Holden Stefan
8/31/2021

Demonstrating Insecure Interaction Between Components

Example 1: CWE-434: Unrestricted Upload of File with Dangerous Type

Overview:

Lack of restrictions on file upload can be a very dangerous vulnerability, leading to quite a few other possible vulnerabilities, including Cross-Site Scripting, Command Injection or Remote Code Execution, or leaking of sensitive data among others. The main reason that this can be so devastating is that attacks can be perpetuated on the server *or* client side of the application. The application that I wrote is a basic Python web app, using the Flask framework. It consists of three pages: a home page, a file upload page, and a page to view the uploads.

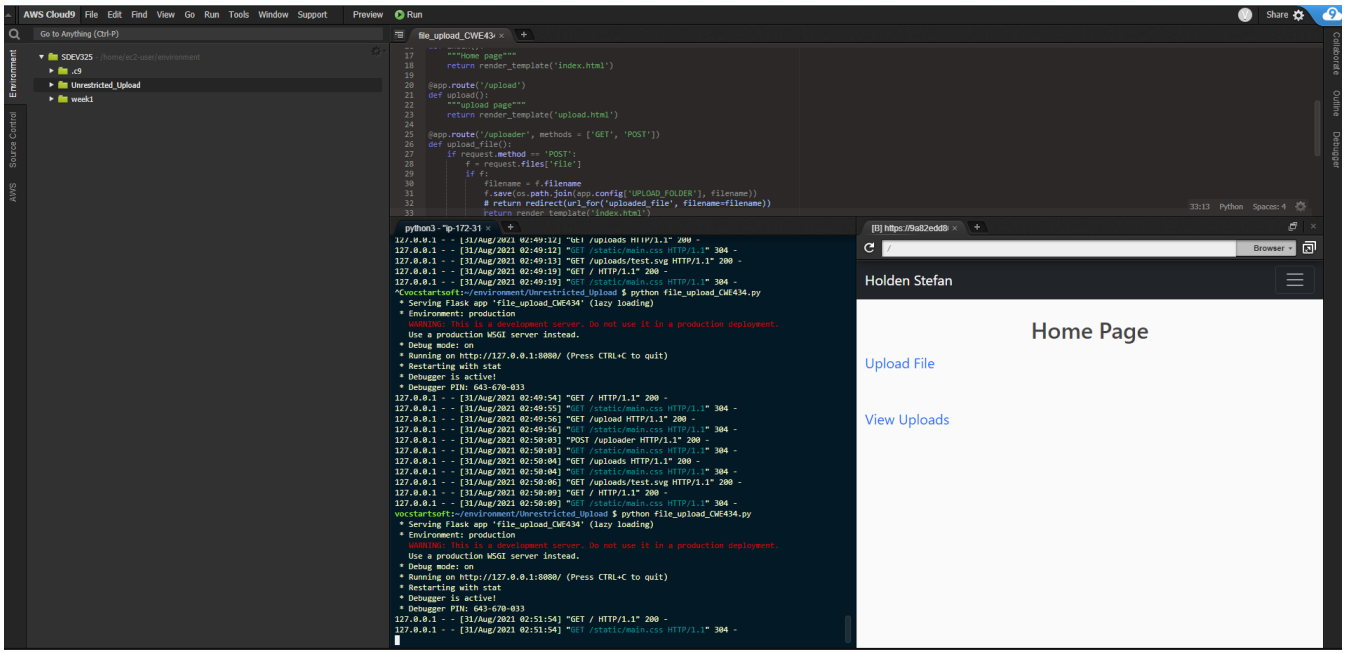


Image 1: Demonstration of application running in Cloud9

Analysis:

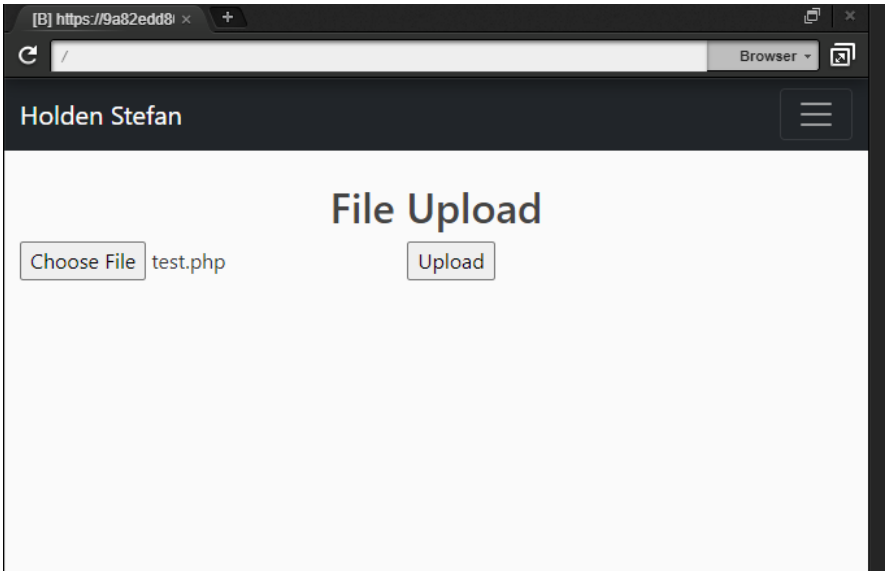
The vulnerability was fairly easy to create once the application was set up. In the upload page, the user is presented with standard file selector and submit buttons, and there are simply zero checks on the file that was selected for upload.

Sample code from the upload route which contains the function:

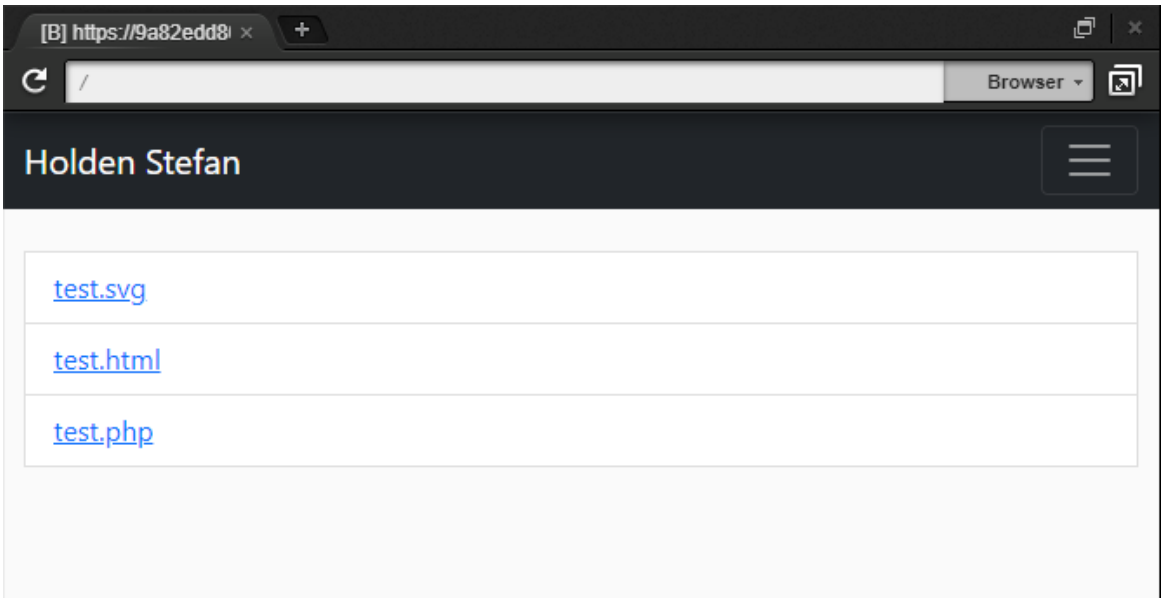
```
@app.route('/uploader', methods = ['GET', 'POST'])
def upload_file():
   if request.method == 'POST':
      f = request.files['file']
      if f:
         filename = f.filename
```

```
    f.save(os.path.join(app.config['UPLOAD_FOLDER'], filename))
    # return redirect(url_for('uploaded_file', filename=filename))
    return render_template('index.html')
```

The selected file is simply saved into the designated ‘UPLOAD FOLDER’ location.



In the images to the left, you can see the upload page, as well as the page to view all of the uploads, which are clearly of dangerous types (.svg, .html, and .php)



These uploaded files could have been expected to be profile images, or pdf files, which the developer intended to be harmless, but could contain essentially anything, including malicious scripts, or even just malicious names that may be executed in the browser from the url.

Mitigation

In order to mitigate this, checks had to be put in place before any files were actually uploaded to the server. The simplest way to do this is to enforce a strict allow-list policy, only allowing the minimum amount of acceptable file types. If images are expected for example, then only allow .jpg or .png extensions. Furthermore, it is important to parse the file names correctly so that a php file cannot be hidden in a name such as “malicious.php.jpg” for example. Lastly, it is a good idea to rename the files altogether, so that no malicious code will ever show up in the url.

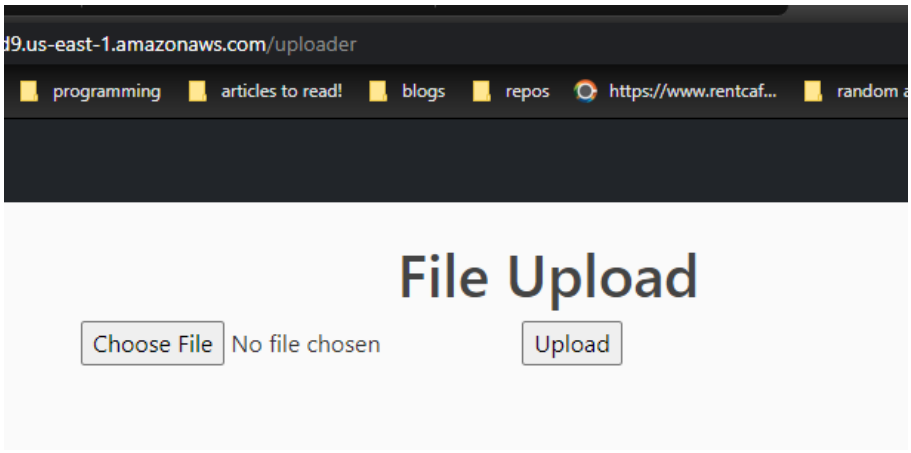
Fixed code:

```
ALLOWED_EXTENSIONS = {'txt', 'pdf', 'png', 'jpg', 'jpeg'}

def allowed_files(filename):
   return '.' in filename and \
      filename.rsplit('.', 1)[1].lower() in ALLOWED_EXTENSIONS

@app.route('/uploader', methods = ['GET', 'POST'])
def upload_file():
   if request.method == 'POST':
      f = request.files['file']
      if f and allowed_files(f.filename):
         # f_name, f_extension = os.path.splitext(str(f))
         f_extension = pathlib.Path(f.filename).suffix
         filename = secure_filename(str(datetime.now().strftime("%H:%M:%S.%f")[:-4]))+f_extension
         f.save(os.path.join(app.config['UPLOAD_FOLDER'], filename))
         # return redirect(url_for('uploaded_file', filename=filename))
         return render_template('index.html')
      return render_template('upload.html')
```

In the code above, there is a strict whitelist for allowed extensions, as well as a parser to extract and compare the uploaded files’ extensions to those in the whitelist. Furthermore, there is a renaming aspect to the function, where the name gets converted into a datetime string, and a “secure_filename” method is used on that, which swaps many escape characters for underscores. Further protection could have been added by uploading to a completely different location that could have acted as a sandbox, and checking the MIME type of the file in comparison to the extension, to make sure that nothing was hidden within an accepted filetype. In this mitigation, a user can select a malicious file, but upon clicking submit, nothing gets uploaded and they are redirected to the upload page.



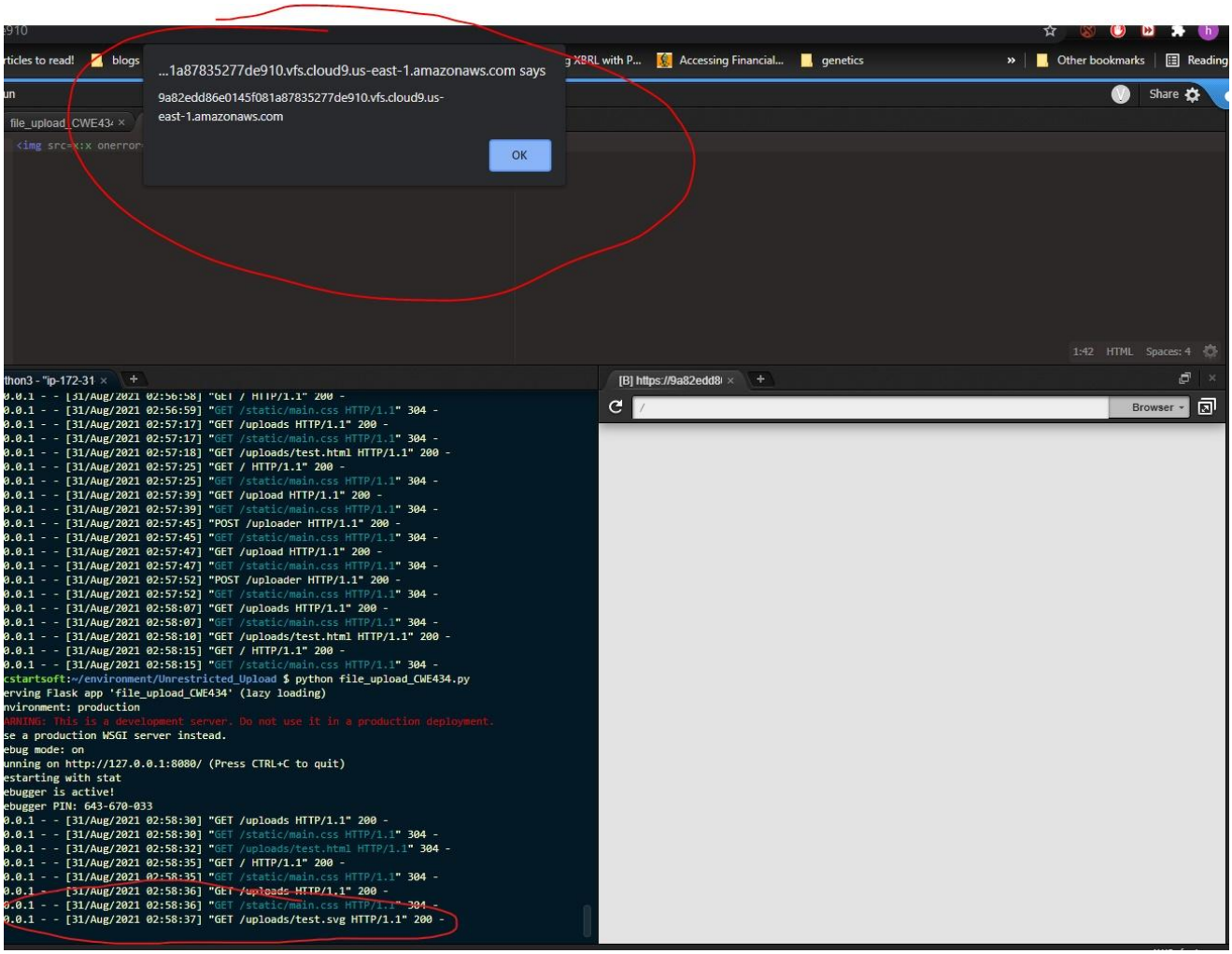
Example 2: CWE-79 : Improper Neutralization of Input During Web Page Generation ('XSS')

Overview:

Cross-site scripting consists of malicious data being loaded onto a webpage. This data could be Javascript, HTML, Flash, etc. The main motivation for an attacker to do this is because all browsers contain scripting engines, and therefore the opportunities are often prevalent, and if done effectively, can be used to conduct either phishing campaigns or if the XSS is stored, watering hole attacks. The Application that I created is a Python-Flask application which stores user's files on the server, and allows other users to view those files. It is the same application shown above, but vulnerable to malicious input on page generation.

Analysis:

The application does not perform any sanitization of filenames or file contents/extensions, and therefore any file can be uploaded, including javascript, svg, html, etc. An svg file is commonly used for storing images (scalable vector graphics), but can contain code. In this case, an svg file containing `<svg xmlns="http://www.w3.org/2000/svg" onload="alert(document.domain)"/>` can be uploaded and stored on the server. When another user browses to that destination, they are susceptible to any malicious script used instead of the "alert(document.domain)".



In the image above, the browser was directed to the “/uploads/test.svg” file. And once the page is loaded, we see the alert pop up, containing the domain of the document (circled in red).

Mitigation:

In order to mitigate this, many things could be and should be done. As XSS is one of the most prevalent vulnerabilities, there are many strategies for mitigation as only a few will not solve the problem at every attack surface. In this case, simply blocking file uploads with extensions such as SVG, JS, or HTML, will solve the problem. Furthermore, changing the input values of the filenames to a strictly numerical value, such as the datetime, can block scripts from being executed from url parameters.

```
ALLOWED_EXTENSIONS = {'txt', 'pdf', 'png', 'jpg', 'jpeg'}

def allowed_files(filename):
   return '.' in filename and \
      filename.rsplit('.', 1)[1].lower() in ALLOWED_EXTENSIONS
```

```
@app.route('/uploader', methods = ['GET', 'POST'])
def upload_file():
   if request.method == 'POST':
      f = request.files['file']
      if f and allowed_files(f.filename):
         # f_name, f_extension = os.path.splitext(str(f))
         f_extension = pathlib.Path(f.filename).suffix
         filename = secure_filename(str(datetime.now().strftime("%H:%M:%S.%f")[:-4]))+f_extension
         f.save(os.path.join(app.config['UPLOAD_FOLDER'], filename))
         # return redirect(url_for('uploaded_file', filename=filename))
         return render_template('index.html')
      return render_template('upload.html')
```

Many more mitigations exist, and while not all are relevant to every application, enforcing request header parameters such as HttpOnly, or allowed data types is a very good idea, and can at least help to protect other users' session information.

References

MITRE. (2006, July 19). *Common weakness enumeration*. CWE. https://cwe.mitre.org/data/definitions/434.html.

MITRE. (2006, July 19). *Common weakness enumeration*. CWE. https://cwe.mitre.org/data/definitions/79.html.