The nVisium Blog (/blog/)

Injecting Flask

Published on December 7, 2015 by Ryan Reid (/about#RyanReid)

In this adventure we will discuss some of the security features available and potential issues within the Flask micro-framework (http://flask.pocoo.org/docs/0.10/) with respect to Server-Side Template Injection, Cross-Site Scripting, and HTML attribute injection attacks, a subset of XSS. If you've never had the pleasure of working with Flask, you're in for a treat. Flask is a lightweight python framework that provides a simple yet powerful and extensible structure (it is Python (https://xkcd.com/353/) after all).

Let's talk about injection

For its presentation layer, Flask leverages the Jinga2 (http://jinja.pocoo.org/docs/dev/) engine. It's easy to use and is configured out-of-the-box to autoescape content in `.html`, `.htm`, `.xml`, and `.xhtml` files. Flask allows for the creation of templates using strings of HTML in the Python source code or laid out in static files in a templates directory local to your project.

Server-Side Template Injection

The template engine provided within the Flask framework may allow developers to introduce Server-Side Template Injection vulnerabilities. If you're unfamiliar check out the whitepaper (https://www.blackhat.com/docs/us-15/materials/us-15-Kettle-Server-Side-Template-Injection-RCE-For-The-Modern-Web-App-wp.pdf)(*PDF*) by James Kettle. Briefly, this vulnerability allows an attacker to inject language/syntax into templates. Execution of this input occurs within the context of the server. Depending on the context of the application this could allow for arbitrary remote code execution (RCE). Let's take a look at using the template string functionality to explore some security concerns. Consider the following snippet of code:

```
from flask import Flask, request, render_template_string, render_template

app = Flask(__name__)

@app.route('/hello-template-injection')
def hello_ssti():
    person = {'name':"world", 'secret':"UGhldmJoZj8gYWl2ZnZoei5wYnovcG5lcnJlZg=="}
    if request.args.get('name'):
        person['name'] = request.args.get('name')
    template = '''<h2>Hello %s!</h2>''' % person['name']
    return render_template_string(template, person=person)

####
# Private function if the user has local files.
###
def get_user_file(f_name):
    with open(f_name) as f:
        return f.readlines()

app.jinja_env.globals['get_user_file'] = get_user_file # Allows for use in Jinja2 templates

if __name__ == "__main__":
    app.run(debug=True)
```

Now, let's navigate to the application:

Great success! What about a named guest? I'm no Bobby Tables (https://xkcd.com/327/) but I've got a few nicknames. Let's try one:

```
Ryan. {{person.secret}}
```

Oh no! Our code just shared the secret! Surely that's the worst of it, right? Let's try another payload, just in case. Fun fact, I happen to know the environment's administrator stores their secrets in the tmp directory. The get_user_file method looks pretty interesting. Maybe we can use it.

```
Ryan. {{ get_user_file("/tmp/secrets.txt") }}
```

Okay, Local File Inclusion (LFI) through the template? This is bad. How do we fix it? The issue arises due to the use of string concatenation or substitution. If you're a Flask developer you probably already know the answer. Jinja2 uses curly braces `{{}}` to surround variables used in the template. By placing our output inside of these braces we will prevent user entered data containing template syntax from executing within the context of our server.

With the SSTI fix in place the full line now reads:

```
template = '<h2>Hello {{ person.name }}!</h2>'
```

There we have it; Server-Side Template Injection mitigated.

Cross-Site Scripting

As stated above, Flask provides an autoescape feature on certain file types. While this is excellent there are some caveats:

- Templates can disable this feature
- Template strings and non-common file extensions do not enable autoescape by default

Take a look at our fix from the last section. Does it protect us against XSS?

Let's try a common XSS test string:

```
Ryan. <script>alert("Full protection not yet achieved.")</script>
```

Remember I said template strings don't autoescape? Well there we have it. To fix the issue, we can pipe our output through the manual escape filter `|e` to ensure proper output escaping before reflection to the user. So our final template string will appear as:

```
template = '<h2>Hello {{ person.name | e }}!</h2>'
```

Now, not every application is going to use on-the-fly templates. So what about more traditional Cross-Site Scripting attacks in the static templates? Consider the following function:

```
def hello_xss():
    name = "world"
    template = 'hello.unsafe' # 'unsafe' file extension... totally legit.
    if request.args.get('name'):
        name = request.args.get('name')
    return render_template(template, name=name)
```

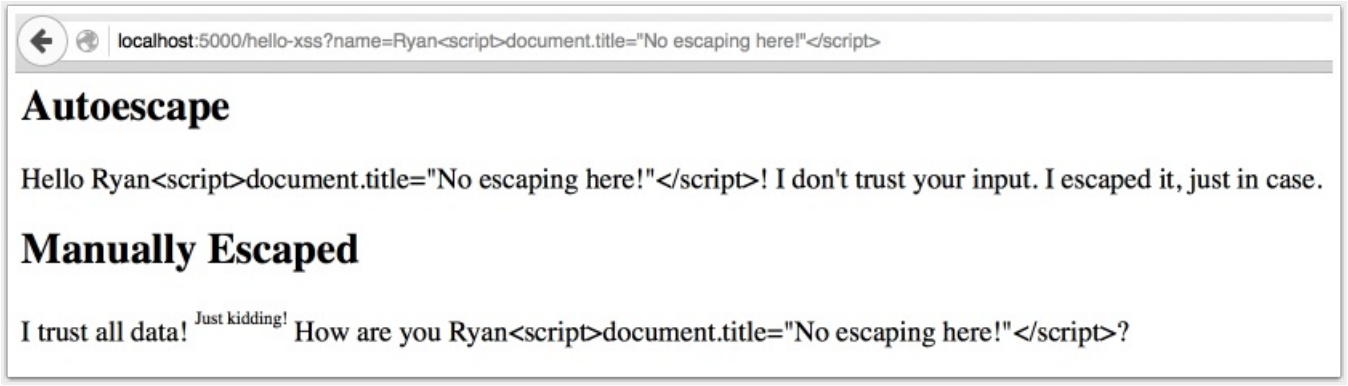
Note: the Python code calls `render_template` with a template that isn't an autoescaped file extension. Depending on the code in the template, `hello.unsafe`, we may be vulnerable to Cross-Site Scripting. Here's the template code:

```
{% autoescape true %}
<h2>Good</h2>
<p>
    Hello {{ name }}! I don't trust your input. I escaped it, just in case.
</p>
{% endautoescape %}
<h2>Bad</h2>
<p>
    I trust all data! How are you {{ name }}?
</p>
```

Let's test:

Interesting, the autoescaped block works as expected; we appropriately escaped the output. However, the second section allowed for the injected payload to execute in the browser.

While the "Good" section leveraged the autoescape function within the Jinga2 engine, we could have also leveraged the `|e` filter as we had in the SSTI context. Here's the output using the `|e` filter on the "Bad" section:



All done, right? Well, no, not just yet.

Remarks about escaping output in Flask

The escaping function doesn't protect against HTML attribute injection. Using the following code as an example:

```
def hello_hi():
    template = '''<title>No Injection Allowed!</title>
        <a href={{ url_for('hello_xss')}}?name={{ name |e}}>
        Click here for a welcome message</a>'''
    name = "world"
    if request.args.get('name'):
        name = request.args.get('name')
    return render_template_string(template, name=name)
```

We can see that we are surrounding our variable with `{{}}` and using the `|e` filter to manually escape output, we should be safe from injection, right?

Now, let's just hover over that link:

Uh oh, the payload executed. First, the problem: our injected payload executed due to the name parameter appearing in the context of an HTML attribute. Second, the fix, encapsulating output in an attribute context in single/double quotes will resolve this issue. The updated link tag:

```
<a href='{{ url_for('hello_xss')}}?name={{ name |e}}'>...</a>
```

Wrapping up

We've taken a look at some of the features provided in Flask for output escaping, the potential issues, and the fixes available should you come across some vulnerable code. Remember, always escape your output but also validate your input! Most names don't include less than and/or greater than symbols. At least in this example, you'd probably be safe with a whitelist and logic to reject input containing special characters.

python (/blog/tags/python/) flask (/blog/tags/flask/)
server-side-template-injection (/blog/tags/server-side-template-injection/) Ryan Reid (/blog/tags/ryan-reid/)

Next: **What to Expect When You're Overriding** (/blog/2015/12/21/what-to-expect-when-you-re-overriding/)

Previous: **Introducing Httpillage** (/blog/2015/11/11/introducing-httpillage/)

Show discussions

(https://twitter.com/nvisium)

(https://www.facebook.com/nVisium)

(https://github.com/nVisium)

(https://www.linkedin.com/company/2422009)