**INTRODUCTION**

The aim of this study is improve knowledge about Obfuscation Techniques that are being used as escaping method on attacks.

**DETAILS**

**What is Obfuscating attacks using encodings?**

In this research we will learn how we can take advantage of the standard decoding permormed by websites to escape from input filters and inject harmful payloads for variety of attacks f.e. XSS and SQLi

Context-specific decoding

Both clients and servers use a variety of different encoding to pass data between systems. If they want to use data, they must decode it. Sequence of decoding steps that are performed depends on the context in which the data appears. For instance, a query parameter is typically URL decoded server-side, while the text content of an HTML element may be HTML decoded client-side.

When preparing our attack, we should think about where exactly our payload is being injected. If we understand how our input is decoded based on the context, we can potentially identify different ways to represent the same payload.

Decoding discrepancies

Injection atacks often involve injecting payload that uses recognizable patterns, such as HTML tags (<>), JavaScript functions(onclick(),eval()) or SQL statements(asd’ OR 1=1). The inputs for these payloads are never expected to contain user-supplied code or markup (meaning no input field requires ‘ OR 1=1 or something), websites often implement defences that block requests containing dangerous patterns.

However, these kinds of input filtering also need to decode the input to check whether it’s safe or not. From a security perspective, it’s so important that the decoding performed when checking the input is the same as the decoding performed by the back-end server or browser when it actually uses data. Any discrepancy makes attacker to sneak harmful payloads that can bypass the filters by applying different encoding that will automatically removed later.

**1-) Obfuscation Via URL Encoding**

In URLs, a series of reserved characters have special meanings. For example, an ampersand (&) is used as a delimiter to seperate 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 will happen when user searchs for “Ali & Veli”?

Browser automatically URL encode any character that may cause ambiguity for parsers. This generally means substituting them with a % character and their 2-digit hex code like below:

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

This ensures that 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 sever are aware, sequences like %22, %3D, and %3E in a query parameter are synonymous with **“**, **<**, **>** characters respectively. In other words, we can inject URL-encoded data via URL and it will usually still be interpreted correctly by the back-end application.

Sometimes we may find that some WAFs and suchlike fail to properly URL decode our input when checking it. In this case we are simply encode characters that are blacklisted in our payload. For example, in a SQL injection attack, we might encode the keywords, so SELECT becomes %53%45%4C%45%43%54 and so on.

**2-) 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 encoding it twice.

Let's say you're trying to inject a standard XSS, 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.

**3-) Obfuscation Via HTML Encoding**

In HTML documents, some characters need to be escaped or encoded to prevent browser from incorrectly interpreting them as part of the markup. This is achieved by substituting the dangerous 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 spesific 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, we can use this obfuscation techniques to bypass any client side protection, and also hiding them from any server-side defences.

In the below payload we can see that payload is being injected inside an HTML attribute, namely the onerror event handler. If server side has a blacklist for the alert() keyword, it might not spot this if we HTML encode one or more of the characters like below:

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

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

**\*\*\*Leading Zeros**

Interesting concept is that when we are using decimal or hex-style HTML encoding, we can optionally include an arbitrary number of leading zeros in the code points. Some WAFs and other input filters fail to validate this.

If our payload sill gets blocked afer HTML encoding it, we may find that we can evade the filter just by prefixing the code points with a few zeros:

<a href="javascript&#**000000000000**58;alert(1)">Click me</a>

* &#58; means “:” character. Zeros behind it just for bypassing a filter.

**4-) Obfuscation Via XML Encoding**

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

Even if we don’t need to encode any special characters to avoid syntax errors, we can potentially take advantage of this behaviour to obfuscate payloads in the same way the we do in HTML encoding. Difference is that, this time our playoad decoded by server itself, rather than the client-side browser. This will come handy when bypassing WAFs and other filters. which may block our request if they detect some dangerous keywords that are associated with SQLi XSS and XXE.

<stockCheck>

<productId>

123

</productId>

<storeId>

999 &#x53;ELECT \* FROM information\_schema.tables

</storeId>

</stockCheck>

**5-) 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 “:”. JavaScript ES6 also supports a new form of Unicode escaping using curly braces: \u{3a}

When parsing strings, most of the programming languages are decoding Unicode escapes. This also includes JavaScript Engines used by browsers. When injecting into a string, we can obfuscate client-side payloads using Unicode, like we did in HTML escapes.

Let’s say we are trying to exploit DOM XSS where our input is passed to the *eval()* function as a string. If our 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, we can escape any character we want. But outside of the string, escaping some characters will give us sytnax error. This includes opening and closing parantheses for example.

Note that 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>

**6-) Obfuscation Via Hex Escaping**

Another option when injecting into a string context is to use hex escapes, which represents characters using their hexadecimal code point, prefixed with \x. For example, the lowercase letter *a* is represented by *\x61*. (a hexadecimal value = 61)

Like Unicode escapes, these will be decoded client-side as long as the input is evaluated as a string:

eval("\x61lert")

Note that we can sometimes also obfuscate SQL statements in a similiar manner using the prefix *0x*. For example 0x53454c454354 may be decoded to form the SELECT keyword. (Select’s hex value = 53454c454354)

**7-) Obfuscation Via Octal Escaping**

Octal escaping Works in prettuy 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 just backslash(\). For example lowercase “a” is represented as *\141*.

eval("\141lert(1)")

**7-) Obfuscation Via Multiple Encodings**

It is important to note that we can combine encodings to hide our payloads behind multiple layers of obfuscation. Look at the javascript: URL below:

<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>

Then this is decoded futher to form a functioning XSS payload:

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

\*\*\* To successfully inject a payload in this way, we need a solid understanding of which decoding is performed on our input and in what order.

**7-) Obfuscation Via SQL CHAR() Function**

Although not an exactly form of encoding, in some cases, w emay be able to obfuscate our 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 frefixed with *0x*. For example, both CHAR(83) and CHAR(0x53) return the capital letter “*S”*.

By concatenating the returned values, we can use this approach to obfuscate blocked keywords. For example, even if *SELECT* is blacklisted, the following injection initially seems harmless by application:

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.

**REFERENCES**

[**https://portswigger.net/web-security/reference/obfuscating-attacks-using-encodings#obfuscation-via-xml-encoding**](https://portswigger.net/web-security/reference/obfuscating-attacks-using-encodings#obfuscation-via-xml-encoding)