proj2: Vulnerability Patching

How to setup and run? (FOR ALL LEVEL)

💡 The following command line are for mac and python 3. In case some of the command may not work on your computer, I comment the purpose of each command, so you can also search the specific command for your computer.

- set up virtual env

```
// create a new env
python3 -m venv myenv

// activate the virtualenv
source myenv/bin/activate

// install requirements
pip install -r requirements.txt
```

- run the application

```
// go to the level folder (level1 - level6)
cd level1

// set up flask app
export FLASK_APP=level
export FLASK_ENV=development

// run the app
flask run
```

- go to http://127.0.0.1:5000/

Level 1: Hello, world of XSS

- The original vulnerability is caused by directly concatenating “query” input to the html. We can patch this vulnerability by adding “query” to inner HTML {{query}}.

Since HTML will prevent script tag in innerHTML from executing, inputting <script> alert(1) <script> won't work.

```
<div>
  Sorry, no results were found for <b> {{query}} </b>. <a href='?'>Try again</a>.
</div>
```

Level 2: Persistence is key

- The original vulnerability is caused by the onerror attribute of image tag. Therefore, we can patch this vulnerability by removing all onerror attribute in the input.

```
html += "<blockquote>" + cleanInput(posts[i].message) + "</blockquote>";
...
function cleanInput(input) {
  return input.replace(/onerror=['"].*['"]/, "");
}
```

Level 3: That sinking feeling...

- The original vulnerability is cause by inputting the request in the URL directly into the html, while we only want a number fro the URL request. Therefore, we can only parse an integer in the input and put the integer into html.

```
let index = parseInt(num);
var html = "Image " + index + "<br>";
html += "<img src='https://xss-game.appspot.com/static/level3/cloud" + index + ".jpg' />";
$('#tabContent').html(html);
```

Level 4: Context matters

- The original vulnerability is caused by unformatted input to the input box. We can patch this bug by checking if the input is a valid integer before passing it to the timer.

```
// level.py
if (request.args.get('timer')):
    # get timer value from request
    if (request.args.get('timer').isdigit()):
        time = int(request.args.get('timer'))
        if (time > 0):
          return render_template("timer.html", timer=str(time))
    return render_template("timer.html", timer="-1")

// timer.html
if (seconds == "-1") {
  window.confirm("Invalid timer value.");
  window.history.back();
  return;
}
```

Level 5: Breaking protocol

- The original vulnerability is caused by redirecting the website to execute malicious code. However, since we know which page is the next page for the current page. We can control the flow so user input won't affect the website's behaviors.

```
def signup():
    next_url = request.args.get('next')
    if next_url != "confirm":
        return render_template("signup.html", next="confirm")
    return render_template("signup.html", next=request.args.get('next'))

def confirm():
    next_url = request.args.get('next')
    if next_url != "welcome":
        return render_template("confirm.html", next="welcome")
    return render_template("confirm.html", next=request.args.get('next'))
```

Level 6: Follow the 🐇

- Allowing user input to influence the URL when loading scripts or other potentially dangerous types of data often leads to serious vulnerabilities. So we can block javascript from loading external data by adding a CSP header.

```
<meta http-equiv="Content-Security-Policy" content="script-src 'self'">
```