘a’ character 408 times because our buffer size is 400 and also our rbp (base pointer) size is 8. After that we will access to the rip (instruction pointer) and we will write there our secretFunction’s address. Then we will be able to access to it even though it’s not called. We write it to a text that name is textFile.

----------------------------------------------------------------------------------------------------------------

We go into the gdb with debug our code and disassable the vuln function to see when the gets function are being called. As you can see the gets function is called in 55 line.

----------------------------------------------------------------------------------------------------------------

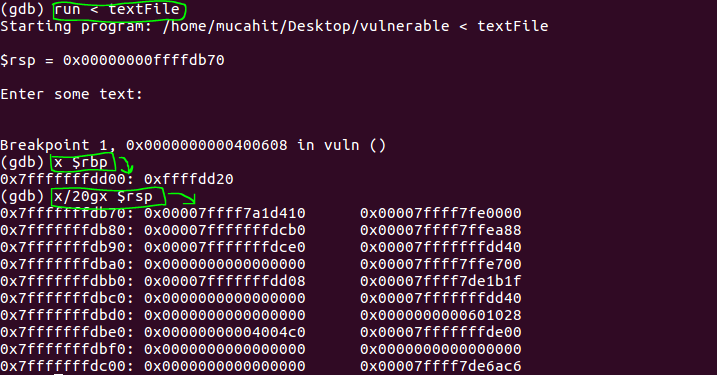
Now we will add the break point on that position and the other position which is after the calling of gets function for example to 60.

----------------------------------------------------------------------------------------------------------------



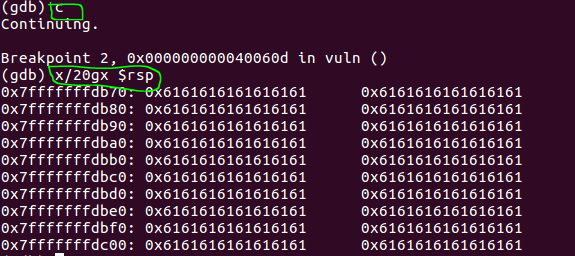
First breakpoint is before the gets function and the second is after the gets function.

----------------------------------------------------------------------------------------------------------------



When we run the code with our pregenerated text file (payload). We will stop at the first break point and we can see our registers status. As you can see now the stack is not filled with ‘a’ values because this time the gets function hasn’t called yet.

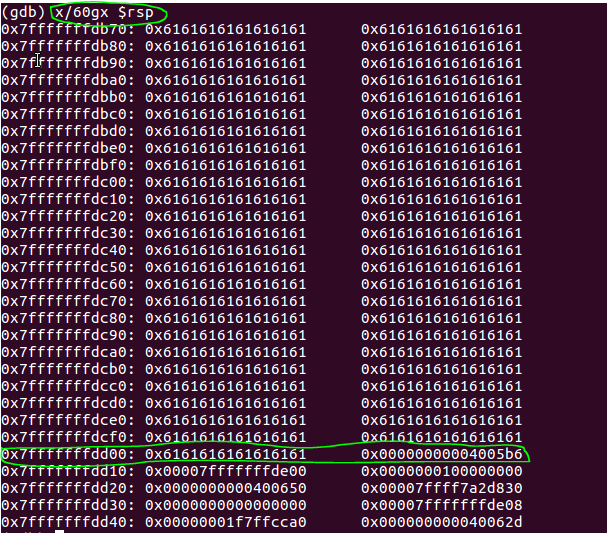
----------------------------------------------------------------------------------------------------------------



When we continue the gets function is called at the second break point and now you can see the hexa values of ‘a’ character in the whole stack.

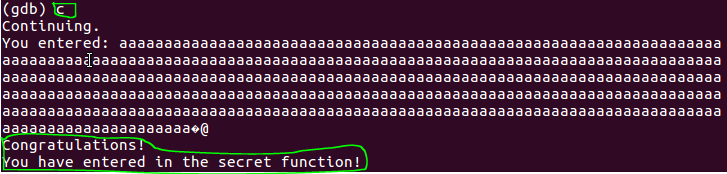
----------------------------------------------------------------------------------------------------------------

----------------------------------------------------------------------------------------------------------------



When we see more area in the stack we can see also our base pointer. As you can see the first 8 bytes of 0x7fffffffdd00 address are belong the rbp address and the second 8 bytes are belong to the instruction pointer. And now you can see the instruction pointer points to the secret function address which we generated before with our payloadGenerator.

----------------------------------------------------------------------------------------------------------------



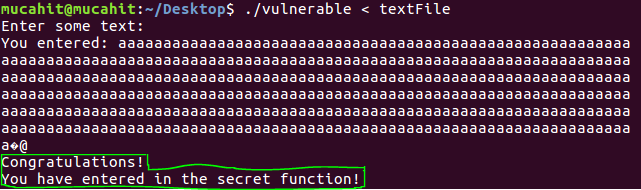
And when we continue we can access to the secret function.

----------------------------------------------------------------------------------------------------------------

----------------------------------------------------------------------------------------------------------------

* **./vulnerable < textFile**

Also we can run the code outside from the gdb. When we run our code with our payload, we faced with the same result:

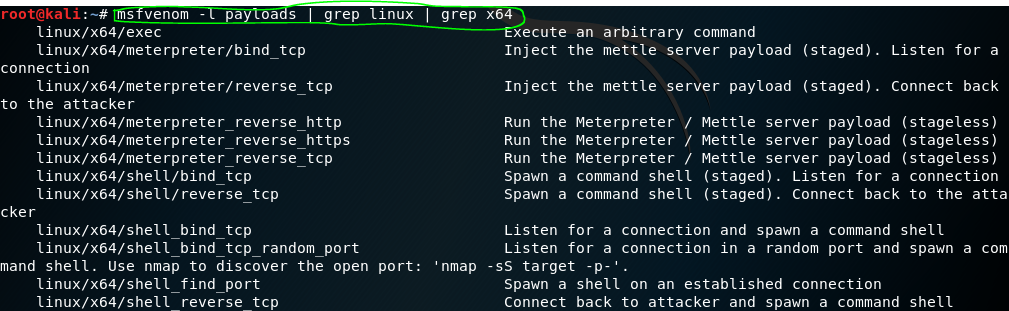


Did you see! We exploited and accessed to the secretFunction.

----------------------------------------------------------------------------------------------------------------

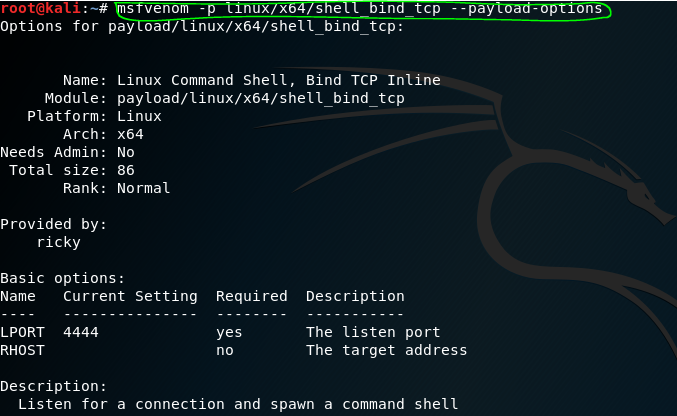
**GENERATING SHELL CODE on KALI LINUX with MSFVENOM**

----------------------------------------------------------------------------------------------------------------



You can choose whatever you want among these types.

----------------------------------------------------------------------------------------------------------------

****

You can see the basic options of shell\_bind\_tcp. You can try with other types like exec, reverse\_tcp and so on.

----------------------------------------------------------------------------------------------------------------

----------------------------------------------------------------------------------------------------------------



Through this comman you can generate shell code in type shell\_bind\_tcp.

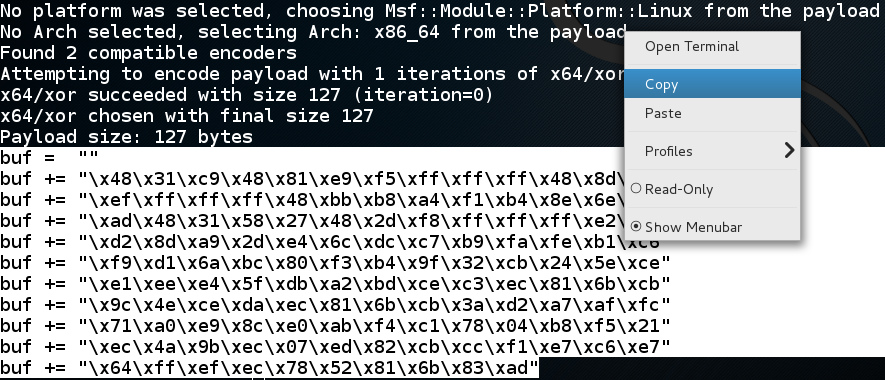
–p parameter means the payload and we passed the linux/x64/shell\_bind\_tcp.

–b parameter means bad-chars. When you –b don’t want to see specific chars in your shell code you should pass them to this parameter. We passed the \x00 because it has a special mean for a string. You can control on asci table. Also we could pass \x0a because also this is a special mean for a string and so on..\

–f means format. Which format did we need? Python. That’s why we passed python if we wanted c we should pass c.

----------------------------------------------------------------------------------------------------------------

Now you can see the generated shell code by msfvenom. Just copy it and use on the next parts of report.



----------------------------------------------------------------------------------------------------------------

**SECOND EXAMPLE (SHELL\_CODE – DELETING FILE)** [7]

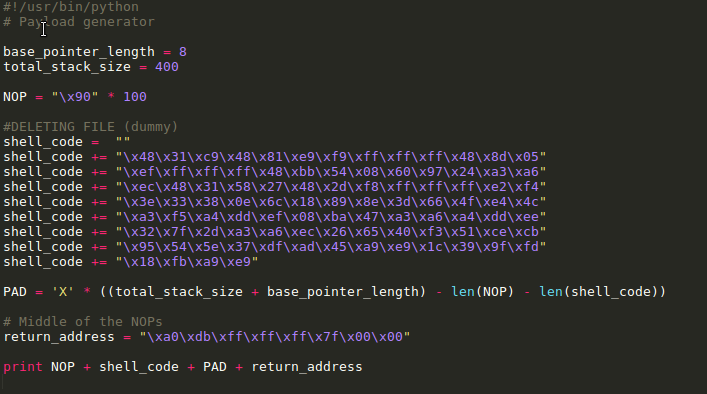
----------------------------------------------------------------------------------------------------------------

We exploited the same code and run a shell script which deletes a file whose name is dummy. [7]



[7]

----------------------------------------------------------------------------------------------------------------



Here it is our payload generator. You can also see the shell code in this.

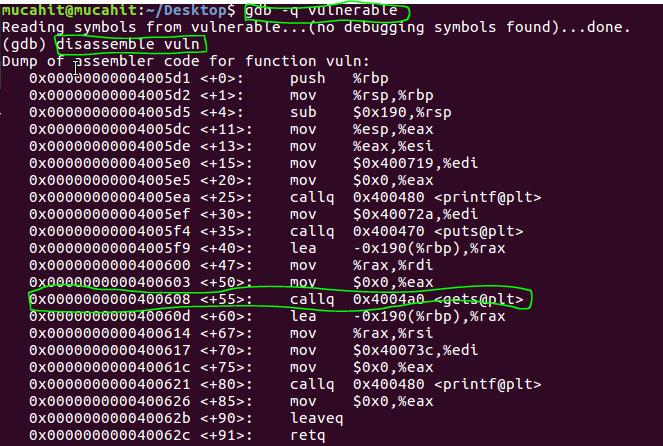
----------------------------------------------------------------------------------------------------------------

----------------------------------------------------------------------------------------------------------------



We used it and write our payload to **textFileShell**.

----------------------------------------------------------------------------------------------------------------

We go into the gdb with debug our code and disassable the vuln function to see when the gets function are being called. As you can see the gets function is called in 55 line.

----------------------------------------------------------------------------------------------------------------

Now we will add the break point on that position and the other position which is after the calling of gets function for example to 60.

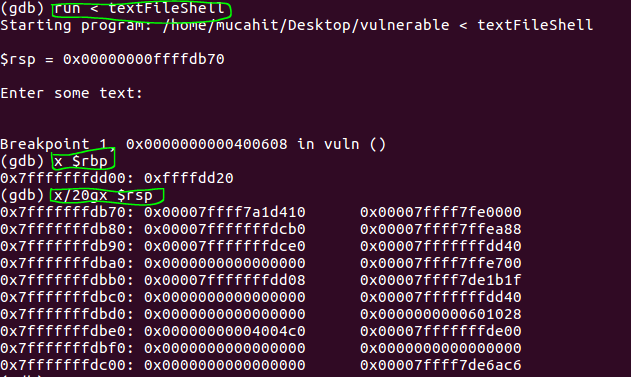
----------------------------------------------------------------------------------------------------------------



First breakpoint is before the gets function and the second is after the gets function.

----------------------------------------------------------------------------------------------------------------

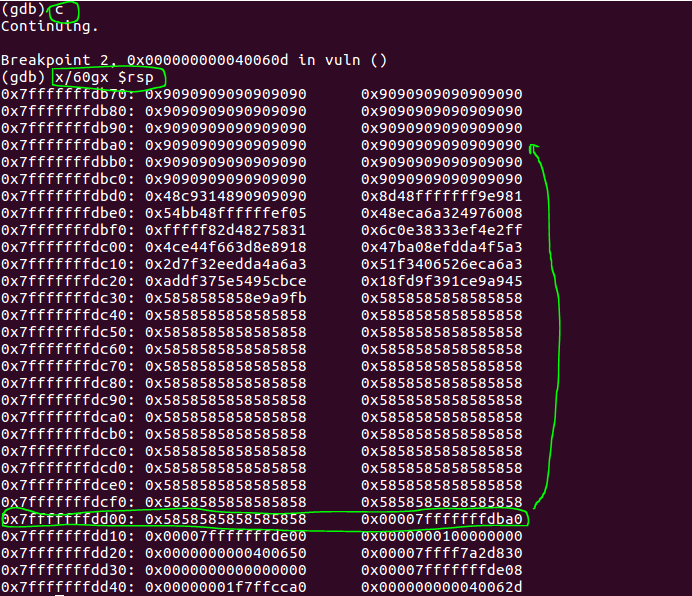
----------------------------------------------------------------------------------------------------------------



When we run the code with our pregenerated text file (payload). We will stop at the first break point and we can see our registers status. As you can see now the stack is not filled with our payload values because this time the gets function hasn’t called yet.

----------------------------------------------------------------------------------------------------------------

----------------------------------------------------------------------------------------------------------------

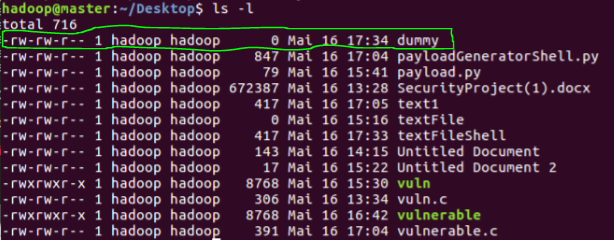


When we continue the gets function is called at the second break point and now you can see the stack filled by our payload. No operations, shell\_code, padding and the return address. As you can see the return address going to the middle of the nops. **We wrote the address directly because we already controlled the stack before. And now you should find the middle of the nops address and you should give it to the return address that in the payloadGeneratorShell.py.** Then the instruction pointer points to the middle of the nops. Finally the code will jump there and run our shell code.

----------------------------------------------------------------------------------------------------------------

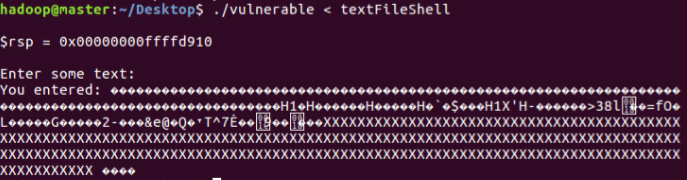
----------------------------------------------------------------------------------------------------------------

We can run the code from outside of the gdb. Now you can see the file which is named as a dummy. We will delete it now.



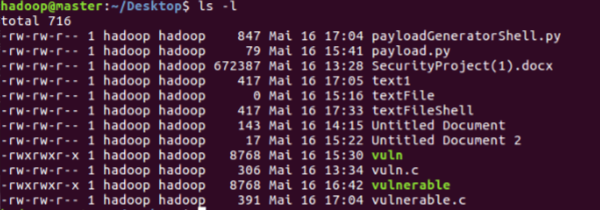
----------------------------------------------------------------------------------------------------------------

We run the code and now let’s look at the files.



----------------------------------------------------------------------------------------------------------------

The dummy file was deleted.



----------------------------------------------------------------------------------------------------------------

**THIRD EXAMPLE (SHELL\_CODE – TCP CONNECTION)** [7]

----------------------------------------------------------------------------------------------------------------

We exploited the same code and run a shell script which deletes a file whose name is dummy. [7]



[7]

----------------------------------------------------------------------------------------------------------------



Here it is our payload generator. You can also see the shell code in this.

----------------------------------------------------------------------------------------------------------------

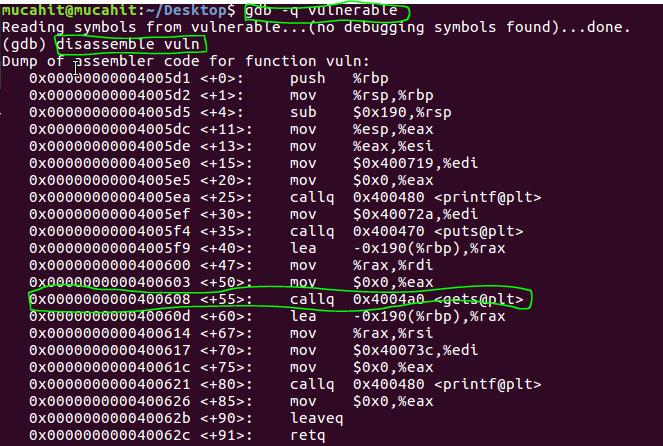
----------------------------------------------------------------------------------------------------------------



We used it and write our payload to **textFileShell**.

----------------------------------------------------------------------------------------------------------------

We just changed the shell code and we did the same transactions again.

We go into the gdb with debug our code and disassable the vuln function to see when the gets function are being called. As you can see the gets function is called in 55 line.

----------------------------------------------------------------------------------------------------------------

Now we will add the break point on that position and the other position which is after the calling of gets function for example to 60.

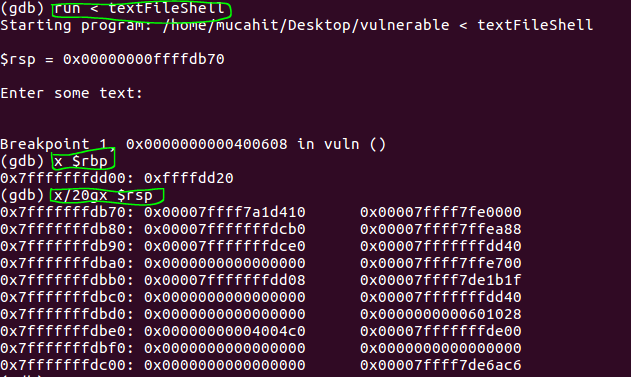
----------------------------------------------------------------------------------------------------------------



First breakpoint is before the gets function and the second is after the gets function.

----------------------------------------------------------------------------------------------------------------

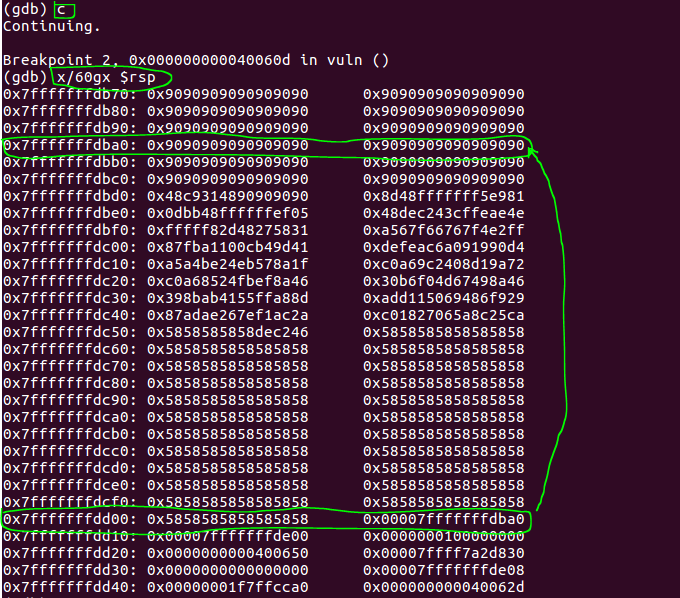
----------------------------------------------------------------------------------------------------------------



When we run the code with our pregenerated text file (payload). We will stop at the first break point and we can see our registers status. As you can see now the stack is not filled with our payload values because this time the gets function hasn’t called yet.

----------------------------------------------------------------------------------------------------------------

----------------------------------------------------------------------------------------------------------------

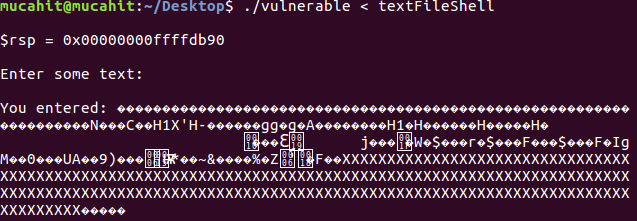


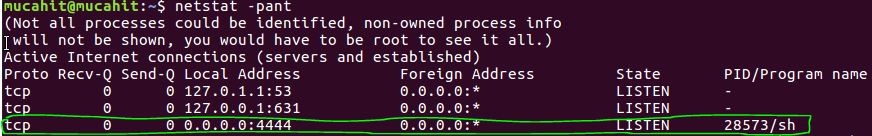
When we continue the gets function is called at the second break point and now you can see the stack filled by our payload. No operations, shell\_code, padding and the return address. As you can see the return address going to the middle of the nops. **We wrote the address directly because we already controlled the stack before. And now you should find the middle of the nops address and you should give it to the return address that in the payloadGeneratorShell.py.** Then the instruction pointer points to the middle of the nops. Finally the code will jump there and run our shell code.

----------------------------------------------------------------------------------------------------------------

----------------------------------------------------------------------------------------------------------------

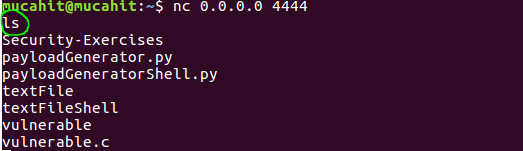
We can run the code from outside of the gdb. When we run the code it listened the tcp port 4444 through our shell code.





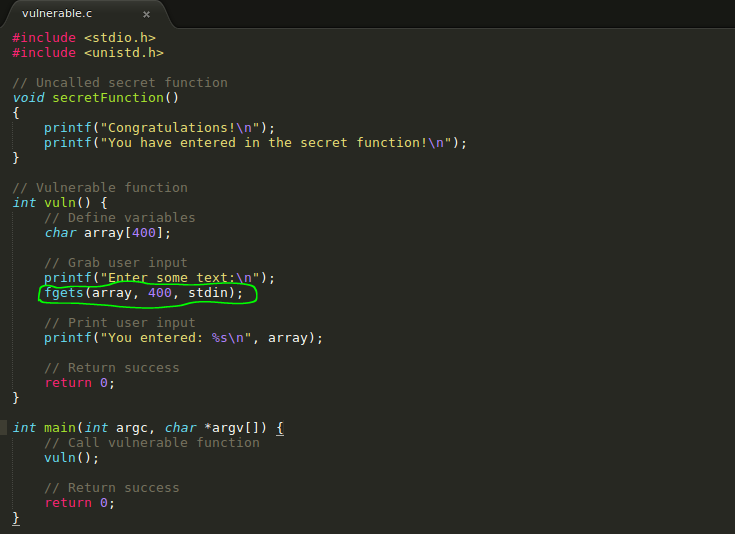
----------------------------------------------------------------------------------------------------------------

As you can see when we netcut our listening port we can access everything. We can see files with **ls** like below screenshot. We can remove files. We can do everything.



----------------------------------------------------------------------------------------------------------------

# PROTECTING



We used the fgets() function to protect ourselves from buffer overflow. It’s just an example. We can use various way for protection.

**assistant resources**

[1] [ Buffer overflow attack with Computerphile ] <https://www.youtube.com/watch?v=1S0aBV-Waeo> [ Better understanding of Shell code exploiting ]

[2] [ Linux 64-bit Buffer Overflow Tutorial ] <http://www.therabb1thole.co.uk/tutorial/linux-64-bit-buffer-overflow-tutorial/> [ Better understanding of gdb ]

[3] [ Basics of 64 bits buffer overflow ] <https://dl.packetstormsecurity.net/papers/attack/64bit-overflow.pdf> [Better understanding Shell code and gdb ]

[4] [Exploiting a 64-bit buffer overflow ] <https://bytesoverbombs.io/exploiting-a-64-bit-buffer-overflow-469e8b500f10>

# References

[5] [ Buffer overflow on 64 bit ] <https://stackoverflow.com/questions/15533889/buffer-overflows-on-64-bit> [ Disabling the protection mechanisms ]

[6] [ Buffer Overflow Exploit ] <https://dhavalkapil.com/blogs/Buffer-Overflow-Exploit/> [ Theoritical part of report ]

[7] [ 64-Bit Buffer Overflow Exploit ] <https://samsclass.info/127/proj/p13-64bo.htm> [ Shell code exploits ]