**计算机系统基础**

**实验报告**

**--**

**实验名称：AttackLab**

**学号：6319000359**

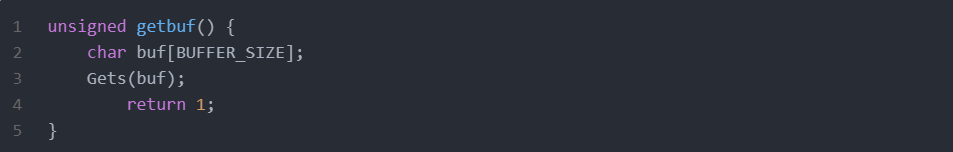
**姓名：张明君**

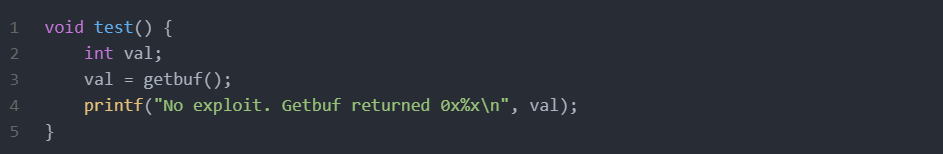
**班级：留学生班**

**日期：2020.05.17**

**题目**

实验中的程序 CTARGET 和 RTARGET 都调用了函数 test：



函数 test 又调用了函数 getbuf：

函数 Gets 从标准输入读取输入。

函数 Gets 没有进行边界检查，因此是不安全。

数组 buf 是局部变量，存储于栈上，因而通过输入可以修改栈上数据达到攻击目的。

BUFFER\_SIZE 是个常量，使用 gdb ctarget 和 disas getbuf 可查看其大小（在本实验

中，BUFFER\_SIZE = 0x28）。

### **PHASE 1**

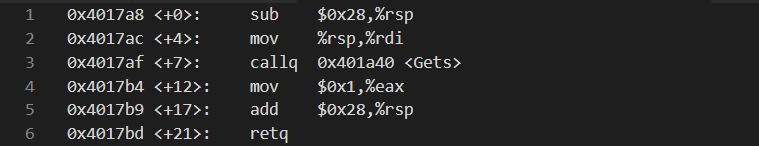
#### **要求**

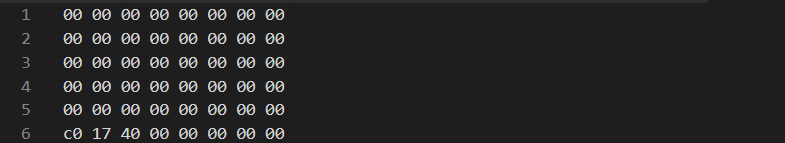
针对程序 CTARGET， 修改函数 test 调用 getbuf 的返回地址，使其返回 touch1 函数而非 test。

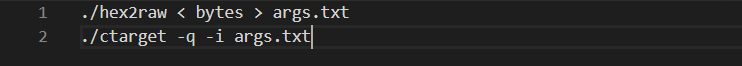
#### **做法**

使用 disas touch1 查看函数 touch1 的地址为 **0x4017c0**。

使用 disas getbuf 查看汇编代码：

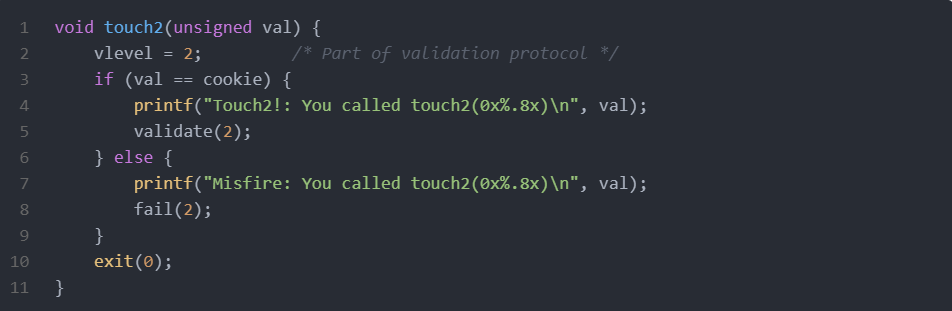


当读入字符串超过 40 时，依据栈帧结构，第 41~48 个字节即为返回地址，因此将其设置为 touch1 的地址即可。实验工具包提供了 hex2raw 将字节码（十六进制）转换成字符串，输入字节为：

使用 hex2raw 转换字节码序列：

### **PHASE 2**

**要求**

针对程序 CTARGET， 修改函数 test 调用 getbuf 的返回地址，使其返回 touch2 函数而非 test。

#### **做法**

使用 disas touch2 查看函数 touch2 的地址为 **0x4017ec**。

函数 touch2 需要传递参数 val，该参数的值等于实验工具包中 cookie.txt 保存值（在本实验中 **cookie=0x59b997fa**）。

函数的参数传递先使用 6 个寄存器，更多的参数保存在栈中。前 6 个寄存器如下图所示：



第一个参数寄存器为 %rdi，因此需要注入执行代码将 cookie 值保存到寄存器 %rdi 中。

首先**执行代码本质**上是字节序列，其次返回地址指向下一条指令的地址，最后 CTARGET 的堆栈位置每次运行都保持一致。利用这三条性质可将返回地址指向栈的某个位置，并在该位置填入合法的指令字节序列，注入攻击代码。

**A. 查看数组 buf 在栈上的保存位置。**

使用 b getbuf 在函数 getbuf 的第一条指令设置断点，然后使用 n 2 执行一下两条指令：

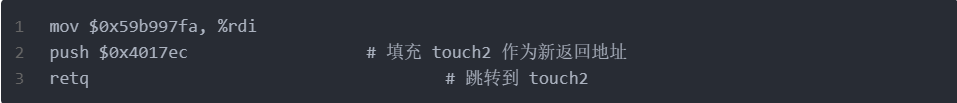


此时 %rdi = %rsp，为函数 Gets 的参数 buf，因此输入字符串的起始地址等于 %rsp。

使用 print /x $rsp 查看 %rsp 的值为 0x5561dc78，即 CTARGET 每次运行时 buf 的地址都是 0x5561dc78。

**B. 生成注入代码的字节序列。**

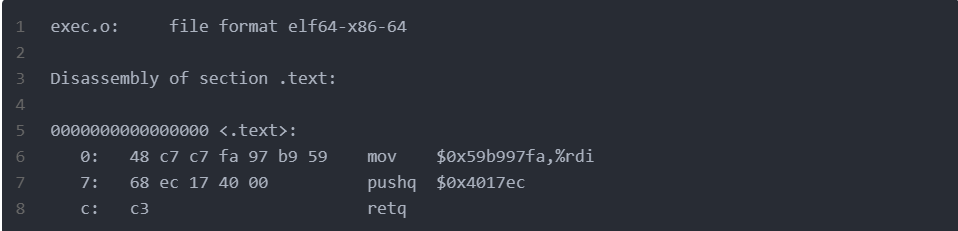
与 Level 1 类似，用 0x5561dc78 覆盖 getbuf 的返回地址，然后从 0x5561dc78 开始填充注入代码：保存 cookie 值到 %rdi，然后跳转执行 touch2，其汇编代码如下：



将上述代码保存为 exec.s，使用以下命令查看指令字节序列：



指令字节序列为



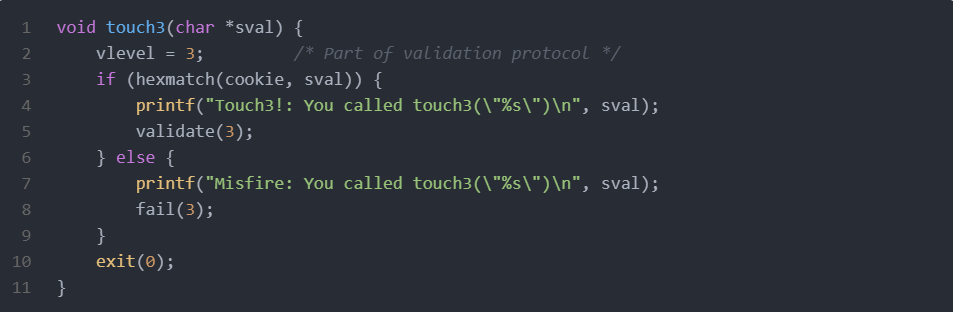
**C. CTARGET 的输入字节序列**



### **PHASE 3**

#### **要求**

针对程序 CTARGET， 修改函数 test 调用 getbuf 的返回地址，使其返回 touch3 函数而非 test。



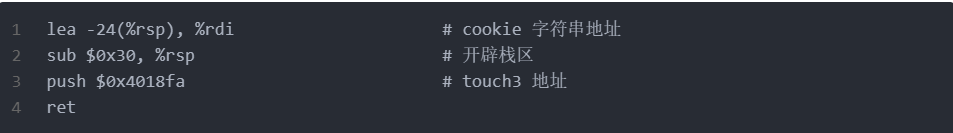
其中，touch3 会调用函数 hexmatch 以比较输入字符串是否和 cookie 的字符串相同，新的函数会使用并覆盖栈上的数据。

#### **做法**

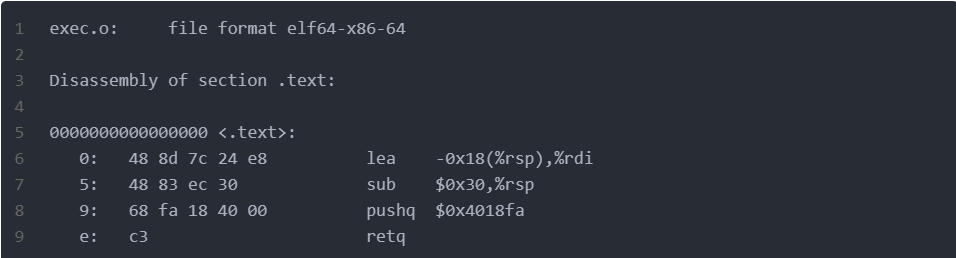
在该实验中需要将 cookie 的字符串保存到栈上，并将其起始地址保存到 %rdi 上。

解法和 Level 2 类似，但要注意一点，touch3 中的 hexmatch 会使用栈而破坏输入的字节序列，导致 cookie 的字符串序列无效。

为了避免上述情况发生，与新函数执行相同，先开辟一段新的栈区以保护输入序列：



使用 gcc -c exec.s 和 objdump -d exec.o 查看指令字节序列：



CTARGET 的输入字节序列：



### **PHASE 4**

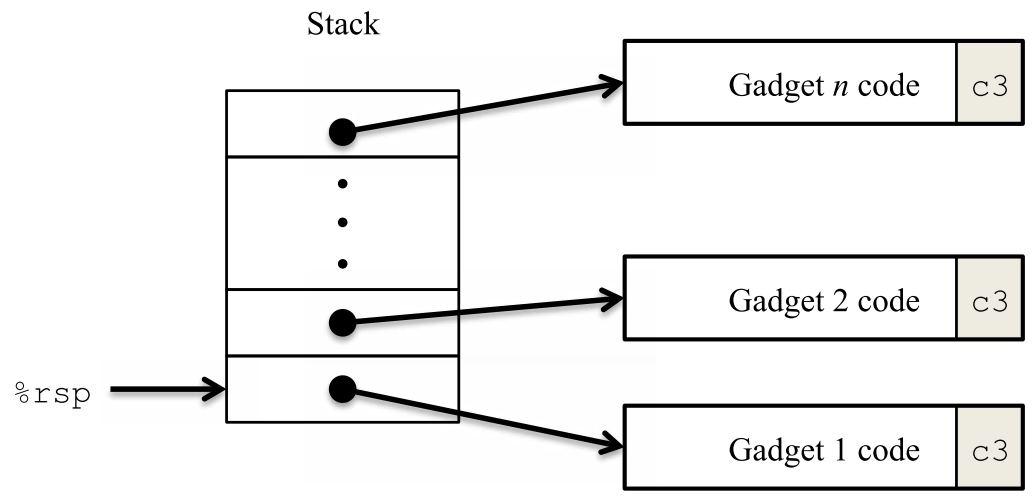
#### **要求**

针对程序 RTARGET，修改函数 test 调用 getbuf 的返回地址，使其返回 touch2 函数而非 test。

在该实验中，禁止执行栈中的指令，并且栈的地址也会发生变化。因此，Level 1~3 的做法失效了。

#### **做法**

在该实验中通过找出已有指令的字节序列作为**工具片段（gadget）**，其原理如下图所示。通过工具片段来解决无法执行栈中指令的问题。



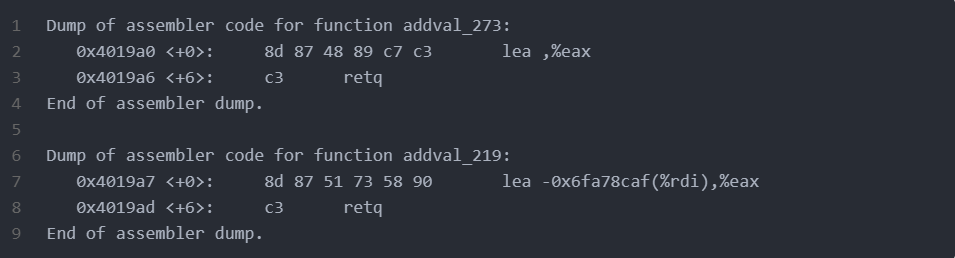
每个工具片段以 ret(编码 0xc3) 结尾，将工具片段地址保存在栈上，ret 依次解栈来执行下一条工具片段。

该实现限制了指令集 movq, popq, ret 和 nop。

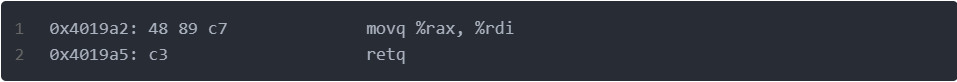
该实验的思路比较清晰：通过 popq 从栈中取数据，然后使用将数据保存到相应的寄存器（如 %rdi）即可。

使用 disas /r getval\_142 依次查看每个工具函数的指令序列。

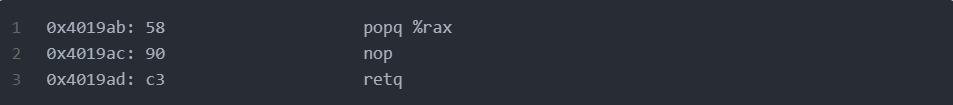
其中，关键的工具函数有 addval\_273 和 addval\_219：



在 addval\_273 中可以提取指令序列：

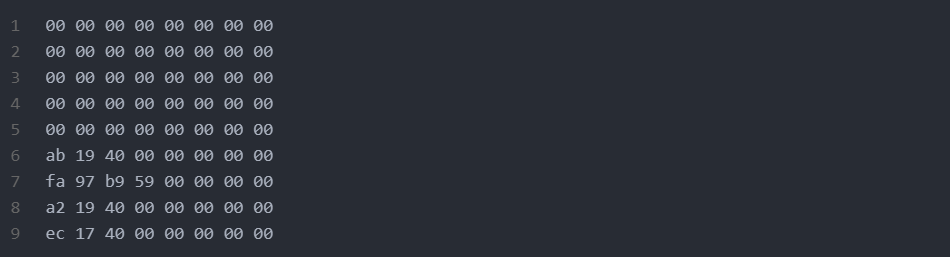


在 addval\_219 中可以提取指令序列：



显然，通过工具片段 0x4019ab 提取 cookie 到 %rax，然后使用 0x4019a2 将 cookie 转移到 %rdi，完成参数构造。

RTARGET 的输入字节序列：



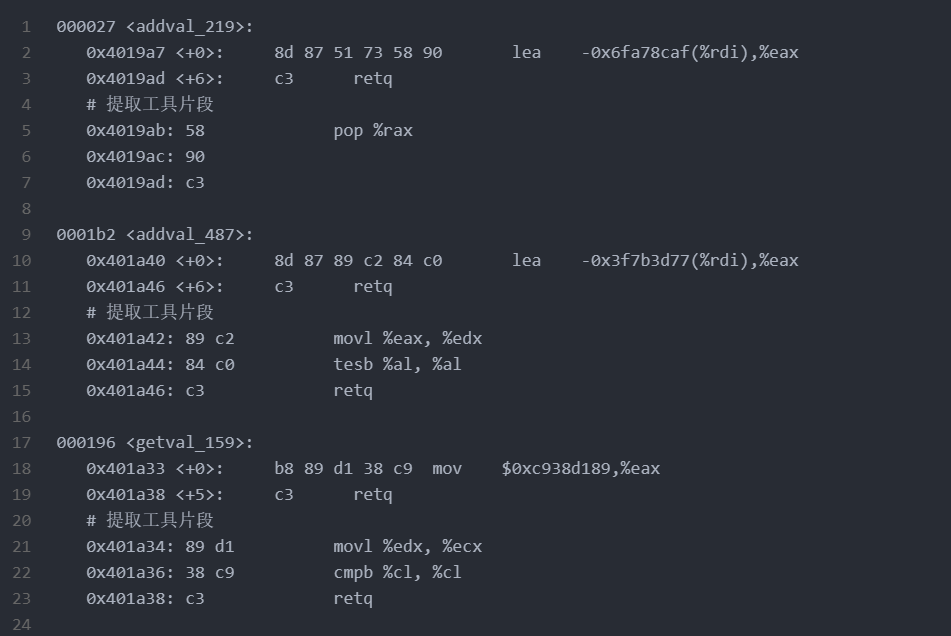
### **PHASE 5**

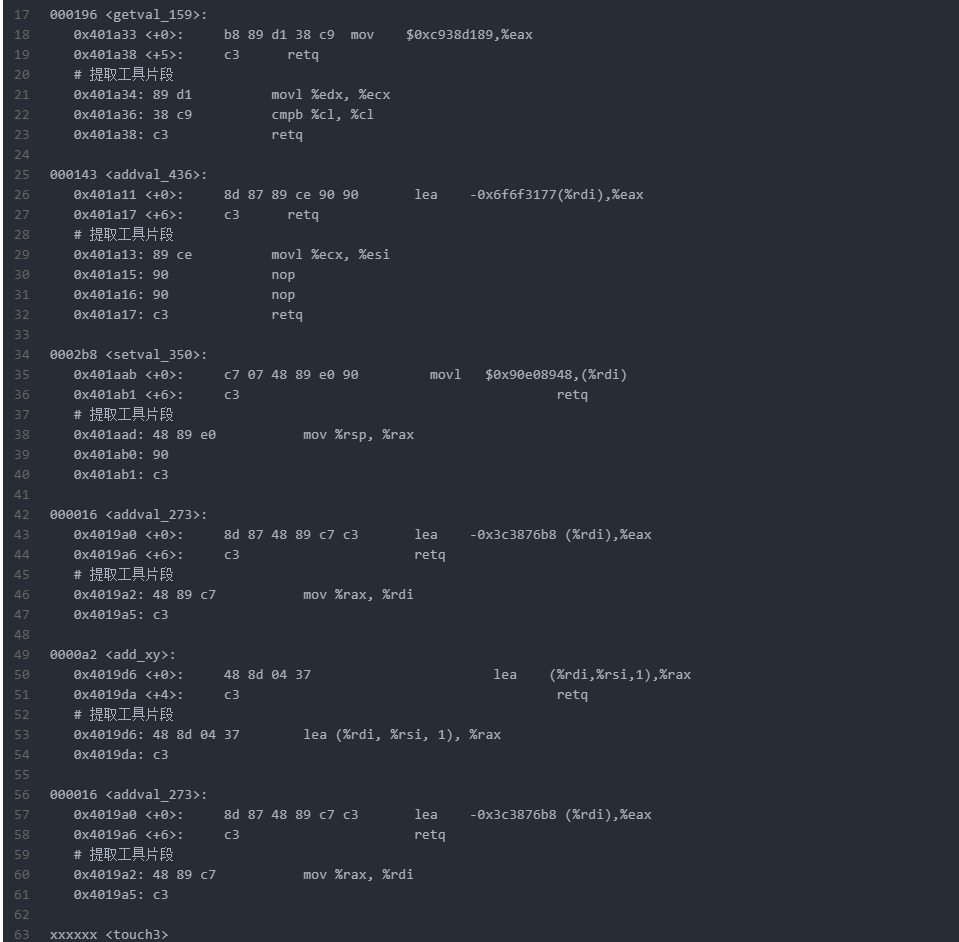
#### **要求**

针对程序 RTARGET，修改函数 test 调用 getbuf 的返回地址，使其返回 touch3 函数而非 test。在该实验中，禁止执行栈中的指令，并且栈的地址也会发生变化。因此，Level 1~3 的做法失效了。

#### **做法**

做法与 Level 4 类似，但是需要更多的工具片段，额外需要一条 lea 指令计算 cookie 字符串的地址。

关键工具函数及提取的工具片段如下，函数顺序表示了执行顺序：



RTARGET 的输入字节序列：