Obfuscating attacks using encodings

In this section, we'll show you how you can take advantage of the standard decoding performed by websites to evade input filters and inject harmful payloads for a variety of attacks, such as XSS and SQL injection.

Context-specific decoding

Both clients and servers use a variety of different encodings to pass data between systems. When they want to actually use the data, this often means they have to decode it first. The exact sequence of decoding steps that are performed depends on the context in which the data appears. For example, a query parameter is typically URL decoded server-side, while the text content of an HTML element may be HTML decoded client-side.

When constructing an attack, you should think about where exactly your payload is being injected. If you can infer how your input is being decoded based on this context, you can potentially identify alternative ways to represent the same payload.

Decoding discrepancies

Injection attacks often involve injecting payloads that use recognizable patterns, such as HTML tags, JavaScript functions, or SQL statements. As the inputs for these payloads are almost never expected to contain user-supplied code or markup, websites often implement defences that block requests containing these suspicious patterns.

However, these kinds of input filters also need to decode the input in order to check whether it's safe or not. From a security perspective, it's vital that the decoding performed when checking the input is the same as the decoding performed by the back-end server or browser when it eventually uses the data. Any discrepancy can enable an attacker to sneak harmful payloads past the filter by applying different encodings that will automatically be removed later.

Obfuscation via URL encoding

In URLs, a series of reserved characters carry special meaning. For example, an ampersand (`&`) is used as a delimiter to separate parameters in the query string. The problem is, URL-based inputs may contain these characters for a different reason. Consider a parameter containing a user's search query. What happens if the user searches for something like "Fish & Chips"?

Browsers automatically URL encode any characters that may cause ambiguity for parsers. This usually means substituting them with a `%` character and their 2-digit hex code as follows:

```
[...]/?search=Fish+%26+Chips
```

This ensures that the ampersand will not be mistaken for a delimiter.

Note

Although the space character can be encoded as `%20` , it is often represented by a plus (`+`) instead, as in the example above.

Any URL-based input is automatically URL decoded server-side before it is assigned to the relevant variables. This means that, as far as most servers are concerned, sequences like `%22` , `%3D` , and `%3E` in a query parameter are synonymous with `"` , `<` , and `>` characters respectively. In other words, you can inject URL-encoded data via the URL and it will usually still be interpreted correctly by the back-end application.

Occasionally, you may find that WAFs and suchlike fail to properly URL decode your input when checking it. In this case, you may be able to smuggle payloads to the back-end application simply by encoding any characters or words that are blacklisted. For example, in a SQL injection attack, you might encode the keywords, so `SELECT` becomes `%53%45%4C%45%43%54` and so on.

Obfuscation via double URL encoding

For one reason or another, some servers perform two rounds of URL decoding on any URLs they receive. This isn't necessarily an issue in its own right, provided that any security mechanisms also double-decode the input when checking it. Otherwise, this discrepancy enables an attacker to smuggle malicious input to the back-end by simply

Track your progress

Learning materials: View all

Vulnerability labs: View all

Level progress:

5 of 52	2 of 147	0 of 35
Apprentice	Practitioner	Expert

Your level:

Ne

NEWBIE

Solve 47 more labs to become an apprentice.

See where you rank on our Hall of Fame »

Obfuscating attacks using encodings

☐ **Mark as complete**

In this topic

All topics

encoding it twice.

Let's say you're trying to inject a standard XSS PoC, such as `<img src=x onerror=alert(1)>`, via a query parameter. In this case, the URL might look something like this:

```
[...]/?search=%3Cimg%20src%3Dx%20onerror%3Dalert(1)%3E
```

When checking the request, if a WAF performs the standard URL decoding, it will easily identify this well-known payload. The request is blocked from ever reaching the back-end. But what if you double-encode the injection? In practice, this means that the `%` characters themselves are then replaced with `%25`:

```
[...]/?search=%253Cimg%2520src%253Dx%2520onerror%253Dalert(1)%253E
```

As the WAF only decodes this once, it may not be able to identify that the request is dangerous. If the back-end server subsequently double-decodes this input, the payload will be successfully injected.

Obfuscation via HTML encoding

In HTML documents, certain characters need to be escaped or encoded to prevent the browser from incorrectly interpreting them as part of the markup. This is achieved by substituting the offending characters with a reference, prefixed with an ampersand and terminated with a semicolon. In many cases, a name can be used for the reference. For example, the sequence `&colon;` represents a colon character.

Alternatively, the reference may be provided using the character's decimal or hex code point, in this case, `&#58;` and `&#x3a;` respectively.

In specific locations within the HTML, such as the text content of an element or the value of an attribute, browsers will automatically decode these references when they parse the document. When injecting inside such a location, you can occasionally take advantage of this to obfuscate payloads for client-side attacks, hiding them from any server-side defences that are in place.

If you look closely at the XSS payload from our earlier example, notice that the payload is being injected inside an HTML attribute, namely the `onerror` event handler. If the server-side checks are looking for the `alert()` payload explicitly, they might not spot this if you HTML encode one or more of the characters:

```
<img src=x onerror="&#x61;lert(1)">
```

When the browser renders the page, it will decode and execute the injected payload.

Leading zeros

Interestingly, when using decimal or hex-style HTML encoding, you can optionally include an arbitrary number of leading zeros in the code points. Some WAFs and other input filters fail to adequately account for this.

If your payload still gets blocked after HTML encoding it, you may find that you can evade the filter just by prefixing the code points with a few zeros:

```
<a href="javascript&#00000000000058;alert(1)">Click me</a>
```

Obfuscation via XML encoding

XML is closely related to HTML and also supports character encoding using the same numeric escape sequences. This enables you to include special characters in the text content of elements without breaking the syntax, which can come in handy when testing for XSS via XML-based input, for example.

Even if you don't need to encode any special characters to avoid syntax errors, you can potentially take advantage of this behavior to obfuscate payloads in the same way as you do with HTML encoding. The difference is that your payload is decoded by the server itself, rather than client-side by a browser. This is useful for bypassing WAFs and other filters, which may block your requests if they detect certain keywords associated with SQL injection attacks.

```
<stockCheck>
    <productId>
        123
    </productId>
    <storeId>
        999 &#x53;ELECT * FROM information_schema.tables
    </storeId>
</stockCheck>
```

LAB PRACTITIONER **SQL injection with filter bypass via XML encoding** » Not solved

Obfuscation via unicode escaping

Unicode escape sequences consist of the prefix `\u` followed by the four-digit hex code for the character. For example, `\u003a` represents a colon. ES6 also supports a new form of unicode escape using curly braces: `\u{3a}`.

When parsing strings, most programming languages decode these unicode escapes. This includes the JavaScript engine used by browsers. When injecting into a string context, you can obfuscate client-side payloads using unicode, just like we did with HTML escapes in the example above.

For example, let's say you're trying to exploit DOM XSS where your input is passed to the `eval()` sink as a string. If your initial attempts are blocked, try escaping one of the characters as follows:

```
eval("\u0061lert(1)")
```

As this will remain encoded server-side, it may go undetected until the browser decodes it again.

Note

Inside a string, you can escape any characters like this. However, outside of a string, escaping some characters will result in a syntax error. This includes opening and closing parentheses, for example.

It's also worth noting that the ES6-style unicode escapes also allow optional leading zeros, so some WAFs may be easily fooled using the same technique we used for HTML encodings. For example:

```
<a href="javascript\u{0000000003a}alert(1)">Click me</a>
```

Obfuscation via hex escaping

Another option when injecting into a string context is to use hex escapes, which represent characters using their hexadecimal code point, prefixed with `\x`. For example, the lowercase letter `a` is represented by `\x61`.

Just like unicode escapes, these will be decoded client-side as long as the input is evaluated as a string:

```
eval("\x61lert")
```

Note that you can sometimes also obfuscate SQL statements in a similar manner using the prefix `0x`. For example, `0x53454c454354` may be decoded to form the `SELECT` keyword.

Obfuscation via octal escaping

Octal escaping works in pretty much the same way as hex escaping, except that the character references use a base-8 numbering system rather than base-16. These are prefixed with a standalone backslash, meaning that the lowercase letter `a` is represented by `\141`.

```
eval("\141lert(1)")
```

Obfuscation via multiple encodings

It is important to note that you can combine encodings to hide your payloads behind multiple layers of obfuscation. Look at the `javascript:` URL in the following example:

```
<a href="javascript:&bsol;u0061lert(1)">Click me</a>
```

Browsers will first HTML decode `&bsol;`, resulting in a backslash. This has the effect of turning the otherwise arbitrary `u0061` characters into the unicode escape `\u0061`:

```
<a href="javascript:\u0061lert(1)">Click me</a>
```

This is then decoded further to form a functioning XSS payload:

```
<a href="javascript:alert(1)">Click me</a>
```

Clearly, to successfully inject a payload in this way, you need a solid understanding of which decoding is performed on your input and in what order.

Obfuscation via the SQL CHAR() function

Although not strictly a form of encoding, in some cases, you may be able to obfuscate your SQL injection attacks using the `CHAR()` function. This accepts a single decimal or hex code point and returns the matching character. Hex codes must be prefixed with `0x`. For example, both `CHAR(83)` and `CHAR(0x53)` return the capital letter `S`.

By concatenating the returned values, you can use this approach to obfuscate blocked keywords. For example, even if `SELECT` is blacklisted, the following injection initially appears harmless:

```
CHAR(83)+CHAR(69)+CHAR(76)+CHAR(69)+CHAR(67)+CHAR(84)
```

However, when this is processed as SQL by the application, it will dynamically construct the `SELECT` keyword and execute the injected query.