

CSCE 413: Software Security
PoC 3

Development of Vulnerable Application (30 Points)

For this assignment, I have created a simple Flask blogging web app that is vulnerable to Stored and Reflected XSS attacks. To run both vulnerable and secure versions of the web application, please follow these steps for Linux;

1. Enter the PoC3 folder.
`cd PoC3`
2. Create a Python virtual environment.
`python3 -m venv .venv`
3. Activate the virtual environment.
`source ./venv/bin/activate`
4. Install all requirements.
`pip install -r ./requirements.txt`
5. Run either the vulnerable or secure versions of the program.
`python3 ./vulnerableWebapp.py`
`python3 ./secureWebapp.py`

The application has two main routes, `localhost:5000/post` and `localhost:5000/search` for creating blog posts and searching through blog posts, respectively. What follows is a brief analysis of its structure and why it is vulnerable.

Stored XSS Vulnerability

Note: Flask uses Jinja2 as its templating engine which automatically escapes HTML characters. I have used the "safe" parameter to prevent it from doing this to HTML templates for demonstration purposes.

Stored XSS vulnerabilities can be found in both `vulnerableWebapp.py` and the templates `blog_post.html` and `blog_search.html`. The Python code fails to protect against Stored XSS because it does not sanitize or escape user input before dynamically loading it into HTML. Instances of this can be found on lines 19 and 43 of `vulnerableWebapp.py`. The HTML templates fail to protect against this by directly displaying the variables provided by the Python script and using the `safe` parameter which disables Jinja2's HTML character escaping. Instances of this can be found on lines 21 and 24 of `blog_post.html` and `blog_search.html`, respectively.

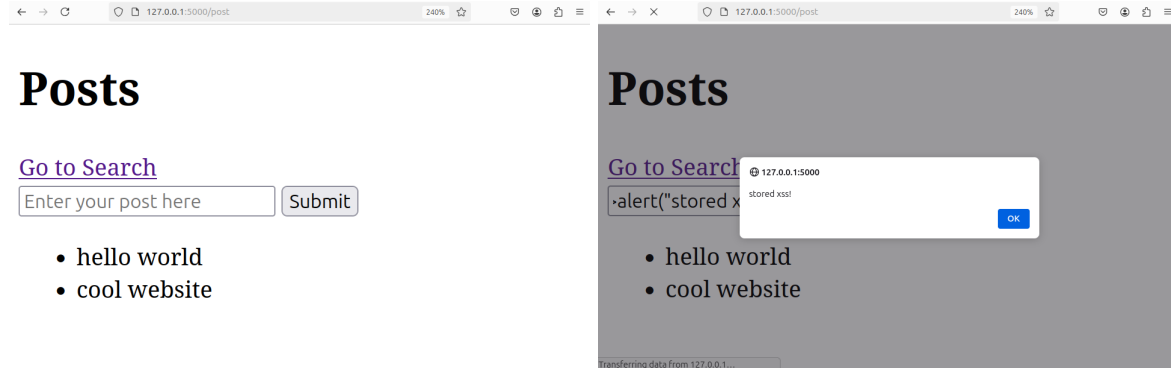
Reflected XSS Vulnerability

A Reflected XSS vulnerability exists in the `/search` route. The `/search` route accepts a query parameter in the URL, `/search?query=...`, and displays the search parameter underneath the search bar via the statement, "Posts containing: ...". A malicious script within the URL will then execute once the page is loaded, creating a Reflection XSS attack. While this vulnerability is primarily because input is not sanitized (as seen on line 20 of `vulnerableWebapp.py`), it is also because the developers chose to explicitly display the search parameter as seen on line 19 of `blog_search.html`.

Exploitation of Vulnerabilities (20 Points)

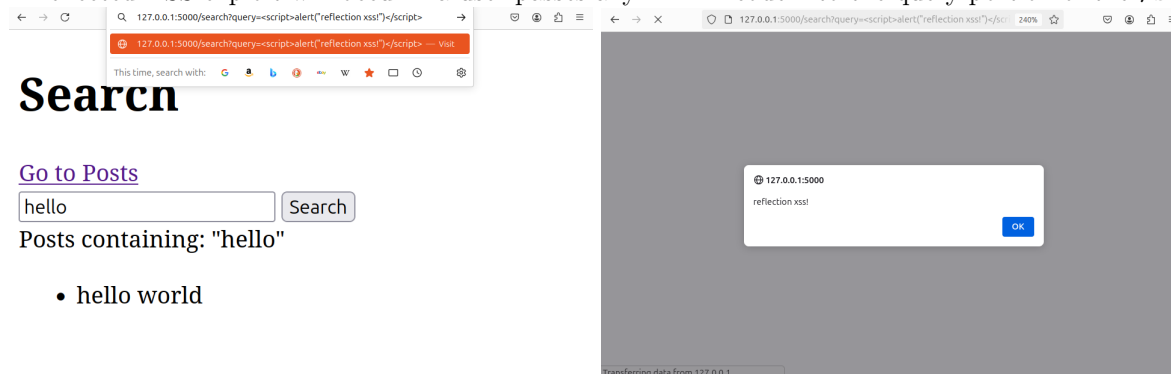
Stored XSS Exploit

A stored XSS exploit will occur if a user passes any HTML code into the "Enter your post here" form in the /post route.



Reflected XSS Exploit

A reflected XSS exploit will occur if a user passes any HTML code into the query portion of the /search route.



Sanitization of User Input (20 Points)

In order to sanitize user input, I created a function called `escape_html_string` which can be found on line 9 of `secureWebapp.py`.

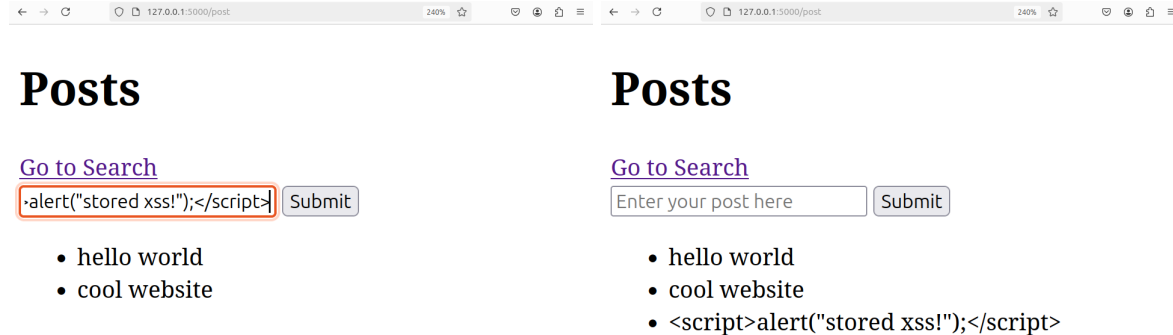
```
1 def escape_html_string(input_string):
2     # A mapping of all characters to their equivalent, HTML-safe counterparts
3     equivalent_characters = {
4         '&': '&amp;',
5         '<': '&lt;',
6         '>': '&gt;',
7         '"': '&quot;',
8         "'": '&#39;',
9     }
10
11     # Replace every character of the string with their HTML equivalent. If there is no
12     # equivalent, return the normal char
13     escaped_string = "".join(equivalent_characters.get(char, char) for char in
14                               input_string)
15     return escaped_string
```

This simple Python function replaces all characters with their HTML counterparts, removing the functionality of any scripts or other HTML elements passed as input. The `escape_html_string` function is then called before any posts are stored to prevent stored XSS attacks and before displaying any URL search queries to prevent reflection XSS attacks. These corrections can be found in `secureWebapp.py` on lines 35, 59, and 62.

Demonstration of Fix Effectiveness (20 Points)

Stored XSS Fix

All attempts at injecting stored XSS are now escaped and presented as normal text. Here is the same attempted exploit as seen earlier.



Posts

[Go to Search](#)

- hello world
- cool website

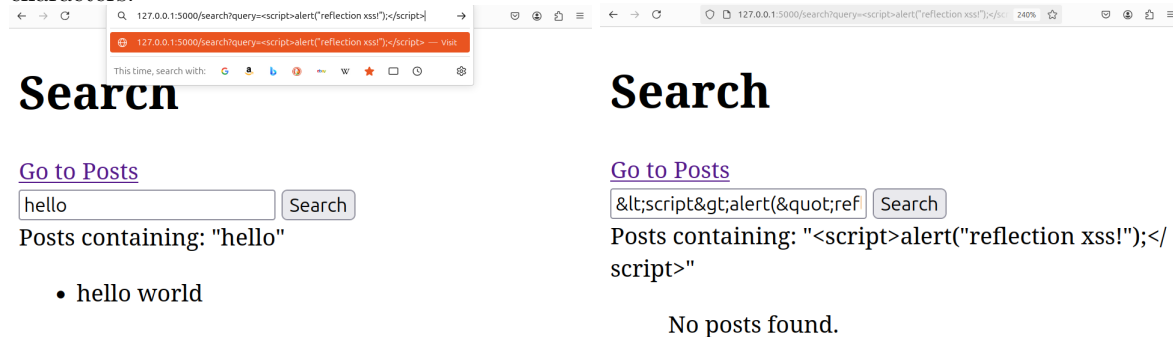
Posts

[Go to Search](#)

- hello world
- cool website
- <script>alert("stored xss!");</script>

Reflected XSS Fix

All attempts at manipulating the query string to reflect XSS are now escaped, presenting the alternate HTML characters.



Search

[Go to Posts](#)

Posts containing: "hello"

- hello world

Search

[Go to Posts](#)

Posts containing: "<script>alert('reflection xss!');</script>"

No posts found.