xss

Release 0.0.1

Jul 12, 2019

Contents:

CHAPTER 1

Cross-Site Scripting

XSS is a vulnerability where a user of an application can send malicious code in the form of a browser side script to some other user of the same application. Flaws that allows these attacks to occur happens whenever a web application uses data taken from users in any way without first validating or encoding it.

It can be broken down into 3 categories.

1.1 Reflected XSS

Reflected XSS is an attack in which injected script is reflected off the web server in the form of a response to the target user. The XSS exploit is provided through a url parameter. Example:

```
https://site.com?data=<script>...</script>
```

If the app is vulnerable it might insert the script in the dom. The script needs to be constructed differently on the basis of where it gets injected.

One of the deficiencies of Reflected XSS is that it is easily detected by the browser and that the user needs to access the vulnerable page from an attacker controlled resource since for the attack to occur data parameter needs to be supplied.

1.2 Stored XSS

In Stored XSS the injected script is stored permanently on the servers of the applications, the exploit is then provided through the website itself.

It can happen if a malicious user is able to inject the exploit into the database of the website which is then served to other users. This is not easily detected by the browser.

A classical example of this is using a img tag as a XSS vector. Say a message board exists to which users of a website can post messages to, now if the website doesn't sanitize the content either on client side or server side then we can inject an exploit in such a manner:

```
<img src=x onerror="__malicious_code__">
```

1.3 DOM XSS

While the server may prevent XSS from its side, it's possible that the client side JS scripts may *accidentally* take a payload and insert it into the DOM and cause the payload to trigger.

That is, the response from the server doesn't change but the client side code executes differently due to some malicious modifications that may have been made to the DOM environment.

CHAPTER 2

XSS Prevention

In an HTML page there are these slots where a developer can put data from an untrusted user. We create a Whitelist model within which we specify slots in which the developer should put untrusted data. Putting data anywhere else is not recommended. Within these slots any data put should be escaped/encoded.

Now one might think that just HTML Encoding the untrusted data should suffice, and it's okay to think that but doing so would handle only a small percentage of XSS attacks that your application might come under. This is because HTML Entity Encoding is okay for untrusted data that might go inside the body of a HTML Entity like a div tag, and it might also work with data that goes into attributes if you put quotes around them. But HTML Entity encoding doesn't work if you are putting untrusted data inside a script tag, or an event handler, or inside CSS, or in URLs. The reason for this is that the browser uses different parsers for different contexts.

Given the way browsers may parse different parts of HTML, each of the different type of slots must be handled differently. When untrusted data is put into these slots you need to make sure that the data does not break out of that slot into a context that allows script execution.

As an Example of this parsing difference consider these two lines of code.

```
<div><script title="</div>">
```

```
<script><div title="</script>">
```

Take a moment to think how the DOM renders when these are encountered. As an hint, even my rst parser doesn't recognize the second one as html.

This is what `<div><script title="</div>">` outputs

```
<div>
    <script title="</div>">
    </script>
</div>
```

This is what `<script><div title="</script>">` outputs

```
<body>"></body>
```

Hopefully it should be apparent by now that you MUST use the escape syntax for the part of the HTML document you're putting untrusted data into since depending on the context different parsers could be used.

Here we only look at how to prevent Reflected and Stored XSS attacks. For DOM XSS you can look at the link provided at the end.

2.1 XSS prevention rules

2.1.1 Rule 0

Don't put **any** untrusted data into your HTML document, unless it is withing one of the slots defined in rules 1 to 5, because there might be some strange contexts for which encoding rules are tricky based on how different browsers handle them.

These strange contexts include nested contexts like a URL inside javascript. Also don't put any untrusted data directly inside a script tag, or inside an HTML comment, or in an attribute or tag name, or directly in CSS.

2.1.2 Rule 1

HTML Escape before inserting Untrusted data into HTML Element Content. One can use HTML entity encoding but using hex codes is recommended in the spec to prevent switching into any execution context such as a script, style, or event handlers.

2.1.3 Rule 2

Attribute Escape before putting untrusted data inside HTML common attributes like `width`, `name`, `value` etc. This rule should not be used for complex attributes like `src`, `href`, etc.

Stress on the word common here as relating to attributes whose contents are not executed.

```
<div attr="...ESCAPE UNTRUSTED DATA BEFORE PUTTING HERE...">content
```

Except for alphanumeric characters encode all other characters with `&#xHH;` format. The reason for this being so broad is that developers often forget to put attribute values in quotes, and quoted characters can only be escaped with a quoting character whereas for unquoted attributes there are many ways to escape.

2.1.4 Rule 3

Javascript Escape before inserting untrusted data into Javascript data values. This concerns both scripts and event-handler attributes. The **only** safe place to put this data is inside a quoted data value since escaping in any other javascript context is very easy.

```
<script>alert('...ESCAPE UNTRUSTED DATA BEFORE PUTTING HERE...')</script>

<script>x='...ESCAPE UNTRUSTED DATA BEFORE PUTTING HERE...'</script>

<div onmouseover="x='...ESCAPE UNTRUSTED DATA BEFORE PUTTING HERE...'"</div>
```

Except for alphanumeric characters escape all others with `\xHH` format. **DO NOT** just escape quote characters like `\"` because the attacker sends `\"` and the vulnerable code turns that into `\\"` which enables the quote.

2.1.5 Rule 4

CSS is surprisingly powerful, and can be used for numerous attacks. Therefore, it's important that you only use untrusted data in a property **value** and not into other places in style data.

CSS Escape and validate before inserting untrusted data into CSS selector values.

```
<span style="property : ...ESCAPE UNTRUSTED DATA BEFORE PUTTING HERE...">text</span>
```

Except for alphanumeric characters, escape all characters with ASCII values less than 256 with the `\HH` format. DO NOT use any escaping shortcuts like `\"`.

Even if you Escape all untrusted data, validation has to be done to ensure that the URLs only start with `http` and not `javascript`.

```
{ background-url : "javascript:alert(1)"; }  // and all other URLs
```

2.1.6 Rule 5

URL Escape before putting untrusted data into HTML URL parameter values. Like when you put data from user in to GET parameters. Except for alphanumeric characters, escape all characters with ASCII values less than 256 with the `%HH` escaping format.

```
<a href="http://www.somesite.com?test=...ESCAPE UNTRUSTED DATA BEFORE PUTTING HERE...
↪">link</a >
```

Do not encode complete or relative URL's with URL encoding! If untrusted input is meant to be placed into href, src or other URL-based attributes, it should be validated to make sure it does not point to an unexpected protocol, especially javascript links.

URL's should then be encoded based on the context of display like any other piece of data. For example, user driven URL's in HREF links should be attribute encoded.

2.1.7 Rule 6

If the data from the user is supposed to contain markup then it is very difficult to validate and encoding is also very difficult as it would break all the tags. There is then a need for a sanitizer library that can parse and clean HTML formatted text. You can look at `DOMPurify`, its link is provided at the end.

2.1.8 Rule 7

Implement Content Security Policy. It's a browser side mechanism that allows you to create source whitelist for client side resources like javascript, images, etc. CSP via special HTTP headers instruct the browser to only execute or render resources from these source.

```
Content-Security-Policy: default-src: 'self'; script-src: 'self' static.domain.tld
```

This header tells the browser to only load resources from the source page and additionally javascript files from `static.domain.tld`

2.2 Security recommendations

- Always sanitize the users' content that comes from forms.
- Always prefer to serialize instead of JSON.stringify.
- Use dangerouslySetInnerHTML only when absolutely necessary.
- Do unit tests for all your components, and try to cover all the possible XSS attacks that some user could do.
- Always encrypt the passwords with sha1 and md5 (together), and also add a salt value (for example, if the password is abc123, then your salt can be encrypted like this: sha1(md5('$4lT3xt_abc123')).
- If you use cookies to store sensitive information (personal information and passwords mainly), you can save the cookie with Base64 to obfuscate the data.
- Add some protection to your APIs (using security tokens) unless you need to have a public API.
- Just because React stops XSS doesn't mean that all code is safe. Be distrustful of all libraries that work outside of React and avoid them if at all possible.
- Wrap all text passed in to dangerouslySetInnerHtml with an XSS filter and create a Lint rule to enforce this in the future.

2.3 Useful Links

https://edx.readthedocs.io/projects/edx-developer-guide/en/latest/preventing_xss/index.html

2.3.1 Sanitizers Pageant

https://www.npmtrends.com/dompurify-vs-sanitize-html-react-vs-xss-vs-bleach

2.3.2 XSS cheatsheets

https://github.com/OWASP/CheatSheetSeries/blob/master/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.md

https://github.com/OWASP/CheatSheetSeries/blob/master/cheatsheets/DOM_based_XSS_Prevention_Cheat_Sheet.md

https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet

2.3.3 Content Security Policy

https://content-security-policy.com/

2.3.4 Browser Side sanitizer

https://github.com/cure53/DOMPurify

2.3.5 Puzzles and Challenges

https://github.com/cure53/XSSChallengeWiki/wiki

2.3.6 DOM Based

https://github.com/WICG/trusted-types

CHAPTER 3

XSS in React

Since React is a library for creating component based interfaces, most of the attacks surfaces in issues related to rendering elements in the DOM.

3.1 JSX Prevents Injection attacks

By default, React DOM escapes any values embedded in JSX before rendering them. Thus it ensures that you can never inject anything that's not explicitly written in your application. Everything is converted to a string before being rendered. This helps prevent XSS (cross-site-scripting) attacks.

```
const title = response.potentiallyMaliciousInput;
const element = <h1>{title}</h1>;
```

According to documentation, Babel compiles JSX down to `React.createElement()` calls.

```
React.createElement(
  type,
  [props],
  [...children]
)
```

`createElement` creates and returns a new React element of the given type, where `props` contains a list of attributes passed to the new element and `children` contains the child node(s) of the new element (which, in turn, are more React components).

So above code may compile to

```
const element = React.createElement(
  'h1',
  {},
  '_escaped_title_'
);
```

Escaping code in React DOM works great when passing a string in `[...children]` as we did with *_escaped_title_*, but the other two arguments, `type` and `props` are passed unescaped.

This could potentially lead to XSS attacks if bad programming practices are used.

3.2 Bad Programming Patterns

- Creating React components from user-supplied objects, i.e. setting the `type` attribute with data supplied by user.
- Explicitly setting the `dangerouslySetInnerHTML` prop of an element;
- Rendering links with user-supplied `href` attributes, or other HTML tags with injectable attributes (*link* tag, HMTL5 imports);
- Passing user-supplied strings to `eval()`.

3.2.1 Controlling Element Type

While creating a dynamic element with type provided by the user isn't on its own harmful since it would only result in a plain attribute-less HTML Element, setting the properties of the newly created element would have dangerous effects.

3.2.2 Injecting Props

Say you have set up the system such that you parse user supplied JSON and then parse it to use the resulting object as props.

```
attacker_props = JSON.parse(stored_value)
React.createElement("span", attacker_props};
```

Here, if the attacker wishes they can use the following payload to set the `dangerouslySetInnerHTML` property. This property is React's replacement for using innerHTML in the browser DOM.

```
{
    "dangerouslySetInnerHTML" : {
        "__html": "<img src=x/onerror='alert(localStorage.access_token)'>"
    }
}
```

3.2.3 dangerouslySetInnerHTML

Avoid this property as much as you can. If need be thoroughly test your app and use preventive measures such as sanitizing both at server side and user side, and use whitelist methods.

```
const aboutUserText = "<img onerror='alert(\"Hacked!\");' src='invalid-image' />";

class AboutUserComponent extends React.Component {
  render() {
    return (
      <div dangerouslySetInnerHTML={{"__html": aboutUserText}} />
    );
```

(continues on next page)

(continued from previous page)

```
  }
}

ReactDOM.render(<AboutUserComponent />, document.querySelector("#app"))
```

3.2.4 Injectable Attributes

If the user controls the `href` attribute of a dynamically generated `a` tag then there is nothing to prevent the attacker from injecting a `javascript:` url.

```
<a href={userinput}>Link</a>
<button form="name" formaction={userinput}>
```

3.2.5 Eval-based Injection

If the attacker can provide an input that is then dynamically evaluated then there is nothing to stop them from injecting harmful code.

```
function antiPattern() {
  eval(this.state.attacker_supplied);
}

// Or even crazier
fn = new Function("..." + attacker_supplied + "...");
fn()
```

3.3 Resources & Interesting Reads

https://github.com/facebook/react/issues/3473#issuecomment-90594748

https://medium.com/dailyjs/exploiting-script-injection-flaws-in-reactjs-883fb1fe36c1

https://medium.com/javascript-security/avoiding-xss-via-markdown-in-react-91665479900

https://github.com/facebook/react/issues/14160

CHAPTER 4

Indices and tables

- search