Cross-site scripting contexts

When testing for reflected and stored XSS, a key task is to identify the XSS context:

- The location within the response where attacker-controllable data appears.
- Any input validation or other processing that is being performed on that data by the application.

Based on these details, you can then select one or more candidate XSS payloads, and test whether they are effective.

> **Note**
>
> We have built a comprehensive XSS cheat sheet to help testing web applications and filters. You can filter by events and tags and see which vectors require user interaction. The cheat sheet also contains AngularJS sandbox escapes and many other sections to help with XSS research.

XSS between HTML tags

When the XSS context is text between HTML tags, you need to introduce some new HTML tags designed to trigger execution of JavaScript.

Some useful ways of executing JavaScript are:

```
<script>alert(document.domain)</script>
<img src=1 onerror=alert(1)>
```

LAB APPRENTICE **Reflected XSS into HTML context with nothing encoded →**

LAB APPRENTICE **Stored XSS into HTML context with nothing encoded →**

Reflected XSS into HTML context with most tags and attributes blocked →

LAB

PRACTITIONER

Reflected XSS into HTML context with all tags blocked except custom ones →

LAB

EXPERT

Reflected XSS with event handlers and `href` attributes blocked →

LAB

PRACTITIONER

Reflected XSS with some SVG markup allowed →

XSS in HTML tag attributes

When the XSS context is into an HTML tag attribute value, you might sometimes be able to terminate the attribute value, close the tag, and introduce a new one. For example:

```
"><script>alert(document.domain)</script>
```

More commonly in this situation, angle brackets are blocked or encoded, so your input cannot break out of the tag in which it appears. Provided you can terminate the attribute value, you can normally introduce a new attribute that creates a scriptable context, such as an event handler. For example:

```
" autofocus onfocus=alert(document.domain) x="
```

The above payload creates an `onfocus` event that will execute JavaScript when the element receives the focus, and also adds the `autofocus` attribute to try to trigger the `onfocus` event automatically without any user interaction. Finally, it adds `x="` to gracefully repair the following markup.

LAB

APPRENTICE

Reflected XSS into attribute with angle brackets HTML-encoded →

Sometimes the XSS context is into a type of HTML tag attribute that itself can create a scriptable context. Here, you can execute JavaScript without needing to terminate the attribute value. For example, if the XSS context is into the `href` attribute of an anchor tag, you can use the `javascript` pseudo-protocol to execute script. For example:

```
<a href="javascript:alert(document.domain)">
```

LAB

APPRENTICE

You might encounter websites that encode angle brackets but still allow you to inject attributes. Sometimes, these injections are possible even within tags that don't usually fire events automatically, such as a canonical tag. You can exploit this behavior using access keys and user interaction on Chrome. Access keys allow you to provide keyboard shortcuts that reference a specific element. The `accesskey` attribute allows you to define a letter that, when pressed in combination with other keys (these vary across different platforms), will cause events to fire. In the next lab you can experiment with access keys and exploit a canonical tag. You can exploit XSS in hidden input fields using a technique invented by PortSwigger Research.

LAB PRACTITIONER

Reflected XSS in canonical link tag →

XSS into JavaScript

When the XSS context is some existing JavaScript within the response, a wide variety of situations can arise, with different techniques necessary to perform a successful exploit.

Terminating the existing script

In the simplest case, it is possible to simply close the script tag that is enclosing the existing JavaScript, and introduce some new HTML tags that will trigger execution of JavaScript. For example, if the XSS context is as follows:

```
<script>
...
var input = 'controllable data here';
...
</script>
```

then you can use the following payload to break out of the existing JavaScript and execute your own:

```
</script><img src=1 onerror=alert(document.domain
```

The reason this works is that the browser first performs HTML parsing to identify the page elements including blocks of script, and only later performs JavaScript parsing to understand and execute the embedded scripts. The above payload leaves the original script broken, with an unterminated string literal. But that doesn't prevent the subsequent script being parsed and executed in the normal way.

LAB PRACTITIONER

Reflected XSS into a JavaScript string with single quote and backslash escaped →

Breaking out of a JavaScript string

In cases where the XSS context is inside a quoted string literal, it is often possible to break out of the string and execute JavaScript directly. It is essential to repair the script following the XSS context, because any syntax errors there will prevent the whole script from executing.

Some useful ways of breaking out of a string literal are:

```
'-alert(document.domain)-'
';alert(document.domain)//
```

LAB APPRENTICE **Reflected XSS into a JavaScript string with angle brackets HTML encoded →**

Some applications attempt to prevent input from breaking out of the JavaScript string by escaping any single quote characters with a backslash. A backslash before a character tells the JavaScript parser that the character should be interpreted literally, and not as a special character such as a string terminator. In this situation, applications often make the mistake of failing to escape the backslash character itself. This means that an attacker can use their own backslash character to neutralize the backslash that is added by the application.

For example, suppose that the input:

```
';alert(document.domain)//
```

gets converted to:

```
\';alert(document.domain)//
```

You can now use the alternative payload:

```
\';alert(document.domain)//
```

which gets converted to:

```
\\';alert(document.domain)//
```

Here, the first backslash means that the second backslash is interpreted literally, and not as a special character. This means that the quote is now interpreted as a string terminator, and so the attack succeeds.

LAB PRACTITIONER **Reflected XSS into a JavaScript string with angle brackets and double quotes HTML-encoded and single quotes escaped →**

Some websites make XSS more difficult by restricting which characters you are allowed to use. This can be on the website level or by deploying a WAF that prevents your requests from ever

other ways of calling functions which bypass these security measures. One way of doing this is to use the `throw` statement with an exception handler. This enables you to pass arguments to a function without using parentheses. The following code assigns the `alert()` function to the global exception handler and the `throw` statement passes the `1` to the exception handler (in this case `alert`). The end result is that the `alert()` function is called with `1` as an argument.

```
onerror=alert;throw 1
```

There are multiple ways of using this technique to call functions without parentheses.

The next lab demonstrates a website that filters certain characters. You'll have to use similar techniques to those described above in order to solve it.

LAB EXPERT

Reflected XSS in a JavaScript URL with some characters blocked →

Making use of HTML-encoding

When the XSS context is some existing JavaScript within a quoted tag attribute, such as an event handler, it is possible to make use of HTML-encoding to work around some input filters.

When the browser has parsed out the HTML tags and attributes within a response, it will perform HTML-decoding of tag attribute values before they are processed any further. If the server-side application blocks or sanitizes certain characters that are needed for a successful XSS exploit, you can often bypass the input validation by HTML-encoding those characters.

For example, if the XSS context is as follows:

```
<a href="#" onclick="... var input='controllable
```

and the application blocks or escapes single quote characters, you can use the following payload to break out of the JavaScript string and execute your own script:

```
&apos;-alert(document.domain)-&apos;
```

The `&apos;` sequence is an HTML entity representing an apostrophe or single quote. Because the browser HTML-decodes the value of the `onclick` attribute before the JavaScript is interpreted, the entities are decoded as quotes, which become string delimiters, and so the attack succeeds.

LAB PRACTITIONER

Stored XSS into `onclick` event with angle brackets and double quotes HTML-encoded and

XSS in JavaScript template literals

JavaScript template literals are string literals that allow embedded JavaScript expressions. The embedded expressions are evaluated and are normally concatenated into the surrounding text. Template literals are encapsulated in backticks instead of normal quotation marks, and embedded expressions are identified using the `${...}` syntax.

For example, the following script will print a welcome message that includes the user's display name:

```
document.getElementById('message').innerText = `W
```

When the XSS context is into a JavaScript template literal, there is no need to terminate the literal. Instead, you simply need to use the `${...}` syntax to embed a JavaScript expression that will be executed when the literal is processed. For example, if the XSS context is as follows:

```
<script>
...
var input = `controllable data here`;
...
</script>
```

then you can use the following payload to execute JavaScript without terminating the template literal:

```
${alert(document.domain)}
```

LAB PRACTITIONER

Reflected XSS into a template literal with angle brackets, single, double quotes, backslash and backticks Unicode-escaped →

XSS via client-side template injection

Some websites use a client-side template framework, such as AngularJS, to dynamically render web pages. If they embed user input into these templates in an unsafe manner, an attacker may be able to inject their own malicious template expressions that launch an XSS attack.

Read more

- Client-side template injection