

**实验报告**



**题目： 缓冲区溢出攻击实验**

**班 级： 2022211305**

**学 号： 2022211119**

**姓 名： 赵宇鹏**

**学 院：计算机学院（国家示范性软件学院）**

**2022年 11 月 23 日**

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。

1. 实验环境
2. 服务器：10.120.11.12（关机时间：2022年12月5日00：00）
3. Linux
4. Objdump命令反汇编
5. GDB调试工具
6. 积分榜：http://10.120.11.13:19320/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；

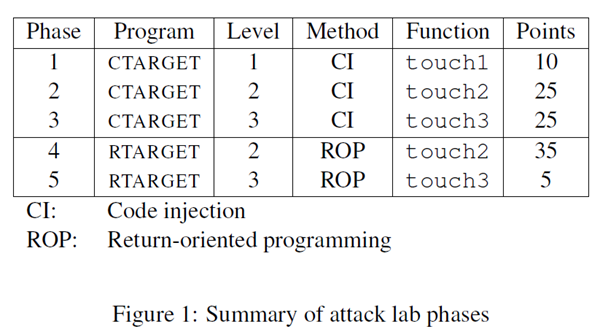
cookie.txt；

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ctraget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：



四、实验步骤及实验分析

**准备工作**：

阅读所给文档，学习所需要的关于栈的知识，然后解压缩给出的压缩包，对ctarget文件和rtarget文件进行反汇编，生成反汇编文件，了解gadget farm和HEX2RAW的作用和用法。

HEX2RAW期望一个或多个空格分隔的两位数十六进制值，并通过这些十六进制数生成相应攻击字符。

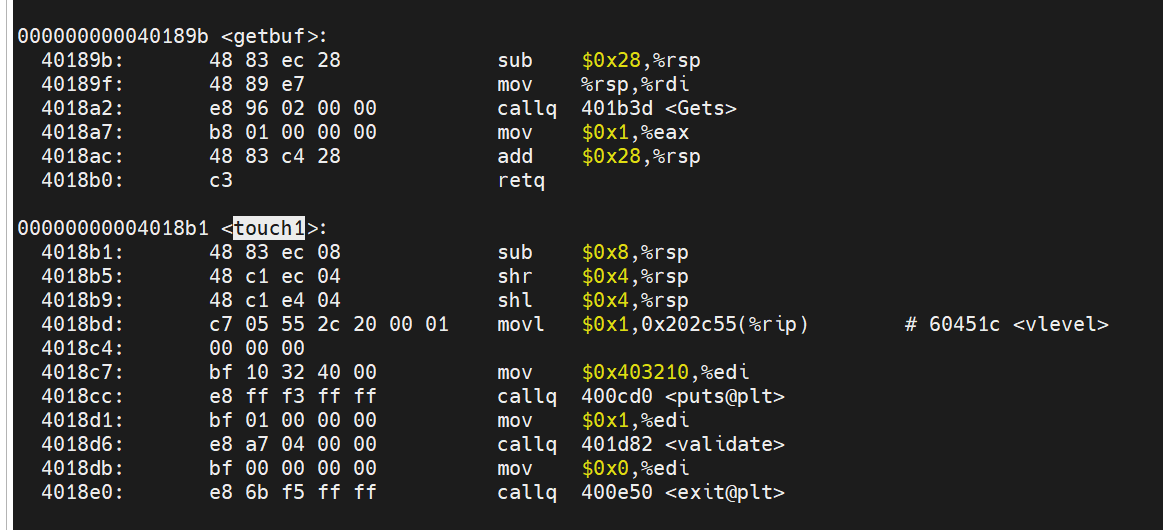
**第一部分：**

在前三个阶段，我们的漏洞字符串将攻击CTARGET。这个程序的设置方式使堆栈位置从一次运行到下一次运行都是一致的，因此堆栈上的数据可以被视为可执行代码。这些特性使程序容易受到攻击，其中利用字符串包含可执行代码的字节编码。所以第一部分的三道题都可通过注入代码实现。

**Level1:**

Phase1要求我们让CTARGET在getbuf执行其返回语句时执行touch1的代码，而不是返回到test。

现在我们首先来查看一下getbuf和touch1的反汇编代码。



可以看到touch1的地址为**0x4018b1**，而getbuf开辟的空间为**0x28**，也就是40，所以我们需要让函数结束时不返回到test而是跳转到touch1，那么我们需要通过缓冲区溢出来进行操作，

|  |
| --- |
|  |
|  |
|  |
|  |
| %rsp//返回地址 |
|  |
|  |
|  |
|  |
|  |

所以我们只需要输入48个字节，前四十个字节填满getbuf开辟的40大小的空间，后八个用来储存touch1的地址，这样，在ret时，就会弹出touch1的地址并且跳转到那里去。

为防止前四十个字节内容产生影响，我们将前面的字节全都设为00

于是输入为

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

b1 18 40 00 00 00 00 00

这里要注意是小端输入，所以输入时顺序需要翻转。

然后使用HEX2RAW来生成注入的数据



**Level2:**

Phase2要求我们调用touch2函数，同时把cookie的值传递给touch2，根据题目所给提示，函数的第一个参数是在寄存器%rdi中传递的，cookie的值需要在%rdi中传递。

因为题目限制不能使用jmp和call语句，所以我们要跳转仍然需要通过ret来进行操作。

首先来查看cookie的值：**0x4dae86d7**



再看看touch2的地址：**0x4018e5**



在 x86 架构中，ret 指令的工作原理如下：

从堆栈顶部弹出返回地址。

将返回地址加载到程序计数器（PC）中。

控制流跳转到返回地址，继续执行调用该函数的地方。

那么我们只需要像phase1一样，在ret处覆盖上我们注入代码的地址，这样ret就会跳转到我们注入的代码，而因为cookie的值需要在%rdi中传递，传递完之后就可以跳转到touch2，所以还需要通过push压入touch2的地址，在ret时弹出，就会跳转至touch2处。

所以我们需要执行的代码为

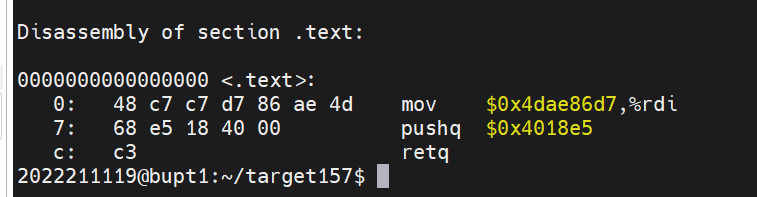
mov $0x4dae86d7, %rdi

push $0x4018e5

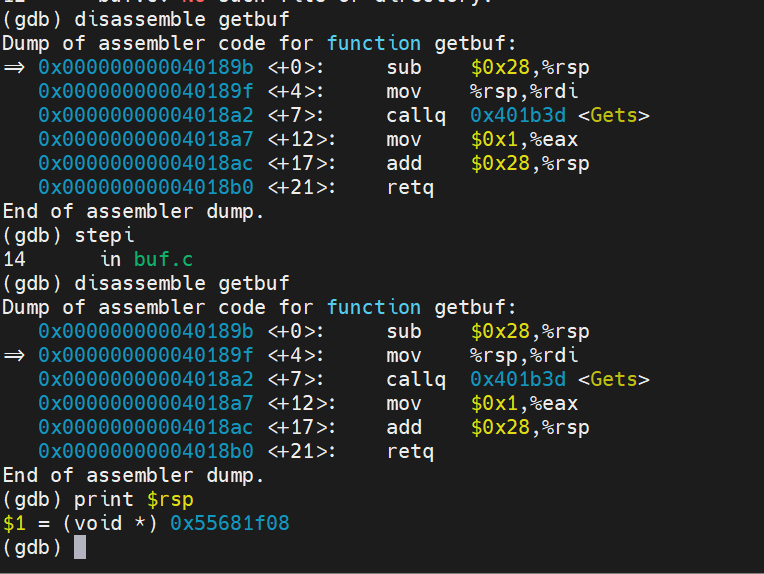
ret

通过unix> gcc -c example.s

unix> objdump -d example.o > example.d指令生成机器码



目前还需要寻找注入代码的存放地址，由于注入代码需要放到栈内，我们可以通过gdb调试指令来查找栈顶地址，将注入代码放在该地址即可



可以看见%rsp地址为**0x55681f08**

所以phase2的输入为

48 c7 c7 d7 86 ae 4d 68

e5 18 40 00 c3 00 00 00//注入的代码

00 00 00 00 00 00 00 00

00 00 00 00 00 00 00 00

00 00 00 00 00 00 00 00

08 1f 68 55 00 00 00 00//返回注入代码地址

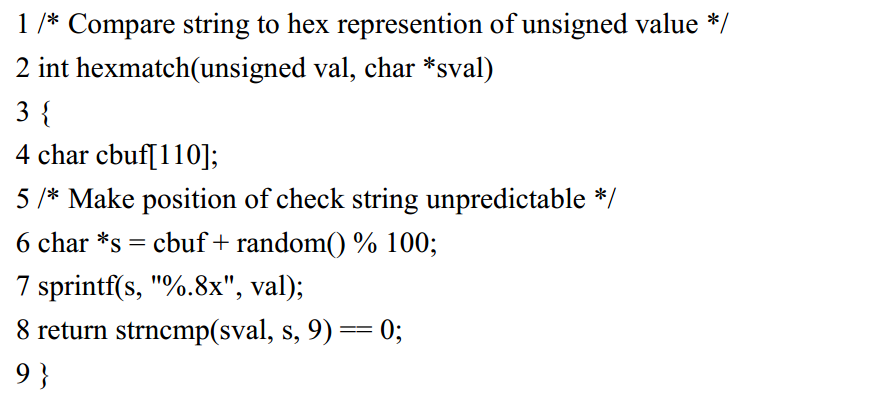
**Level3:**

Phase3要求为栈中覆盖指定信息，需要我们将cookie作为字符串传递给touch3并且跳转到touch3。

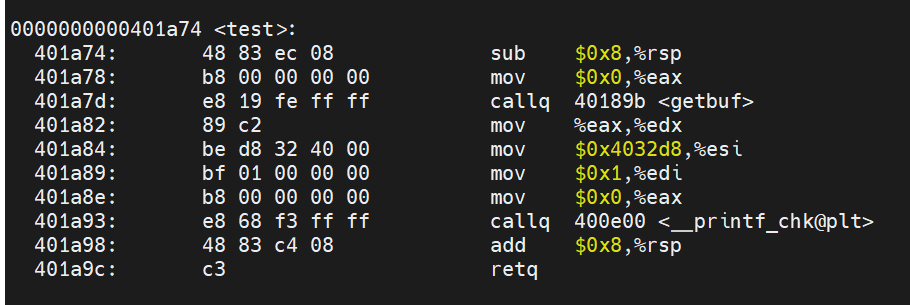
根据提示，当调用函数hexmatch和strncmp时，它们将数据推入堆栈，覆盖getbuf使用的缓冲区的内存部分。因此，需要小心放置cookie的字符串表示形式的位置。

首先，根据cookie值转换成ascii码：**34 64 61 65 38 36 64 37**

然后来查看提示里面提到的hexmatch以及touch3



这里使用了random，所以s说随机的，有可能会覆盖掉我们储存在栈内的字符串，所以字符串需要储存在一个安全的地址，我们可以考虑一开始就出现的test函数：

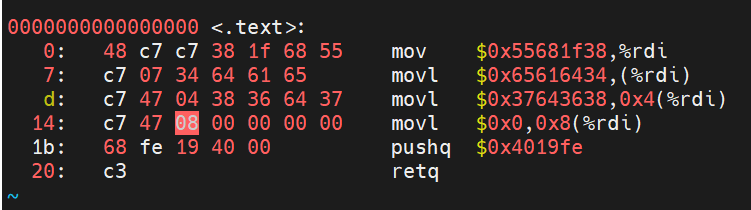




同phase2，我们找到了test的栈顶地址用来储存我们的cookie值，地址为**0x55681f38**。然后查看touch3的地址：**0x4019fe**

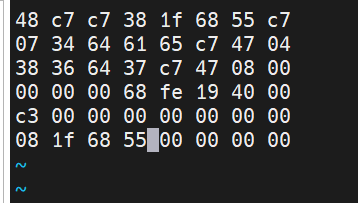


所以本题的汇编代码以及对应编码为：



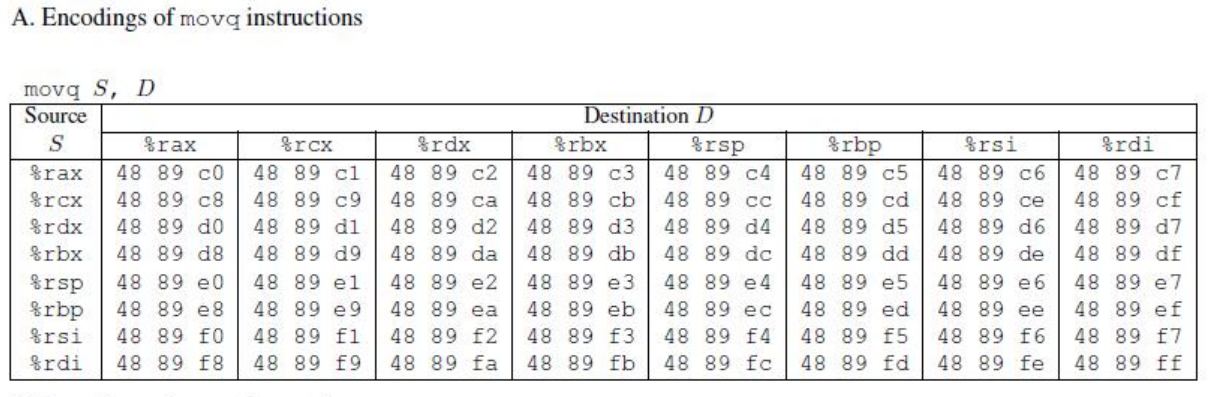
此代码将cookie的值四个字节一次地赋值给0x55681f38以及后续偏移地址，然后根据题目提示加上00作为结尾，最后压入touch3的地址来最后弹出跳转。其他思路同phase2。

所以最后的机器码如下：



**第二部分：**

对程序RTARGET执行代码注入攻击要比对CTARGET执行代码注入攻击困难得多，因为它使用两种技术来阻止这种攻击:它使用随机化，以便每次运行时堆栈位置不同。这使得我们无法确定注入代码的位置。它将保存堆栈的内存部分标记为不可执行，因此即使我们可以将程序计数器设置为注入代码的开始，程序也会因分段错误而失败。幸运的是，聪明的人已经设计出一些策略，通过执行现有代码而不是注入新代码来完成程序中有用的事情。最一般的形式被称为面向回报的规划(ROP)[1,2]。ROP的策略是在现有程序中识别由一条或多条指令后面跟着指令ret组成的字节序列。这样的段称为小部件。也就是说，第二部分我们无法直接注入我们的代码，所以我们需要通过寻找farm文件中的小工具，根据它们的机器码拼接成我们所需要的代码。



**Level2:**

对于阶段4，我们将重复阶段2的攻击，但是在程序RTARGET上使用我们的小工具场中的小工具进行攻击。我们可以使用由以下指令类型组成的小工具构建解决方案，并且只使用前8个x86-64寄存器(%rax - %rdi)。并且我们只需要两个小工具就可以完成这一次攻击。

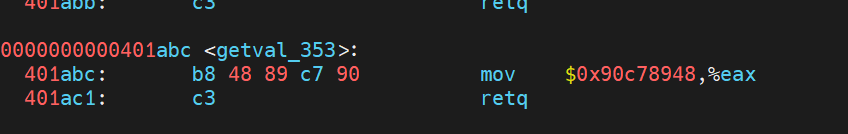
要求调用touch2，也就是要求

mov $0x4dae86d7,%rdi

push 0x4018e5

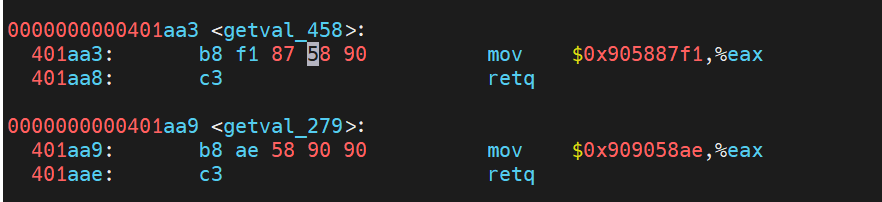
Ret

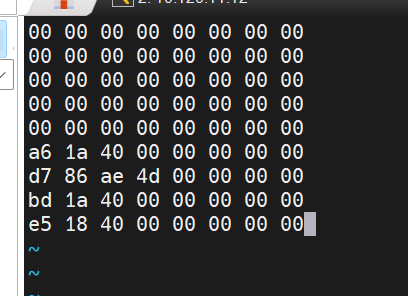
可是我们并没有找到带立即数的小工具，可以采用pop %rdi再直接给%rdi赋值的方式，可是遍历搜索下没有找到%rdi，那么我们可以间接传递，先找到可以传递给%rdi的寄存器的操作mov %rax, %rdi对应 48 89 c7



找到相关地址为**0x401abd**

再找pop %rax对应58





我们取**0x401aa6**

那么顺序为

pop %rax

0x4dae86d7

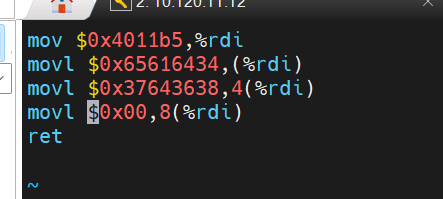
mov %rax, %rdi

0x4018e5

**Level3:**

Phase5要求调用touch3，即传递cookie字符串，通过八个小组件来实现第三题的功能

第三题汇编代码为



Cookie : 0x4dae86d7

寻找对应ascii码 34 64 61 65 38 36 64 37

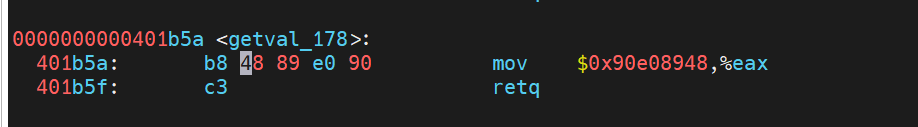
00000000004019fe <touch3>

可是在第二部分里面栈的地址是随机的，无法直接把cookie存进去

那么先找找怎么把%rsp的值移出来

查表发现所有对%rsp的操作都为48 89 \*\*

搜索发现

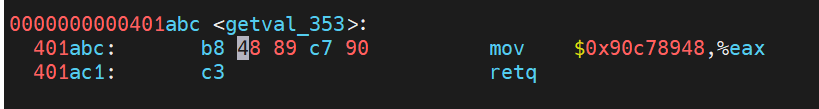


48 89 e0对应movq %rsp, %rax，所有第一步是把栈顶%rsp储存到%rax中，地址为**0x401b5b**，我们需要根据栈顶地址和偏移量来计算cookie储存位置，在搜索48时，发现了一个特殊的小组件



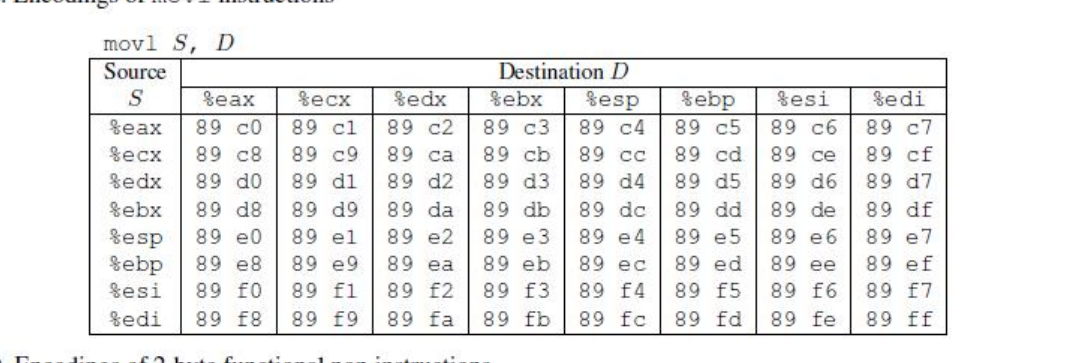
地址为**0x401ada**，所以之后的cookie地址计算可以通过它来实现基址和偏移量一个存%rdi，一个存%rsi

首先看看基址能否直接从%rax移到%rdi，对应48 89 c7

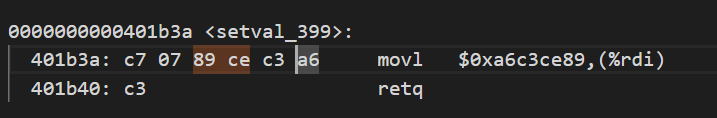


地址为**0x401abd**，移到了%rdi，则基址移动完成，接下来看如何移动到%rsi。

首先搜索各种以%rsi为终点的操作，查询失败，在farm中无该操作对应机器码，那么根据

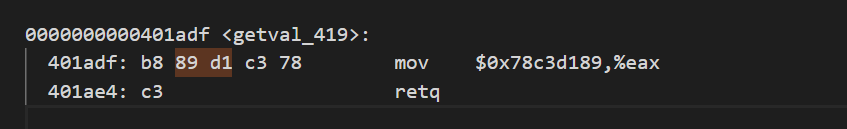


查找移到%esi的操作



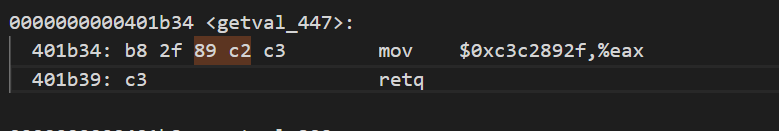
对应mov %ecx, %esi地址为**0x401b3c**

再寻找移到%ecx的



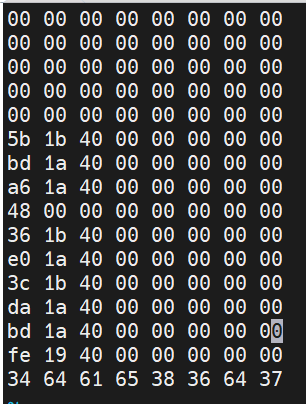
对应mov %edx, %ecx地址为**0x401ae0**

再找移动到%edx的



对应mov %eax, %edx地址为**0x401b36**

寻找结束，偏移量的赋值我们可以使用pop %rax来进行赋值，然后按照

Movl %eax, %edx

Movl %edx, %ecx

Movl %ecx, %esi

来实现偏移量的转移

所以总体顺序为

Cookie : 0x4dae86d7

寻找对应ascii码 //为防止被当成指令，需要放到所有代码的最后

计算偏移量：

当执行movq %rsp, %rax时，%rsp已经往地址增大方向移动八字节了，所以此时存在%rax里面的%rsp指向的是mov %rax, %rdi语句，计算上为偏移量预留的8字节，最终cookie的偏移量为**0x48**，所以最终汇编代码逻辑如下

movq %rsp, %rax 0x401b5b

mov %rax, %rdi 0x401abd

pop %rax 0x401aa6

0x48 48

mov %eax,%edx 0x401b36

mov %edx,%ecx 0x401ae0

mov %ecx,%esi 0x401b3c

lea (%rdi, %rsi, 1), %rax 0x401ada

mov %rax, %rdi 0x401abd

Touch3 0x4019fe

34 64 61 65 38 36 64 37

五、总结体会

1、在进行phase2时，传递cookie时没有添加$符号，导致传递的不是立即数，代码出错，进行仔细比对后发现问题并改正。

2、在phase3中储存cookie时，刚开始想随便找一个地址应该就可以运行，于是找到了main函数的地址，而非栈内空间，导致发生段错误，后经尝试，将cookie存进了test函数的栈顶地址,解决问题。

3、做phase5时遇到的问题最多，由于phase5要求调用touch3，所以需要想办法把cookie的ascii码传进去，在phase3中我采用了movl的办法分两次将cookie传进相应地址，在phase5中却没有对立即数直接进行操作的指令，同时栈的地址也不是固定的，后发现有lea操作，同时意识到可以直接将cookie的值写入缓冲区，利用lea来确定其地址，在执行该过程中，需要通过各种mov操作来将偏移量移到%rsi寄存器中，在寻找这些操作时，只顾着寻找编码相同的操作，却忽视了若没有c3结尾，则计算机会一直沿着这条指令往下继续执行其他指令直到c3返回为止，因此在传输偏移量时找到的mov指令全部都是错误指令，最后用到的小工具也只有7个，发生段错误。在改正指令后，对于cookie的存放位置也有错误，之前为图方便直接把cookie的值放到了所有代码的第一行，这样偏移量即为0x8，此处便出现了两个错误，第一，cookie放到所有代码前面会导致计算机将其也识别为一个操作而非ascii码，第二，当执行一条操作语句时，%rsp会移向下一条操作语句，所以当计算机执行movq %rsp, %rax时，%rsp指向的是下一条语句，存进%rax的地址也是下一条语句的地址，因此偏移量的计算要从下一条语句开始才可以。为了纠正错误，我把cookie放到了最后，这样计算机在跳转到touch3后就会结束，不会执行cookie，第二，我根据cookie的位置重新计算量偏移量，总共有十一条语句，除去第一条还剩十条，偏移量为0x48。

本次实验大概花费了10小时，其中小半时间用在了阅读文档上面，文档提供的信息非常关键，如果不仔细阅读，很容易忽视一些重要细节和题目要求，当失败时，我会再去通篇阅读一下实验说明以寻找关键信息来检查自己代码是否符合规范，剩下的时间中又有一半用来做phase5，因为其要求同phase3，而phase4的解决过程相对来说并不复杂，于是便想挑战一下phase5，在phase5中我也发现很多认识上的错误，对汇编语言有了更深刻的理解。五个Phase的难度是层层递进的，Part 1让我对部分汇编指令以及栈的原理有了更深的领悟，Part 2让我学习到了跳转拼接的相关知识，实现成功的攻击需要对每一个字节都能有足够的仔细认真。

意见和建议：本次实验文档均为英文，阅读难度较大，同时HEX2RAW文件的用法在文档的最后，导致刚开始做时手无足措，不知道如何使用，希望可以适当中文化同时微调一下内容，把相关使用方法移到前面。

六、诚信声明（不签扣10分）

需要填写如下声明，并在底部给出手写签名的电子版。

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：

1. 来自高天涛同学提醒的细节：计算偏移量时要注意%rsp已经往地址增大方向移动了一条语句。
2. 来自王骊琨同学的提醒：语句的结尾需要为c3，否则不会停止，会一直往下执行

此外，我还参考了以下资料：

1. <https://blog.csdn.net/m0_65159351/article/details/128642734>了解ret指令返回操作。

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。

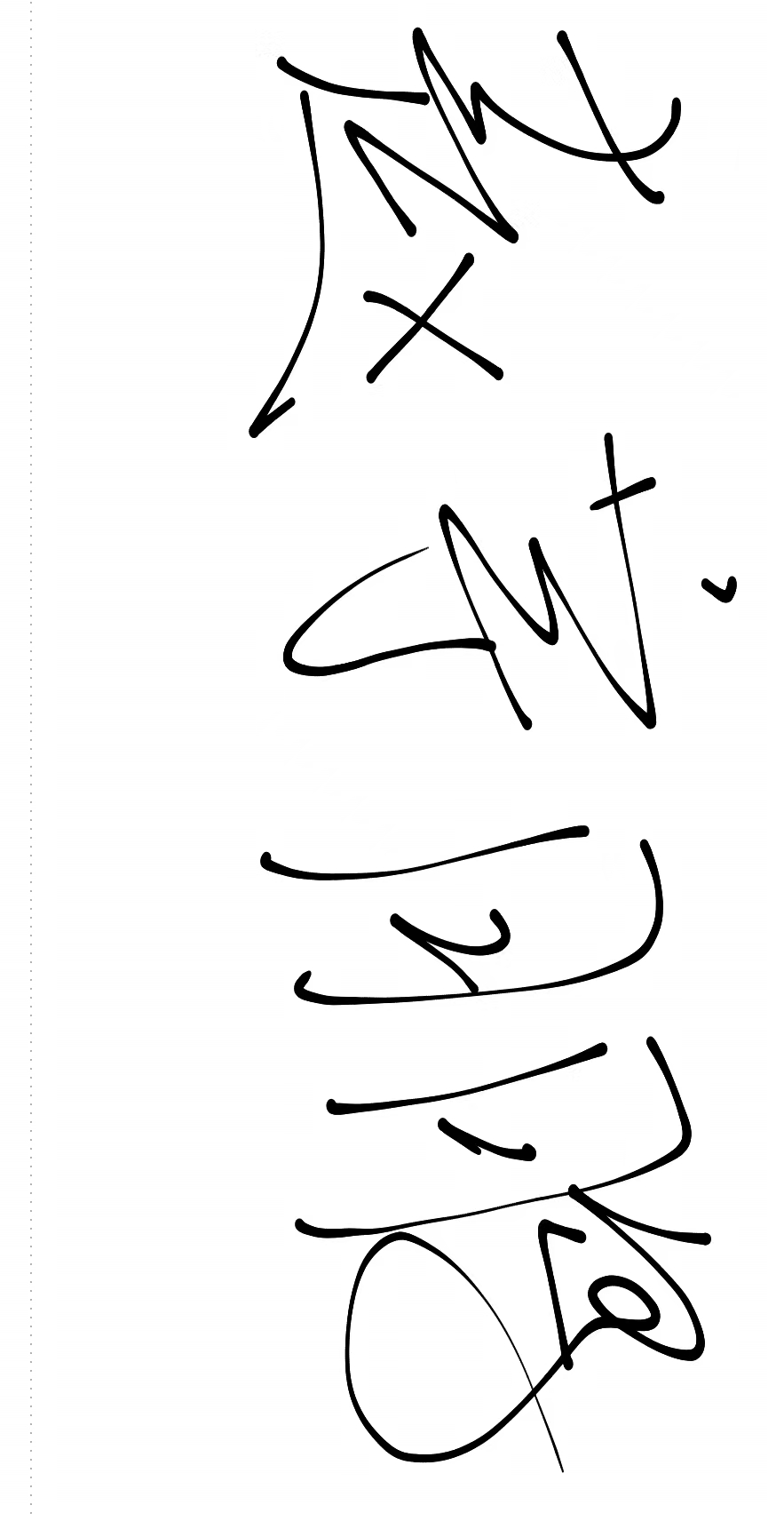
我清楚地知道，从以上方面获得的信息在一定程度上降低了实验的难度，可能影响起评分。

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）