

Reflected XSS Attack Report

Disclaimer: All presented attacks were executed against a test application for educational purposes only. The author of this report is not responsible for any illegal actions that readers of this report may take against real (production) systems. Use the information provided responsibly and with care!

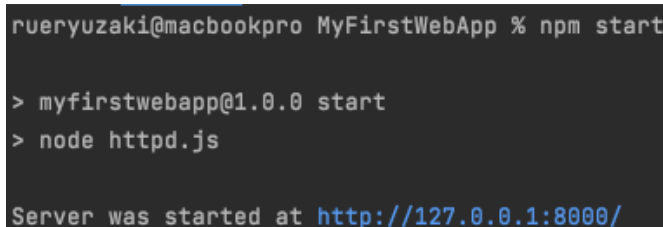
Author: Adil Nurmanov (a.nurmanov1998@gmail.com).

Abstract: Reflected cross site scripting (XSS) attack is a type of attack where malicious user supplied input is reflected back to the user of the application which results in arbitrary script execution in the victim's browser [1]. In this report, reflected XSS attack is presented with a step-by-step guide in order to reproduce it. After exploiting and understanding the roots of the vulnerabilities patches will be presented in order to eliminate them. Code samples will be run on the NodeJS platform but the same vulnerabilities may present in other platforms as well.

Prerequisites:

- Source code for target applications (includes both unpatched and patched versions) [3].
- NodeJS runtime.
- Npm.

To start a sample application, download source code for an unpatched app, open the terminal, go inside the *MyFirstWebApp* directory and run the “*npm start*” command. If command was successful than output should look like this:



```
rueryuzaki@macbookpro MyFirstWebApp % npm start
> myfirstwebapp@1.0.0 start
> node httpd.js

Server was started at http://127.0.0.1:8000/
```

Picture 1 - Application run

1. Inserting untrusted user data into HTML content

When untrusted user data is included into HTML content generated by the server it may result in code injection attack(for example, XSS attack).

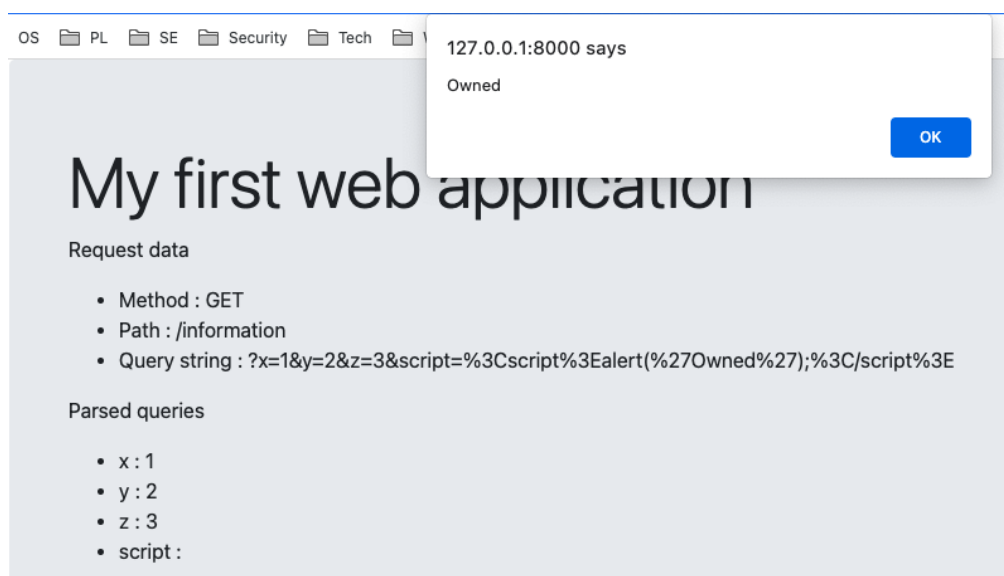
Exploit: If we try to access test application using the provided url <http://127.0.0.1:8000/information?x=1&y=2&z=3> we can see a result page like below.



Picture 2 - Information about user request

It looks like this page shows the HTTP method, requested path and query string information. We assume that the application prints all arbitrary parameters that the user can supply.

But what happens if we try to supply parameters like this [http://127.0.0.1:8000/information?x=1&y=2&z=3&script=%3Cscript%3Ealert\(%27Owned%27\);%3C/script%3E](http://127.0.0.1:8000/information?x=1&y=2&z=3&script=%3Cscript%3Ealert(%27Owned%27);%3C/script%3E)



Picture 3 - Passing javascript code as a parameter value

If we decode the supplied url we get something like this [http://127.0.0.1:8000/information?x=1&y=2&z=3&script=<script>alert\('Owned'\);</script>](http://127.0.0.1:8000/information?x=1&y=2&z=3&script=<script>alert('Owned');</script>)

The provided parameter value is a valid javascript code which was successfully executed in the browser after submitting the request.

```
3 <head>
4   <meta charset="utf-8">
5   <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">
6   <link rel="stylesheet" href="css/bootstrap.min.css">
7   <title>Request information</title>
8 </head>
9 <body>
10 <div class="container">
11   <div class="jumbotron">
12     <div class="container">
13       <h1 class="display-4">My first web application</h1>
14       <p>Request data</p>
15       <ul>
16         <li>Method : GET</li>
17         <li>Path : /information</li>
18         <li>Query string : ?x=1&y=2&z=3&script=%3Cscript%3Ealert(%27Owned%27);%3C/script%3E</li>
19       </ul>
20       <p>Parsed queries</p>
21       <ul>
22         <li>x : 1</li><li>y : 2</li><li>z : 3</li><li>script : <script>alert('Owned');</script></li>
23       </ul>
24     </div>
25   </div>
26 </div>
27 <script src="js/jquery-3.3.1.slim.min.js"></script>
28 <script src="js/popper.min.js"></script>
29 <script src="js/bootstrap.min.js"></script>
30 </body>
31 </html>
```

Picture 4 - View page source

By viewing the page source it is clear that user input was rendered as a valid html tag and its content was executed as a javascript code which provided a windows with 'Owned' message.

This type of attack is called a reflected XSS attack.

Mitigation: The main idea when dealing with untrusted user input like this is to encode it and present it to the end user in a safe manner. Browser treats special constructions like this as DOM elements and tries to render them appropriately. We need to encode the following characters with HTML entity encoding to prevent switching into any execution context, such as script, style, or event handlers. Using hex entities is recommended in the spec [2].

Sadly, javascript does not have a special function for making HTML encoding so we will implement our own.

```

/**
 * Encode HTML entities
 *
 * @param input input
 * @returns encoded html input
 */
function encodeHtmlEntities(input) {
  return input.replace(/([<>"])/g,
    tag => ({
      '&': '&amp;',
      '<': '&lt;',
      '>': '&gt;',
      '"': '&#39;',
      "'": '&quot;',
    })[tag]));
}

```

This function takes input and uses string replace function to replace all special symbols on their hex representations. After encoding these special characters are not treated as special by the browser and can be safely presented to the user. The last part is to detect all parts of our application where user supplied input is shown to the user. Then we use *encodeHtmlEntities* function to encode any user supplied input.

We need to change function *information* because this function is responsible for rendering this page.

```

/**
 * Send information about client request
 *
 * @param req request
 * @param res response
 */
function information(req, res) {
  try {
    const {reqUrl, path} = parseUri(req, res);
    fs.readFile(templateDir + '/information.template', (err, data) => {
      if (err) {
        console.error(err);
        res.statusCode = 500;
        res.end();
      } else {
        let queries = "";
        for (let param in reqUrl.query) {
          queries += `<li>${param} : ${encodeHtmlEntities(reqUrl.query[param])}</li>`;
        }
        res.statusCode = 200;
        res.end(data.toString()
          .replace('{{method}}', encodeHtmlEntities(req.method))

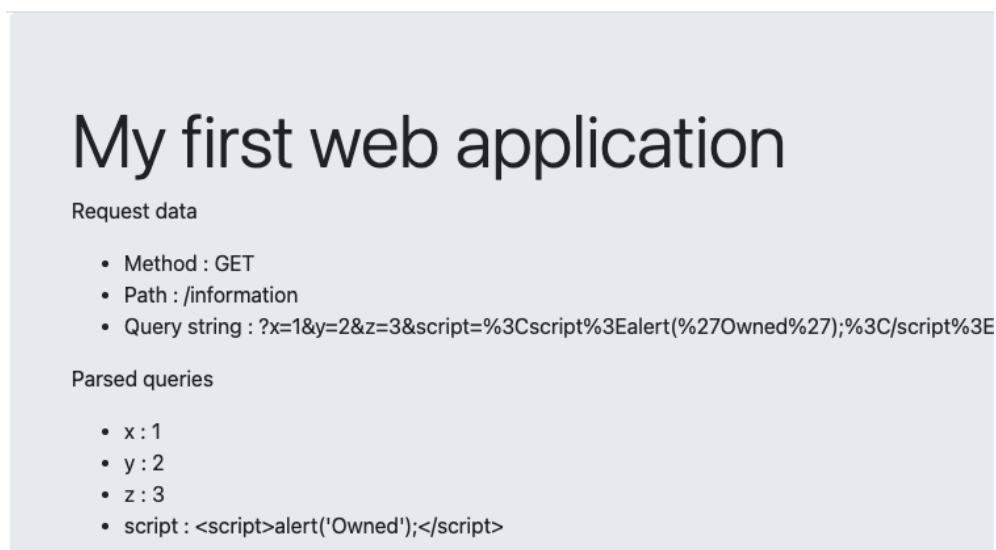
```

```

        .replace('{{path}}', encodeHtmlEntities(path))
        .replace('{{query}}', encodeHtmlEntities(reqUrl.search ?? ''))
        .replace('{{queries}}', queries));
    }
  });
} catch (e) {
  res.statusCode = 400;
  res.end();
}
}
}

```

If we try to submit the previous request again javascript will not be executed.



Picture 5 - Information page

Line wrap ☐

```

1 <!doctype html>
2 <html lang="en">
3 <head>
4   <meta charset="utf-8">
5   <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">
6   <link rel="stylesheet" href="css/bootstrap.min.css">
7   <title>Request information</title>
8 </head>
9 <body>
10  <div class="container">
11    <div class="jumbotron">
12      <div class="container">
13        <h1 class="display-4">My first web application</h1>
14        <p>Request data</p>
15        <ul>
16          <li>Method : GET</li>
17          <li>Path : /information</li>
18          <li>Query string : ?x=1&y=2&z=3&script=%3Cscript%3Ealert(%27Owned%27);%3C/script%3E</li>
19        </ul>
20        <p>Parsed queries</p>
21        <ul>
22          <li>x : 1</li><li>y : 2</li><li>z : 3</li><li>script : &lt;script&gt;alert(&#39;Owned&#39;);&lt;/script&gt;</li>
23        </ul>
24      </div>
25    </div>
26  </div>
27  <script src="js/jquery-3.3.1.slim.min.js"></script>
28  <script src="js/popper.min.js"></script>
29  <script src="js/bootstrap.min.js"></script>
30 </body>
31 </html>

```

Picture 6 - View page source

At this time when we look at the page source we will see that all input that was taken from the user was properly encoded into html hex objects so content is not treated as a special DOM object anymore.

References:

1. <https://owasp.org/www-community/attacks/xss/>
2. https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html#xss-prevention-rules
3. <https://github.com/Shazam2morrow/dva489>