# 汇编与接口探究实验1 刘思锐 3200102708

## 一、背景说明

笔者最初希望探究的主题是不同编译器对rip寻址的处理。但是在研究如下代码时，出现了几行特殊的汇编指令。

```
// origin.c
static int i[] = {1, 1, 2, 3, 5, 8, 13};
int main (){
  int j[] = {1, 1, 2, 3, 5, 8};
  int k = i[1] + j[2];
  return 0;
}
```

上述代码在GCC 9.4下的编译结果为：

```
0x0000555555555149 <+0>:     endbr64
// do sth.
0x00005555555551a1 <+88>:    mov    -0x8(%rbp),%rcx
0x00005555555551a5 <+92>:    xor    %fs:0x28,%rcx
0x00005555555551ae <+101>:   je     0x5555555551b5 <main+108>
0x00005555555551b0 <+103>:   call   0x555555555050 <__stack_chk_fail@plt>
0x00005555555551b5 <+108>:   leave
0x00005555555551b6 <+109>:   ret
```

注意程序开头在压栈之前有一句作用不明的指令 `ENDBR64`；返回之前对rcx寄存器的值做了检查，如果检查不通过会跳转到 `__stack_chk_fail` 而不是正常ret。上述两条语句并没有在C语言源码中体现而是由编译器自动添加的。

于是笔者转而对此展开了探究。

## 二、探究过程

### 2.1 ENDBR保护

#### 2.1.1 原理探究

`endbr64` 显然是一条固定的指令，在查找相关资料后，从《Intel® 64 and IA-32 Architectures Software Developer’ s Manual》中得到了如下信息。

> If the next instruction retired after an indirect JMP or CALL is not an ENDBR32 instruction in legacy and compatibility mode, or ENDBR64 instruction in 64-bit mode, then a #CP fault is generated.

`endbr` 是从属于Intel CET（CONTROL-FLOW ENFORCEMENT TECHNOLOGY）机制的一条指令。支持CET机制的处理器要求每一条非直接跳转的JMP或CALL语句必须跳转到一条 `endbr64`（在32位模式下则为 `endbr32`）指令。这条指令耗时一个时钟周期，不会改变任何reg和flag的值，但如果JMP或CALL语句跳转向了非 `ender` 语句，意味着跳转语句可能已经被篡改，跳向了危险的代码段，因此CPU将会抛出 `#CP` 错误并终止代码执行。

继续翻阅Intel开发手册后得知，出于兼容性和可扩展性的原因，`endbr`检查并不是始终开启的，譬如：

- `endbr64`机器码为`f3 0f 1e fa`（`endbr32`机器码为`f3 0f 1e fb`）。在不支持CET的处理器中，因为逃逸码0f的存在，这条指令会被当做NOP处理。
- 在支持CET的处理器中，可以通过修改CET相关的MSR状态寄存器全局关闭endbr检查。
- 更细粒度的，在支持CET的处理器中，可以通过修改修改CET相关的MSR状态寄存器启用NO_TRACK_EN。启用后允许在间接跳转指令前使用3EH前缀，表示仅这一条跳转指令不需要endbr检查。（直接跳转指令，比如Far Jump，不能通过3EH前缀跳过检查）

endbr检查的可能情况汇总如下：

Table 18-1. Indirect Branch Tracking State Machine

| Current State | Trigger | Next State |
|---|---|---|
| TRACKER=IDLE, SUPPRESS=0, ENDBR_EN=1 | Instructions other than indirect CALL/JMP or 3EH prefixed near indirect CALL/JMP and NO_TRACK_EN=1 | TRACKER=IDLE, SUPPRESS=0, ENDBR_EN=1 |
| | Indirect CALL/JMP without 3EH prefix<br>Indirect CALL/JMP with 3EH prefix and NO_TRACK_EN=0<br>Far CALL/JMP | TRACKER=WAIT_FOR_ENDBRANCH, SUPPRESS=0, ENDBR_EN=1 |
| TRACKER= WAIT_FOR_ENDBRANCH, SUPPRESS=0, ENDBR_EN=1 | INT3/INT1 | TRACKER= WAIT_FOR_ENDBRANCH, SUPPRESS=0, ENDBR_EN=1 |
| | ENDBRANCH instruction | TRACKER=IDLE, SUPPRESS=0, ENDBR_EN=1 |
| | Successful ENCLU[ERESUME] | TRACKER=IDLE, SUPPRESS=0, ENDBR_EN=1 |
| | Instructions other than ENDBRANCH, successful ENCLU[ERESUME] or INT3 or INT1 | If legacy compatibility treatment is not enabled or if not allowed by legacy code page bitmap:<br>▪ No state change and deliver #CP (ENDBRANCH)<br>If legacy compatibility treatment is enabled and transfer allowed by legacy code page bitmap:<br>▪ TRACKER=IDLE, SUPRESS=!SUPPRESS_DIS, ENDBR_EN=1 |
| TRACKER=x, SUPPRESS=x, ENDBR_EN=0 | All instructions | TRACKER=x, SUPPRESS=x, ENDBR_EN=0 |
| TRACKER=IDLE, SUPPRESS=1, ENDBR_EN=1 | Far CALL/JMP, INTn/INT3/INTO | TRACKER=WAIT_FOR_ENDBRANCH, SUPPRESS=0, ENDBR_EN=1 |
| | ENDBRANCH instruction<br>Successful ENCLU[ERESUME] | TRACKER=IDLE, SUPPRESS=0, ENDBR_EN=1 |
| | All other instructions including indirect CALL/JMP | TRACKER=IDLE, SUPPRESS=1, ENDBR_EN=1 |
| TRACKER=1, SUPPRESS=1, ENDBR_EN=1 (This state cannot be reached by hardware and is disallowed as a valid state by WRMSR/XRSTORS/VM entry/VM exit) | NA | NA |

在Ubuntu 18.04 + GCC 9.2.0编译环境下，ENDBR保护默认启用，如有需要可以通过`-fcf-protection=none`参数手动关闭。

### 2.1.2 测试ENDBR保护的效果

笔者构造了如下样例：

```
1 // endbr.c
2 #include <stdio.h>
3
4 void foo() {
5     puts("inside foo");
6     return;
7 }
8
9 int main()
```

```
{
    int(*func_p)() = foo;
    (*func_p)();
    // (*(func_p + 0X04))();
    // (*(func_p + 0x05))();

    return 0;
}
```

在Ubuntu 18.04 + GCC 9.2.0环境下使用 `gcc test.c -o test.exe -fno-stack-protector -g` 命令编译，再使用 `objdump --disassemble --show-raw-insn filename` 查看foo函数的机器码：

```
1     1149:       f3 0f 1e fa                 endbr64
2     114d:       55                          push   %rbp
3     114e:       48 89 e5                    mov    %rsp,%rbp
4     1151:       48 8d 05 ac 0e 00 00        lea    0xeac(%rip),%rax        # 2
5     1158:       48 89 c7                    mov    %rax,%rdi
6     115b:       e8 f0 fe ff ff              call   1050 <puts@plt>
7     1160:       90                          nop
8     1161:       5d                          pop    %rbp
9     1162:       c3                          ret
```

当调用 `(*func_p)();` 时，输出inside foo：

```
➜ 3200102708 ./test.exe
inside foo
```

但令人奇怪的是，调用 `(*(func_p + 4))();` ，即跳过endbr64直接从114dH位置开始执行时，仍然能够输出inside foo。**猜想这是因为测试使用的电脑是AMD处理器，不支持Intel CET规范，因此endbr64会被作为nop执行。**

而当调用 `(*(func_p + 5))();` ，即跳过endbr64与push，从114eH位置开始执行时，因为返回地址没有被push，报出SIGSEGV错误。

```
0x0000555555555187      12          (*(func_p + 5))();
(gdb)
0x000055555555514e in foo () at test.c:4
4       void foo() {
(gdb)
5           puts("inside foo");
(gdb) ni
0x0000555555555158      5           puts("inside foo");
(gdb)
0x000055555555515b      5           puts("inside foo");
(gdb)

Program received signal SIGSEGV, Segmentation fault.
```

通过gdb单步调试可见， `(*(func_p + 5))();` 正常进入了foo程序，而在pop、ret时报出段错误。

## 2.2 Canary保护

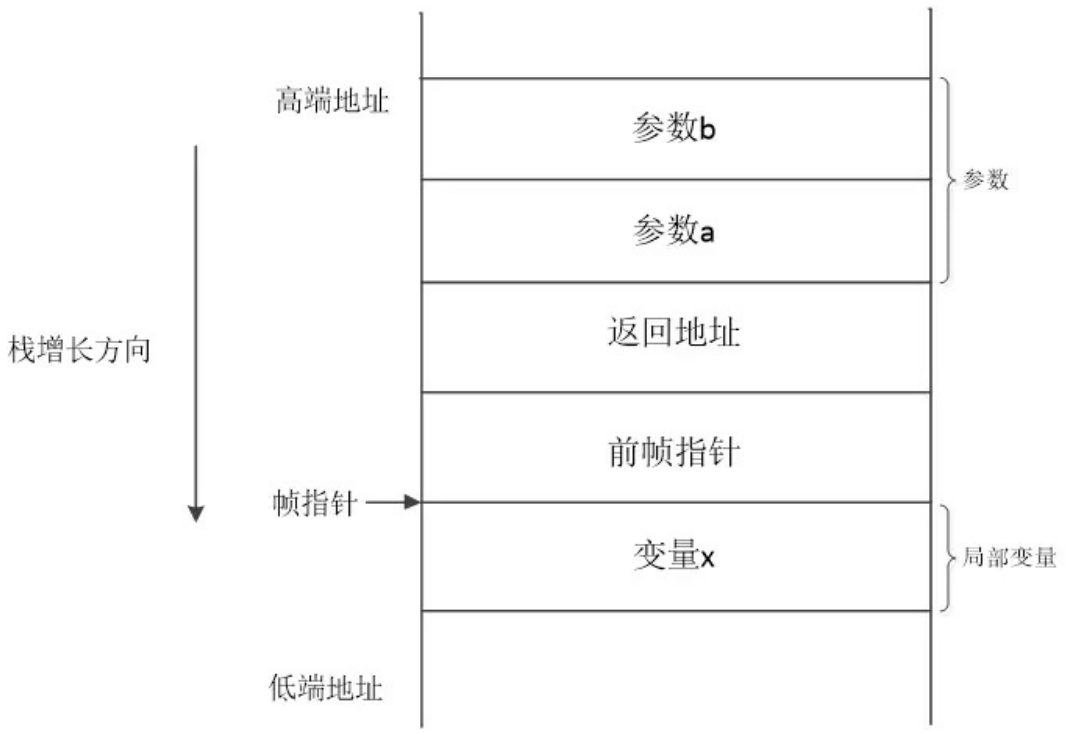
### 2.2.1 原理探究

通过 `objdump --disassemble --show-raw-insn filename` 命令查看完整机器码，得到 `__stack_chk_fail@plt` 程序段。

```
1 <__cxa_finalize@plt>:
2   f3 0f 1e fa             endbr64
3   f2 ff 25 ad 2f 00 00    bnd jmpq *0x2fad(%rip)
4   # 3ff8 <__cxa_finalize@GLIBC_2.2.5>
5   0f 1f 44 00 00          nopl   0x0(%rax,%rax,1)
```

可见其是由GLIBC库所支持的函数，猜想其实现与编译器相关。查询gcc.gnu.org官网后得到了关键词 `Canary保护` 。继续搜集信息后了解到Canary保护的原理如下。

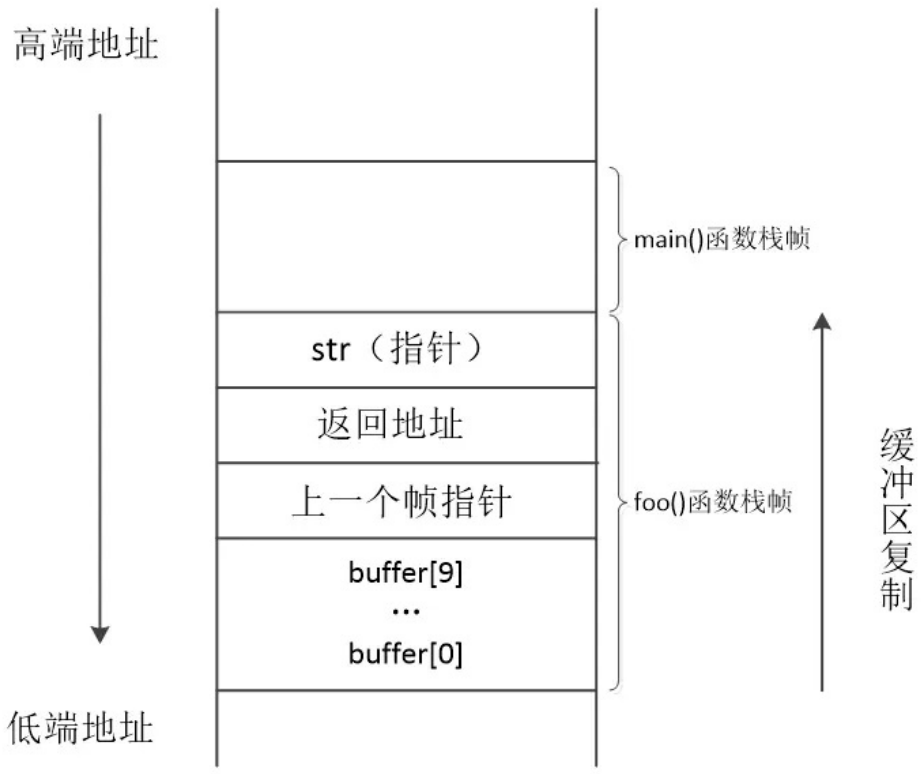
X86架构程序的栈结构如下图所示，而局部变量区中如果有数组等结构存在，它的增长方向可以是从低地址向高地址；那么如果发生数组越界，局部变量就有可能覆盖高地址的返回地址、前帧指针等重要信息。




对下面这个具体的例子：

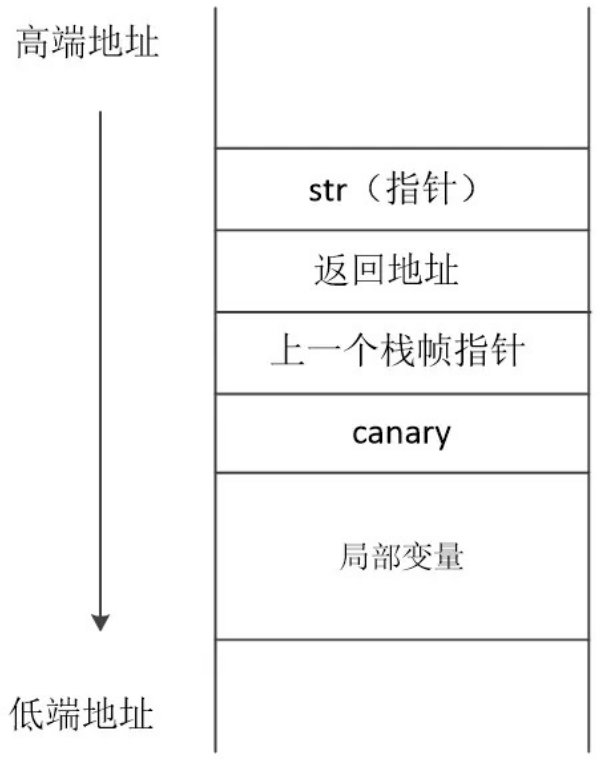
```
1  // canary.c
2  #include <string.h>
3
4  void foo(char *str) {
5      char buffer[10];
6      strcpy(buffer, str);
7  }
8
9  int main(int argc, char **argv) {
10     char str[100] = {...};
11     foo(str);
12     puts("return successfully");
13     return 0;
14 }
```

其栈的结构可以是：




那么当buffer发生写入溢出时，就可能覆盖所有原有的返回地址，从而使程序跳转到任意代码段。

而Canary保护机制通过在返回地址与缓冲区之间添加一个“哨兵”（随机数），在退出代码段时检测“哨兵”是否被修改来判断是否发生了缓冲区溢出。Canary随机数是根据本机周围环境的噪声信息利用SHA、MD5等哈希算法产生的。




在Ubuntu 18.04 + GCC 9.2.0编译环境下，Canary保护默认启用，如有需要可以通过 `-fno-stack-protector` 参数手动关闭。

### 2.2.2 Linux平台汇编分析

在Ubuntu 18.04 + GCC 9.2.0环境下使用 `gcc test.c -o test.exe -g` 命令编译如下样例代码：

```
#include <string.h>

void foo(char *str) {
    char buffer[0x10];
    strcpy(buffer, str);
}

```

```c
int main(int argc, char **argv) {
    char str[100] = {...};
    foo(str);
    puts("return successfully");
    return 0;
}
```

使用 `objdump --disassemble --show-raw-insn filename` 查看foo函数的机器码：

```
0000000000001189 <foo>:
    1189:       f3 0f 1e fa                 endbr64
    118d:       55                          push   %rbp
    118e:       48 89 e5                    mov    %rsp,%rbp
    1191:       48 83 ec 30                 sub    $0x30,%rsp
    1195:       48 89 7d d8                 mov    %rdi,-0x28(%rbp)
    1199:       64 48 8b 04 25 28 00        mov    %fs:0x28,%rax
    11a0:       00 00
    11a2:       48 89 45 f8                 mov    %rax,-0x8(%rbp)
    11a6:       31 c0                       xor    %eax,%eax
    11a8:       48 8b 55 d8                 mov    -0x28(%rbp),%rdx
    11ac:       48 8d 45 e0                 lea    -0x20(%rbp),%rax
    11b0:       48 89 d6                    mov    %rdx,%rsi
    11b3:       48 89 c7                    mov    %rax,%rdi
    11b6:       e8 b5 fe ff ff              callq  1070 <strcpy@plt>
    11bb:       90                          nop
    11bc:       48 8b 45 f8                 mov    -0x8(%rbp),%rax
    11c0:       64 48 33 04 25 28 00        xor    %fs:0x28,%rax
    11c7:       00 00
    11c9:       74 05                       je     11d0 <foo+0x47>
    11cb:       e8 c0 fe ff ff              callq  1090 <__stack_chk_fail@plt>
    11d0:       c9                          leaveq
    11d1:       c3                          retq
```

比对前一节中的栈结构图可以确定：

- -0x28(%rbp)位置存放函数参数，即str指针
- -0x20(%rbp)位置起存放函数的局部变量，即buffer，随下标向高地址方向增长
- -0x08(%rbp)位置存放Canary数

且注意到如下细节：

- 根据Linux手册，64位模式下 `%fs:0x28` 内存位置由操作系统负责维护，实时根据程序上下文生成伪随机数用于栈保护检验；32位模式下则改为 `%fs:0x14` 位置。因此可以观察到在默认设置下，Canary数总是从 `%fs:0x28` 取出，笔者已经通过各种样例确认了这一点。

  > On x86_64, segmented addressing is no longer used, but the both the `FS` and `GS` registers can be used as base-pointer addresses in order to access special operating system data-structures. So what you're seeing is a value loaded at an offset from the value held in the `FS` register, and not bit manipulation of the contents of the `FS` register.
  >
  > Specifically what's taking place, is that `FS:0x28` on Linux is storing a special sentinel stack-guard value, and the code is performing a stack-guard check.

- 这里buffer的大小定义为0x10，但是其与Canery数之间相隔0x18字节，两者并不是紧密相接的。但是如果将buffer大小改为10，编译器又会将buffer起始地址设为 `-0x12(%rbp)` ，两者重新相接。笔者猜想此处可能与内存对齐有关。

### 2.2.3 Windows平台汇编分析

在Windows11 22H2 + GCC 6.3.0环境下使用 `gcc test3.c -o test3.exe -fstack-protector-all -g` 命令编译同一段代码。注意不同于Linux，在Windows平台下GCC的栈保护默认关闭，我们需要在编译时使用 `-fstack-protector` 参数手动启用。使用gdb调试查看foo函数的反汇编如下：

```
=> 0x00401460 <+0>:     push   %ebp
   0x00401461 <+1>:     mov    %esp,%ebp
   0x00401463 <+3>:     sub    $0x48,%esp
   0x00401466 <+6>:     mov    0x8(%ebp),%eax
   0x00401469 <+9>:     mov    %eax,-0x2c(%ebp)
   0x0040146c <+12>:    mov    0x68cc6020,%eax
   0x00401471 <+4>:     mov    %eax,-0xc(%ebp)
   0x00401474 <+7>:     xor    %eax,%eax
   0x00401476 <+9>:     mov    -0x2c(%ebp),%eax
   0x00401479 <+12>:    mov    %eax,0x4(%esp)
   0x0040147d <+16>:    lea    -0x1c(%ebp),%eax
   0x00401480 <+19>:    mov    %eax,(%esp)
   0x00401483 <+22>:    call   0x403b20 <strcpy>
   0x00401488 <+27>:    nop
   0x00401489 <+28>:    mov    -0xc(%ebp),%eax
   0x0040148c <+31>:    xor    0x68cc6020,%eax
   0x00401492 <+4>:     je     0x401499 <_fu1____stack_chk_guard+11>
   0x00401494 <+6>:     call   0x401568 <__stack_chk_fail>
   0x00401499 <+11>:    leave
   0x0040149a <+12>:    ret
```

其与Linux平台相似之处有：

- 栈结构相同：
  - -0x2c(%ebp)位置存放函数参数，即str指针
  - -0x1c(%ebp)位置起存放函数的局部变量，即buffer，也向高地址方向增长
  - -0x0c(%ebp)位置存放Canary数
- Canary检测的原理和时机相同，都是在函数返回之前将程序栈上的值取出与原值进行比较。

但是Windows平台也有与Linux不同之处：

- 根据GCC官网的描述，Windows操作系统也维护有用于栈保护的伪随机数，存放于TLS块的特定结构体当中。因此可以观察到在Windows平台下，取用Canary数的位置和寻址方式都发生了变化。在本例中，Canary数来自硬编码的 `0x68cc6020` 位置。经过测试，对同一段代码，不同时刻编译运行，Canary数的来源可能改变，这与Windows系统的多线程实现机制有关。

  > Generate stack protection code using canary at guard. Supported locations are ‘global’ for global canary or ‘tls’ for per-thread canary in the TLS block (the default).

- Windows平台下buffer与Canary数前后相接，中间没有留空。这再次佐证了Linux平台下buffer与Canary之间留有间隔是出于内存对齐的猜想。

### 2.2.4 测试Canary保护的效果

同样对如下样例进行测试：

```
#include <string.h>

void foo(char *str) {
```

```
    char buffer[0x10];
    strcpy(buffer, str);
}

int main(int argc, char **argv) {
    char str[100] = {...};
    foo(str);
    puts("return successfully");
    return 0;
}
```

前文已经分析过，可能出于内存对齐的原因，在linux平台下编译时buffer与Canary数之间留有8字节的间隔。因此当str如下初始化时：

```
char str[100] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
                 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 0};
```

可以观察到尽管事实上buffer已经发生了越界，但函数仍然正常返回。

```
➜ 3200102708 gcc test.c -o test.exe -g
test.c: In function ‘main’:
test.c:15:5: warning: implicit declaration of function ‘puts’ [-Wimplicit-function-declaration]
   15 |     puts("return successfully");
      |     ^~~~
➜ 3200102708 ./test.exe
return successfully
```

如果继续加大str的长度：

```
char str[100] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
                 11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
                 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 0};
```

可以观察到程序异常退出并报stack smashing detected，表示检测到了栈溢出覆写。

```
➜ 3200102708 gcc test.c -o test.exe -g
test.c: In function ‘main’:
test.c:16:5: warning: implicit declaration of function ‘puts’ [-Wimplicit-function-declaration]
   16 |     puts("return successfully");
      |     ^~~~
➜ 3200102708 ./test.exe
*** stack smashing detected ***: terminated
[1]    11471 abort      ./test.exe
```

上述测试结果说明了，至少在GCC编译器下，Canary保护并不是类似Array.get()函数一样，严格的下标检查，**程序员不能依赖该机制作为避免所有下标越界的手段**。Canary保护更本质的作用是避免因数组越界读写没有访问权限的内存位置。

### 2.2.5 自定义Canary数

如果攻击者能够通过某种手段获取到了Canary数，那么他只要巧妙的设计上面例子中的str，使其覆写内存时将Canary数保持原样，就能绕过Canary防护继续劫持函数返回地址。因此Canary防护生效的前提是其本身不被泄露。

在查阅GCC编译器手册时发现了如下编译参数：

> `-mstack-protector-guard=`*`guard`*
>
> `-mstack-protector-guard-reg=`*`reg`*

> `-mstack-protector-guard-offset=offset`
>
> Generate stack protection code using canary at *guard*. This option has effect only when -fstack-protector or -fstack-protector-all is specified.
>
> With the latter choice the options -mstack-protector-guard-reg=reg and -mstack-protector-guard-offset=offset furthermore specify which segment register ( `%fs` or `%gs` ) to use as base register for reading the canary, and from what offset from that base register. The default for those is as specified in the relevant ABI.

即除使用操作系统自动维护的栈保护随机数（例如Linux下的 `%fs:0x28` 和 `%fs:0x14` ）之外，程序员还可以使用私有的算法生成Canary数放置于内存的特定位置，然后通过编译参数手动指定存放Canary数的基址寄存器和偏移量，将栈检查引向正确的位置。以上三条命令使Canary数的位置和生成算法都不再随运行环境而固定，无疑进一步提高了安全性。

同一段C语言测试样例，在Ubuntu 18.04 + GCC 9.2.0环境下使用 `gcc test3.c -o test3.exe -mstack-protector-guard-reg=gs -mstack-protector-guard-offset=0x08 -g` 命令编译，再次查看foo()函数的汇编代码：

```
0000000000001189 <foo>:
    1189:       f3 0f 1e fa                 endbr64
    118d:       55                          push   %rbp
    118e:       48 89 e5                    mov    %rsp,%rbp
    1191:       48 83 ec 30                 sub    $0x30,%rsp
    1195:       48 89 7d d8                 mov    %rdi,-0x28(%rbp)
    1199:       65 48 8b 04 25 08 00        mov    %gs:0x8,%rax
    // the same as previous one
    11bc:       48 8b 45 f8                 mov    -0x8(%rbp),%rax
    11c0:       65 48 33 04 25 08 00        xor    %gs:0x8,%rax
    11c7:       00 00
    11c9:       74 05                       je     11d0 <foo+0x47>
    11cb:       e8 c0 fe ff ff              callq  1090 <__stack_chk_fail@plt>
    11d0:       c9                          leaveq
    11d1:       c3                          retq
```

可见压栈时进行了 `mov %gs:0x8,%rax` 与 `mov %rax,-0x8(%rbp)` ；返回前进行了 `xor %gs:0x8,%rax` 。这标志Canary数的来源已经不是默认的 `%fs:0x28` ，而是手动指定的 `%gs:0x8` 。

## 三、效果分析

实验仔细分析了Canary防护的实现机制与实际防护效果，并通过翻阅GCC编译命令找到了进一步增强其保护能力的方法；也从汇编代码的分析中总结了这种防护方式应用的局限性与可能的攻击手段。

出于硬件所限，对ENDBR防护只能停留在理论分析，无法实机演示。

## 四、实验体会

本次实验中，我最大的体会是计算机学科的庞大与个人知识的局限。看似简单的一条指令，背后的相关知识量是极其庞大的。

例如为了学习Canary防护的原理和实现行为，我首先需要学习x86函数调用和栈帧原理的知识；为了分析栈结构我又必须学习rbp、rsp、fs、gs等等寄存器的专门含义，以及call、leave、ret等等指令在x86架构下的具体行为；很多汇编语句的行为含义又与其所运行的环境有关系，因此我还需要了解Linux与Windows这两大操作系统的一些异同。

为了在探究中控制变量，我需要仔细阅读GCC手册，学习如何以正确的姿势手敲编译命令；而为了可控的观察汇编指令的内容和运行过程，我还需要学习objdump、gdb等底层开发调试相关工具的使用。

以上提到的诸多相关内容，可能在实验报告中仅仅占有一句话的篇幅，但的的确确消耗了我非常多的时间。

说回实验本身。本次实验也让我明白动手的重要性。譬如Windows与Linux默认行为不一致、Canary数与局部变量之间可能留有空隙等有趣的细节，恐怕没有教科书会详细的列举，只能通过亲身的实验来收获。

## 五、经验教训

**官方手册是个好东西。**对计算机专业领域的问题，尤其是明确指向编译器、操作系统等具体事物的问题，在搜索引擎上漫无目的翻阅相互重复还夹杂各种错误的资料远不如直接查询官方手册来得高效。本次实验中许多问题都是通过官方手册解决的。而为了能高效阅读官方手册，英语能力也同样重要。

## 六、参考文献


gcc.gnu.org

ubuntu.com

https://launchpad.net/~ubuntu-toolchain-r/+archive/ubuntu/test/

《2019-Beginning x64 Assembly Programming》

《Intel® 64 and IA-32 Architectures Software Developer’ s Manual》