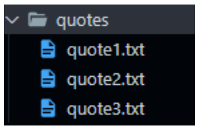
Level 1

We are presented with the following code:

```
import subprocess
from flask import Flask, request

app = Flask(__name__)

@app.route('/')
def index():
    return '''
    <h1>VSauce Quotes</h1>
    <p>You can choose from 3 text files which contain VSauce quotes.</p>
    <p>quote1.txt, quote2.txt, and quote3.txt</p>
    <form action="/read" method="POST">
        <label>Quote file to read: <input type="text" name="quote"/></label><br>
        <input type="submit" value="Read quote"/>
    </form>
    '''

@app.route('/read', methods=['POST'])
def read():
    selected_quote = request.form['quote']

    linux_command = f'cat quotes/{selected_quote}' # create the linux command to find the specific word(s) in the text provided
    result = subprocess.run(linux_command, stdout=subprocess.PIPE, stderr=subprocess.PIPE, shell=True) # run the linux command on the serve

    if result.returncode != 0: # if there was an error when the linux command was executed
        return result.stderr, 500

    return result.stdout.decode('utf-8') # return the output of the ran linux command

if __name__ == '__main__':
    app.run(host='0.0.0.0', debug=False)
```

This Python program creates a web application for reading VSauce quotes. Crazy, right? Anyways, on the URL `/` there is a web form where you input a selected quote, which is the name of one of the three text files which hold the quotes. You can see the quote files in the `/quotes` folder.



Whenever we submit this form, the selected quote filename we inputted is sent to the `/read` URL through an HTTP POST request.

If we take a look at the following lines in the code:

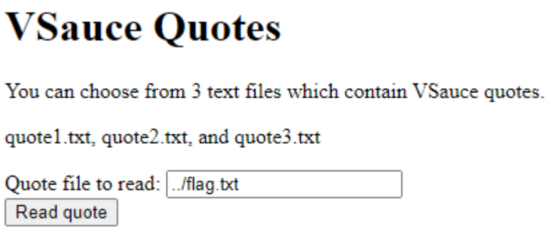
```
linux_command = f'cat quotes/{selected_quote}'
result = subprocess.run(linux_command, stdout=subprocess.PIPE, stderr=subprocess.PIPE, shell=True)
```

We can see that a string called `linux_command` is created by adding the string "`cat quotes/`" with the selected quote filename. For example, if the selected quote filename was `quote1.txt` the `linux_command` variable would equal "`cat quotes/`**`quote1.txt`**".

This string is then ran as a Linux command on the server.

The Exploit

The goal of this challenge is to get the contents of the `flag.txt` file. What would happen if I inputted “`../flag.txt`" into the input bar?



VSauce Quotes

You can choose from 3 text files which contain VSauce quotes.

quote1.txt, quote2.txt, and quote3.txt

Quote file to read: ../flag.txt

Let’s track our input in the app’s code.

We know that whatever we input in the app will be sent to `/read` via an HTTP request. So “`../flag.txt`" will be sent to `/read`.

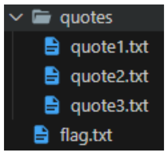
```
@app.route('/read', methods=['POST'])
def read():
    selected_quote = request.form['quote']

    linux_command = f'cat quotes/{selected_quote}'
    result = subprocess.run(linux_command, stdout=subprocess.PIPE, stderr=subprocess.PIPE, shell=True)

    if result.returncode != 0:
        return result.stderr, 500

    return result.stdout.decode('utf-8')
```

It will then be stored in the `selected_quote` variable. A new variable `linux_command` is defined. It concatenates the strings “`cat quotes/`” and the value of `selected_quote` which is “`../flag.txt`".

This will result in `linux_command` having the value “`cat quotes/../flag.txt`".

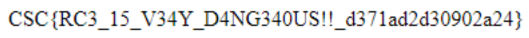
As a result, “`cat quotes/../flag.txt`" is then ran as a Linux command on the server. The command’s output is then stored to a variable called `result` and is then printed to the screen.

Since the `flag.txt` file is outside of the `quotes/` directory by one step, the Linux command “`cat quotes/../flag.txt`" will successfully get the contents of the `flag.txt` and will print it to the screen.



```
quotes
    quote1.txt
    quote2.txt
    quote3.txt
flag.txt
```

As a result, the web app was tricked into giving us the flag!



CSC{RC3_15_V34Y_D4NG340US!!_d371ad2d30902a24}

We essentially manipulated the Linux command being ran by the application into doing whatever we chose. This is why it is VERY important to NEVER include any form of user input inside a string which will be ran as a command.

What to take away from this challenge (optional):

Vulnerabilities which allow you to execute arbitrary code or operating system commands on a server are called Remote Code Execution (RCE) vulnerabilities and are almost always considered severe. This web-application of course, suffers from an RCE vulnerability. This vulnerability does not just allow you to read other files on the server, but it allows you to execute any other command of your choosing onto the server.

You will see this in the solutions for level 2 and 3.

Not all RCE vulnerabilities are as straightforward as this. They can sometimes become quite complex and hard for developers to notice when developing an application.

For example, **Log4shell** is a very well known Remote Code Execution vulnerability. It's a vulnerability in Log4j, a Java library used almost everywhere. Upwards to 3 billion devices were affected by this vulnerability.

Useful links:

- https://www.ncsc.gov.uk/information/log4j-vulnerability-what-everyone-needs-to-know
- https://youtu.be/0-abhd-CLwQ

There are so many other RCE vulnerabilities found in different products. If you are interested, I recommend browsing Google to learn about them.