Preface

Somebody asked a question on Security Stack Exchange because they needed help with an XSS challenge. I took a quick look at the code and saw that it was pretty short and looked pretty simple. So even though I'm not an expert on XSS, JS, or PHP, I got curious and decided I wanted to understand what was going on. Especially because the answer on Stack Exchange said it was a *very easy* capture-the-flag challenge, yet I can't solve it. Since I spent more than enough time thinking about this, now I'd really like to know what's wrong with my reasoning, or what kind of skill or knowledge I lack exactly. By the way, I'm not even trying to compete in the challenge, I'm just trying to understand how it works and hopefully learn something new.

The challenge

The challenge can be found at this link. It's a static HTML page with some JS code. There is also a PHP file that accepts a couple of parameters, source code is provided. The page has a form, and if you provide a secret code you will get a link to access a secret page. But the challenge has nothing to do with the secret page: the challenge is to be able to execute **alert(document.domain)** in the page. I am now going to show you the relevant code, and I will add some thoughts in the comments.

HTML and JS code (only the relevant parts)

```
<!-- In the head the encoding is set to utf-8 -->
<meta charset="utf-8">

<!-- I don't see anything wrong with the following form, except there's no token
against CSRF, but CSRF doesn't seem relevant in this case, and the form will be
submitted automatically anyway. -->
<form name="form">
    <input id="c" type="text">
    <input type="submit" value="Submit">
</form>
<div id="result"></div>
<script>

  // MAJOR WEAKNESS HERE: you can use this function to inject HTML code
  var callback = function(msg) {
    result.innerHTML = msg;
  }

  // Nothing seems wrong here: when the page is loaded, it basically does
  // the same thing as submitting the form (= checking the code)
  document.addEventListener('DOMContentLoaded', function(event) {
    if (getQuery('code')) {
      var code = getQuery('code'); // get code from URL
      c.value = code;              // put code in the input field
      checkCode(code);             // check code
    }
  });
```

```
  // Here's what happens when you submit the form: it just checks the code
  form.addEventListener('submit', function(event) {
    checkCode(c.value);
    event.preventDefault();
  });

  // The function that checks the code. It adds a script in the page,
  // and the content of the script comes from vulnerable PHP code (see below)
  function checkCode(code) {
    var s = document.createElement('script');
    // WEAKNESS HERE: encodeURI is wrong, it should be encodeURIComponent.
    // So here you can actually modify every parameter in the query string.
    s.src = `/xss_2020-06/check_code.php?callback=callback&code=${encodeURI(code)}`;
    document.body.appendChild(s);
  }

  // Get a parameter from the query string in the URL of this page
  function getQuery(name) {
    return new URL(location.href).searchParams.get(name);
  }
</script>
```

PHP code (complete code)

```
<?php
$callback = "callback";
if (isset($_GET['callback'])) {
    // $callback can be passed as a parameter, but can only accept [a-zA-Z0-9.]
    $callback = preg_replace("/[^a-z0-9.]+/i", "", $_GET['callback']);
}

$key = "";
if (isset($_GET['code'])) {
    $key = $_GET['code'];
}
// Whatever the key is, you will end up with only three possible cases,
// and you can modify the result in only one of those cases.
// The length is checked in UTF-8, but I see nothing suspicious here,
// since UTF-8 also set in the HTML page and in this PHP file (see below).
if (mb_strlen($key, "UTF-8") <= 10) {
    if ($key == "XSS_ME") {
        // Nothing can be modified here
        $result = "Okay! You can access <a href='#not-implemented'>the secret
                   page</a>!";
    } else {
        // This is the only case where you can modify the result,
        // but remember that $key must be max 10 UTF-8 characters!
        $result = "Invalid code: '$key'";
    }
} else {
    // Nothing can be modified here
    $result = "Invalid code: too long";
}
// $result is a string, so I suppose $json will always be a quoted string too
$json = json_encode($result, JSON_HEX_TAG);

header('X-XSS-Protection: 0');
header('X-Content-Type-Options: nosniff');
header('Content-Type: text/javascript; charset=utf-8'); // UTF-8, as expected
// WEAKNESS HERE: the following line creates JS code where
// $callback and $json can (partially) be modified by the user.
print "$callback($json)";
```

Analysis

The most obvious weakness is the function `callback(msg)` in JS, which would allow injecting HTML in the page without restrictions. However the only way to call it is via the function `checkCode(code)`, which will fetch some JavaScript generated by the PHP script, and then append and execute it. So in the end it looks like it all depends on what the PHP script can return.

The PHP script only accepts two parameters: `code` and `callback`. We can control both of them thanks to the weakness related to `encodeURI(code)`, so if in the browser's address bar we write a query string like `?code=123%26callback%3Dfoo` then the PHP script will return the JS code `foo("Invalid code: '123'")`.

Now, the solution would be very simple if we could execute `callback("<script>...</script>")` or `eval("...")`, but the parameters we can pass to the PHP script actually have some limitations. In `$callback($json)` only the characters [a-zA-Z0-9.] are allowed in the function name, and its only argument is always going to be a string with some limitations. Basically I think you can only get a line of JS code like one of the following ones:

```
someThing.with.limitedChars("Invalid code: too long")
someThing.with.limitedChars("Okay! You can access <a href=...")
someThing.with.limitedChars("Invalid code: 'anything here, but max 10 chars!'")
```

Using `callback(msg)` as the function, you can inject HTML code in the page, but 10 chars are not enough AFAIK for XSS in HTML context. Something like `eval` won't work either, because the first part of the argument will cause a syntax error. In any case 10 chars wouldn't be enough for the desired payload. It looks like I can only call a function that accepts a fixed string as a parameter, and the actual payload should probably come from somewhere else (like the URL), so I realized there's the function `getQuery(name)` in the code that could become useful. I could call `getQuery("Invalid code: 'foo'")` to get a parameter named `Invalid%20code%3A%20%27foo%27` from the query, but that function will just *return* the value of the parameter from the URL, while I would actually want to *execute* it. I would need another piece of code to append in front of `getQuery()` to execute the value it returns, but of course that piece of code can't have parentheses, spaces, or symbols other than dots. This is where I'm stuck.

I don't think there are too many possible reasons why I'm stuck. If I'm on the right track, then I'm stuck because I don't know what JS code I should use in the last step, no matter if it's basic JS or some advanced trick. Otherwise there's something I'm missing, but honestly I don't think there are many other places where things can go wrong. Encoding? It's UTF-8 everywhere (although the content-type HTTP headers for the HTML page are not set). Workarounds on `preg_replace`? But it looks like it also removes null characters and newlines, so it's expected to remove everything. Issues with JSON? But I guess a

string in JSON will always end up being a quoted string anyway. I even tried to pass the parameters as arrays, but all I got was `Array("Invalid code: 'Array"')` . Cool tricks that allow XSS with 10 chars? But according to some answers I read on Stack Exchange, and judging by the cheatsheets I checked, 10 chars are definitely not enough in this case. So after all these thoughts, I think I need to know the solution to this challenge before I go insane.