Lab Exercise xss

This is a lab exercise on developing secure software. For more information, see the introduction to the labs.

1. Task

Please practice countering Cross-Site Scripting (XSS) with Flask and the templating engine Jinja2.

2. Background

In this exercise, we'll implement mechanisms to broadly counter Cross-Site Scripting (XSS) attacks, as described in our course. We noted that "The standard way to counter XSS is to escape all output that might be from an attacker and is not specifically approved. ... In most cases, the best solution for XSS is to choose a framework or library that automatically escapes HTML output for you." That is, we want to use a system that automatically escapes characters like "<" by converting them into "&lt;" unless we specifically note otherwise. That way, those special characters are rendered harmless.

In *theory* you could call an escape routine every time you make a call to generate an output. In *practice* this approach is insecure. Sooner or later a developer will accidentally forget to call the escape routine while generating output. It's much safer to use mechanisms which escape *by default*.

Flask is a lightweight server-side web application framework for the Python programming language. Programs using Flask often use the Jinja2 template library. Jinja2 has a mechanism for automatically escaping HTML.

Jinja2's version 3.1.x series does *not* enable this "autoescape" mode by default. This may change in a future version of Jinja, but even so, this serves as a great example. In short, sometimes libraries must be specially configured to be less dangerous to use. This isn't ideal, but such libraries can still be used. You simply need to ensure that you *correctly* configure the library to be used securely.

Flask by default configures Jinja2 to automatically escape HTML. So as far as users of *Flask* are concerned, the Jinja templating system *does* automatically escape HTML by default.

3. Task Information

Please change the code below to counter XSS vulnerabilities. Use the “hint” and “give up” buttons if necessary. This is a multi-part task; the specific task instructions are interspersed below.

4. Interactive Lab (COMPLETE!)

4.1. Part 1

In this part, you are *not* using Flask. You're instead using Python and directly using the Jinja2 templating engine. The version of Jinja2 we're considering does *not* automatically escape unless it's configured to do so. Modify the Jinja2 configuration to automatically escape HTML, by passing the field named `autoescape` with `select_autoescape()` as its value.

```
from jinja2 import Environment, PackageLoader, select_autoescape
env = Environment(
    loader=PackageLoader("yourapp"),
autoescape=select_autoescape()
)
```

Hint Reset Give up

4.2. Part 2

In this part we *are* using Flask. Flask configures Jinja2 to automatically escape HTML. The code below (based on the Flask quickstart) uses Flask's template rendering system, which in turn calls Jinja2 to render the result:

```
from flask import render_template
@app.route('/hello/')
@app.route('/hello/')
def hello(name=None):
    return render_template('hello.html', person=name)
```

The sample code above uses the template `hello.html`, which is shown below. Note that values to be substituted in the template are surrounded by `{{ ... }}`.

Unfortunately, this template below has a vulnerability. Its "| safe" marking tells the templating system that the data is safe and shouldn't be escaped. However, as shown in the code above, the person's name is from an untrusted user. Thus the person's name (as with most data) is *not* safe. Currently an attacker can slip characters like "<" into a name as a way to attack others. Please fix this vulnerability.

```
<!doctype html>
<title>Hello from Flask</title>
```

```
{% if person %}
  <h1>Hello {{ person }}!</h1>
{% endif %}
```

Hint Reset Give up

4.3. Part 3

In this part we continue to use Flask. However, sometimes you need more sophisticated control over what is and is not escaped. Most web application frameworks have a type or class that records HTML values, and lets you specify what is to be escaped or not. In Flask this typically done with its Markup class.

A instance of a `Markup` class is created by calling `Markup`. A string is passed during its original construction is assumed to be safe and is *not* escaped. You can concatenate a normal string to a Markup value, and those additions *will* be escaped. For example, computing `Markup("<em>Hello</em> ") + "<foo>"` produces a Markup instance containing the Unicode string value `'<em>Hello</em> &lt;foo&gt;'` - note how the first part isn't escaped but the latter part *is* escaped. You can even create a Markup instance with an empty string, and every concatenation of a normal string will be escaped. Since every concatenation to a Markup value of a normal string will be escaped by default, the default is the safe (escaping) operation. The code also clearly indicates what is considered safe and what is not. The Markup class supports many other methods not described here to simplify control over what is escaped. The templating system will directly include an instance of Markup without further escapes, because the Markup instance has already escaped whatever is supposed to be escaped.

The Python code below tries to use `Markup` to include a `name`. However, it's incorrect and insecure. The problem is that `name` is an untrusted value and it shouldn't be passed directly to `Markup` as part of its trusted value. Modify the code below so that `result` will include the *escaped* version of `name`. Note that in Python `+` is the string concatenation operation.

```
  result = Markup('Original name=' )+ name
```

Hint Reset Give up

This lab was developed by David A. Wheeler at The Linux Foundation.

Completed 2025-03-06T21:34:42.830Z b6ed8fad-646a-4715-a789-60862df728b9 0q1o5ne191hcqr