XSS ATTACKS

MARU LUCENA T.

Hi, I'm Maru!

When I started working as a dev I had very little knowledge about cyber security. Thankfully, I had an awesome team of senior devs that guided me, especially the first time I had to fix an XSS vulnerability.

This zine is based on my personal story and it's a compilation of all the explanations that have helped me understand XSS and HTML escaping (with some fun drawings!).

It's not a deep dive into the topic but rather an overview of the main concepts necessary to prevent XSS attacks, I would say it's a good introduction!

I don't take any credit for the technical explanations. I used OWASP (Open Web Application Security Project) as my main source.

I hope reading it brings awareness to one of the most common web security vulnerabilities, making this zine certainly did it for me :)

Special thanks to Jesse and Danilo for always teaching me how to be a better dev.

Also, special thanks to Julia Evans because her amazing zines inspired me to make my own.

The first time I heard about the term XSS was something like this:

HEY MARU THERE'S AN XSS VULNERABILITY IN THIS FORM, CAN YOU FIX IT?
YES, SURE!
XSS WHAT?!?!

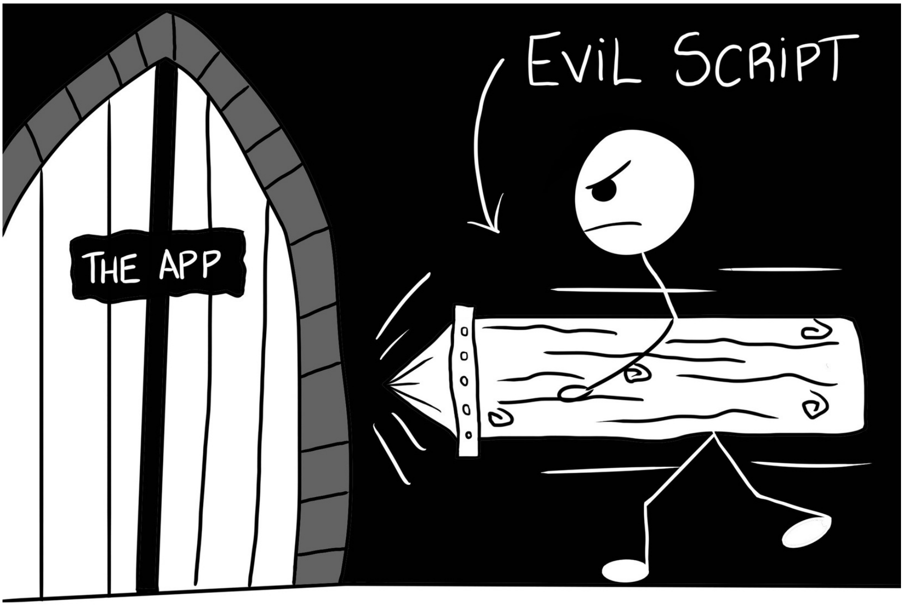
Therefore, I looked it up:

OWASP.org

"Cross-Site Scripting (XSS) attacks are a type of injection, in which malicious scripts are injected into otherwise benign and trusted websites.

XSS attacks occur when an attacker uses a web application to send malicious code, generally in the form of a browser side script, to a different end user's browser who has no way to know that the script should not be trusted, and will execute it."

Reading the word "attack" made me panic a bit. Before that I hadn't dealt with a security vulnerability, thankfully nothing terrible had happened and it was reported on time but I imagined something like this:



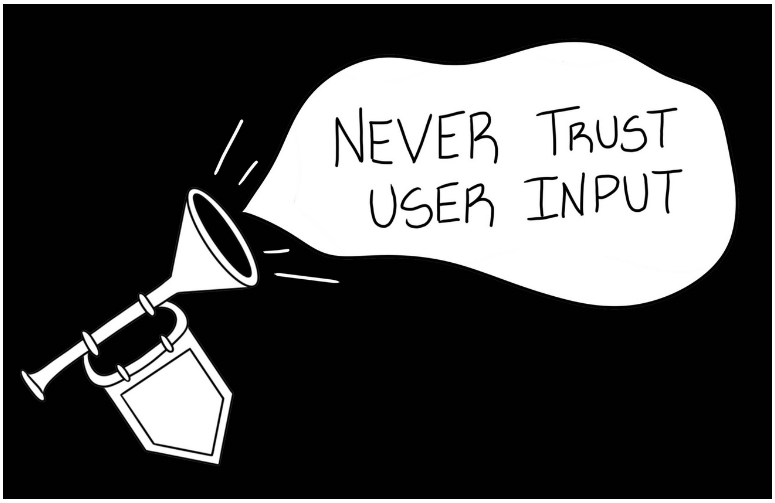
The idea of someone with bad intentions being able to inject malicious code in our app made me nervous but also inspired me to learn how to prevent it.

As it turns out, we had several XSS vulnerabilities in our legacy codebase.

I learned that the most common one was happening because we were displaying unescaped user input where we shouldn't have.

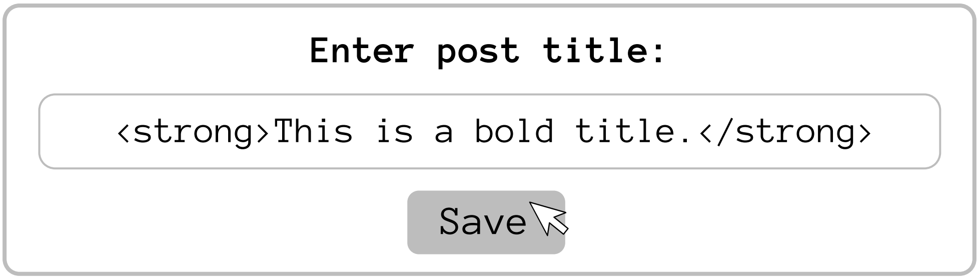
Understanding the difference between escaped and unescaped user input was very straightforward but I kept mixing up the terms because they sound similar, so one way I like to differentiate them is by remembering that "unescaped" could potentially mean "unsafe".

LESSON #1:



For example, let's say you have an app that allows users to share blog posts and there's a form where they can enter the title of the post.

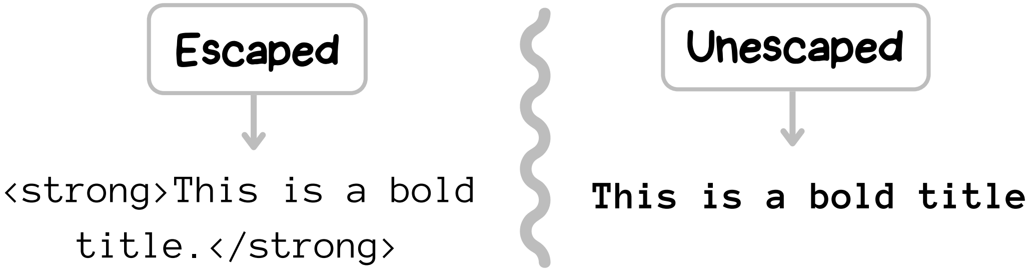
What if someone saves this as the post title?



Enter post title:

```
<strong>This is a bold title.</strong>
```

Depending on how it's being displayed, the output could be either of these:



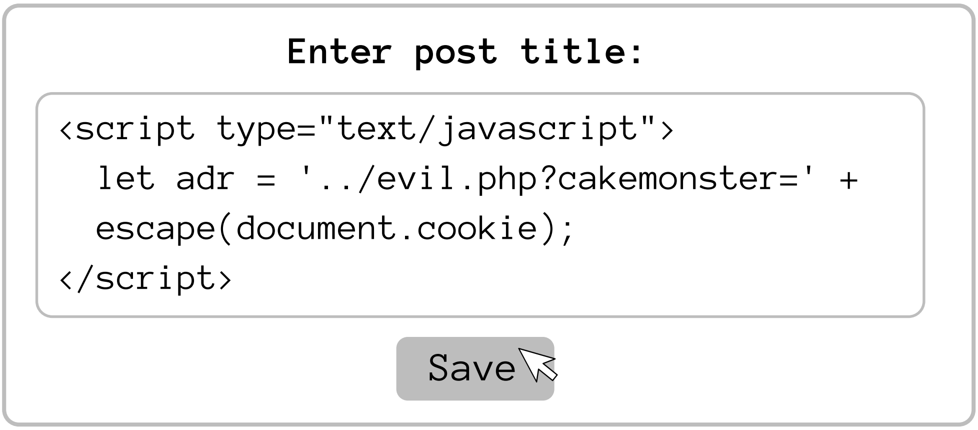
```
<strong>This is a bold
title.</strong>
```

`This is a bold title`

As you can see in the unescaped version, the code is being executed. Instead of displaying the text "as it is" the browser is interpreting it as HTML.

That previous example was just styling but it can be a not so friendly script that would run for all users who visit the page where that blog title is!

Like a cookie grabber script:



```
<script type="text/javascript">
  let adr = '../evil.php?cakemonster=' +
  escape(document.cookie);
</script>
```

Example from OWASP.

If the app doesn't escape the input and validates it, an attacker can steal an authenticated user's cookie by passing its content to the evil.php script in the "cakemonster" variable.

The attacker then checks the results of their script and uses the cookie.

Think about every place in your app where you're ingesting user input. It could be text, links, attributes.

If not handled properly, those are all opportunities for users to inject malicious code. Making sure the data is properly encoded is very important to prevent that from happening.

"XSS attacks may occur anywhere that possibly malicious users are allowed to post unregulated material to a trusted website for the consumption of other valid users."

HMM... SEEMS SUSPICIOUS
THE APP
FRIENDLY USER INPUT

HTML escaping

We saw that displaying escaped user input prevents the browser from interpreting the text as HTML and executing the code.

HTML escaping, also known as character encoding, is the process of replacing special characters with their corresponding HTML entities to make sure they're displayed correctly.

For example, if the less than symbol (<) and the greater than symbol (>) are not properly encoded, they can be interpreted as part of an HTML tag, which can cause unexpected behaviors.

HTML escaping involves replacing these special characters with their corresponding HTML entities, such as &lt; for < and &gt; for >.

Going back to our example of bold text, the escaped output would look like this to users:

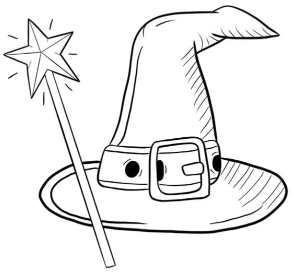
```
<strong>This is a bold title.</strong>
```

But, if you go to "View source code" in the browser and find that same element, you'd see the escaped HTML like this:

```
&lt;strong&gt;This is bold text.&lt;/strong&gt;
```

Note: this can't be seen using the inspect tool because the inspector shows a view of the DOM and at that point the HTML has already been parsed and the entities converted to characters in text nodes.

It kind of feels like magic, right? ;)



"BY THE MYSTIC ECHOES OF CODE AND WEB, I SUMMON THE HTML ESCAPE, A SHIELD TO WEAVE < AND >, NOW TRANSFORMED IN THIS ARCANE DANCE, PROTECT MY TEXT FROM MISCHIEF'S CHANCE. RENDERED SAFE IN THE WEB'S GRAND MAZE, LET THIS SPELL GUARD MY CONTENT'S DAYS!"

Terminology: *escaping vs encoding*

I've mentioned a couple of times the words "escaping" and "encoding" and when it comes to XSS these are largely used interchangeably as both help render unsafe *external* inputs safe in an *executable* context.

But they are technically two *different* things:

Encoding involves translating special characters into some *different* but *equivalent* form that is no longer *dangerous* in the target interpreter.

$$< \longrightarrow \texttt{\&lt;}$$

Escaping is a subset of *encoding*, it involves adding a special character before the character/string to avoid it being misinterpreted.

```
'that\'s it'
```

$\hookrightarrow$ escape character

Source

Here's a summary of the terms recently mentioned:

HTML escaping/encoding

Converting special characters that have special meaning in HTML into HTML entities

HTML entities

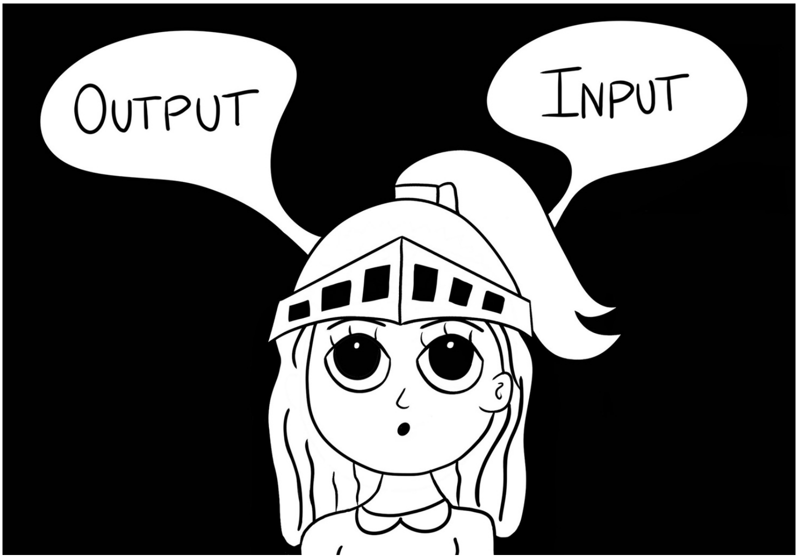
A way of representing special characters or symbols that cannot be directly used in HTML

&lt; is one of the many HTML entities

When should you escape?

Should you escape user input before storing it in the database or save it as it is and escape it while retrieving? This was a question I saw in StackOverflow that I thought would be interesting to mention here.

My friend Jesse was explaining this difference to me, he mentioned that you can either escape on output or input and that generally you always want to escape on output.



Output is any data that leaves your application bound for another client or application. The receiving client or application expects the data to be of a specific format (HTML, SQL, etc.)

Therefore, the only code that knows what characters are dangerous is the code that's outputting in a given context.

So the better approach is to store the data as it's intended in the database, and then have the template system HTML-escape when outputting HTML.

In the previous sections I focused on output encoding for "HTML Contexts" but there are many different output encoding methods because browsers parse HTML, JavaScript, URLs, and CSS differently.

Therefore it's important to use the right encoding method for each context, otherwise the wrong method may introduce weaknesses or harm the functionality of your app.

Other contexts may include:

- HTML Attribute Contexts
- JavaScript Contexts
- CSS Contexts

Check out this OWASP cheatsheet to learn about all the other contexts.

All contexts

How to safely display content when users need to author HTML

Let's say you want to allow users to change the styling or structure of content of a blog post inside a WYSIWYG editor. Output encoding here will prevent XSS, but it will break the intended functionality of the editor as the styling will not be rendered. In these cases, HTML Sanitization should be used.

HTML sanitization is the process of examining an HTML document and producing a new one that preserves only whatever tags are designated "safe" and desired.

OWASP recommends DOMPurify for HTML Sanitization.

There are some further things to consider:

- If you sanitize content and then modify it afterwards, you can easily void your security efforts.

- If you sanitize content and then send it to a library for use, check that it doesn't mutate that string somehow. Otherwise, again, your security efforts are void.

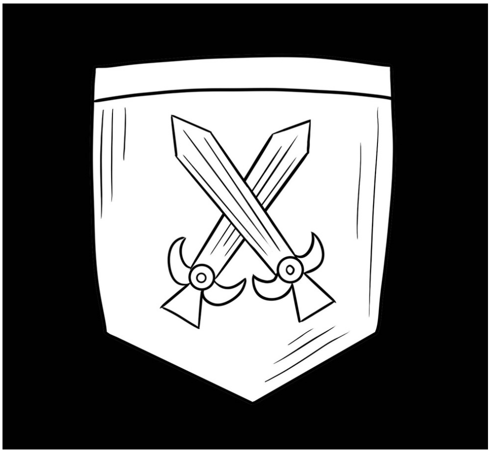
Using frameworks that have built in measures to prevent XSS

Apps built with modern frameworks that default to safely performing output encoding are apps with fewer XSS bugs.

OWASP says:
"These frameworks steer developers towards good security practices and help mitigate XSS by using templating, auto-escaping, and more."

Here are some examples:

- ReactJS
- AngularJS
- Handlebars
- LiquidJS
- Rails



It's important you understand how your framework prevents XSS and where it has gaps.

In general you don't want to bypass the auto-escaping but there will be times where you need to do something outside the protection provided by your framework.

This is where Output Encoding and HTML Sanitization are critical.

Special mention

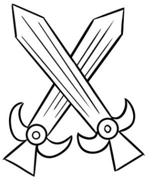
I wanted to make one special mention to input validation since it can significantly contribute to prevent XSS.

Input validation is the testing of any input (or data) provided by a user or application against expected criteria. It prevents malicious or poorly qualified data from entering an information system to prevent attacks and mistakes.

I like to think of adding validation rules as affirmations for my inputs.

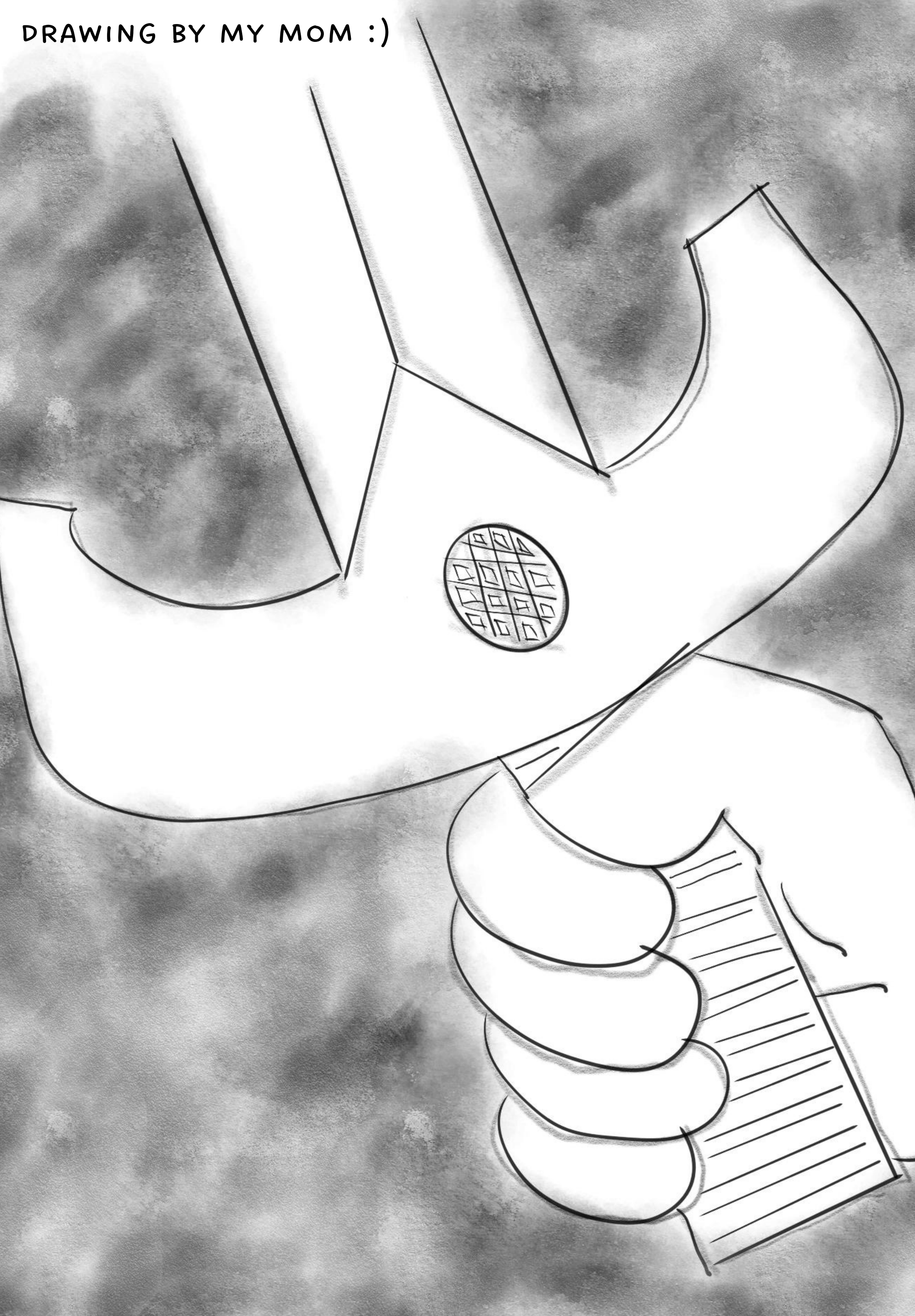
"For XSS attacks to be successful, an attacker needs to insert and execute malicious content in a webpage. Ensuring that all variables go through validation and are then escaped or sanitized is known as perfect injection resistance.

Any variable that does not go through this process is a potential weakness. "



- OWASP

DRAWING BY MY MOM :)



Thanks for reading!

marucodes.com