Pentest-Report DOMPurify 02.2015

@filedescriptor

Index

Introduction

“DOMPurify is a DOM-only, super-fast, uber-tolerant XSS sanitizer for HTML, MathML and SVG. It's written in JavaScript and works in all modern browsers (Safari, Opera (15+), Internet Explorer (9+), Firefox and Chrome - as well as almost anything else using Blink or WebKit). It doesn't break on IE6 or other legacy browsers. It simply does nothing there.

DOMPurify sanitizes HTML and prevents XSS attacks. You can feed DOMPurify with string full of dirty HTML and it will return a string with clean HTML. DOMPurify will strip out everything that contains dangerous HTML and thereby prevent XSS attacks and other nastiness. It's also damn bloody fast. We use the technologies the browser provides and turn them into an XSS filter. The faster your browser, the faster DOMPurify will be.”

From https://github.com/cure53/DOMPurify

This penetration test was carried out over the period of three days. The test yielded an overall five issues, including two high-range vulnerabilities and three minor weaknesses. The testing methodology involves source code analysis as well as browser quirks investigation (e.g. Mutation XSS). The test, unsurprisingly, resulted in an low amount of exploitable vulnerabilities.

Scope

- **DOMPurify Implementation**
  - **https://github.com/cure53/DOMPurify/blob/master/purify.js**

Identified Vulnerabilities

The following sections list both vulnerabilities and implementation issues spotted during the testing period. Note that findings are listed in a chronological order rather than by their degree of severity and impact, which is simply given in brackets following the title heading for each vulnerability. Each vulnerability is additionally given a unique identifier (e.g. *DOM-01-001*) for the purpose of facilitating any future follow-up correspondence.

DOM-01-002 Double-clobbering enables sanitization bypass *(High)*

Some DOM method calls lack clobbering protection. Under certain situations, attackers are able to bypass the sanitization with multiple calls on *DOMPurify.sanitize*.

DOMPurify provides fallback for legacy browsers by simply returning the original input. Specifically, a user agent is regarded as legacy if it does not support *DOMImplementation.createHTMLDocument* method. This check however, is vulnerable to DOM clobbering attack.

```
if (typeof document.implementation.createHTMLDocument === 'undefined') {
    if (typeof window.toStaticHTML === 'function' && typeof dirty === 'string') {
        return window.toStaticHTML(dirty);
    }
    return dirty;
}
```

In Google Chrome and Mozilla Firefox, properties on the document object can be overridden. In other words, the attacker can clobber *document.implementation* and force DOMPurify to degrade to fallback mode, therefore bypassing the entire sanitization. However, the attack is only possible when the default DOM clobbering protection on DOMPurify is disabled (i.e. SANITIZE_DOM flag set to false). Still, given that DOMPurify provides the option, it is not a desirable behavior in terms of security goals cosidering the corresponding impact.

Proof-of-Concept

```
elem.innerHTML += DOMPurify.sanitize('<img name=implementation>', {SANITIZE_DOM: false});
elem.innerHTML += DOMPurify.sanitize('<img src=x: onerror=alert(1)>');
```

The above PoC utilizes the double-clobbering technique. In the first line, the *img* element after being sanitized is inserted into the *elem*, clobbering the *document.implementation* method. Thereafter, any call on DOMPurify.sanitize will bypass the sanitiztion due to the misled fallback handling.

Resolution

Relevant DOM methods are now refrained from clobbering by hard-wiring them into the clobber-check. While not being the ideal solution, there is no easy way to get fresh copies of them.

DOM-01-004 Mutation on XML Namespaces enables sanitization bypass *(Critical)*

Microsoft Internet Explorer (MSIE) 9 incorrectly handles XML Namespaces with innerHTML. This allows attackers to bypass the sanitization on DOMPurify.

When accessing the innerHTML property of an unknown element with xmlns attribute, MSIE 9 will automatically insert the value of xmlns into the Processing Instruction *<? XML:NAMESPACE>* without proper delimiters, causing a structural mutation. The attacker can then inject arbitrary markups with crafted xmlns in an unknown element and hence bypassing the sanitization.

Proof-of-Concept

```
DOMPurify.sanitize("<math xmlns='\"><iframe onload=alert(1)>'></math>")
```

The above call will generate the following output:

```
<?XML:NAMESPACE PREFIX = "[default] "><iframe onload=alert(1)>" NS = ""><iframe
onload=alert(1)>" /><math xmlns='"><iframe onload=alert(1)>'></math>
```

In MSIE 9, the *math* element is recognized as an unknown element, meeting the requirement to perform the attack. In addition, there are other HTML5 elements that are not introduced in MSIE 9, but appear in the allowed element name list of DOMPurify, such as *article*. After the input is mutated, the original value of xmlns breaks the delimiters of the Processing Instruction. Since the as-is implementation does not cover sanitation on it, the payload is able to cause XSS when being injected into an element.

Resolution

Considering that this involves a browser's bug rather than an implementation flaw on DOMPurify, fixing it programmatically is considered unfeasible. After discussing the issue, the DOMPurify team agreed to drop support for MSIE 9 and degrade it to fallback mode instead. The decision was made also due to the fact that there might be other potential hidden issues on MSIE 9.

Miscellaneous Issues

This section covers those noteworthy findings that did not lead to an exploit but might aid an attacker in achieving their malicious goals in the future. Most of these results are vulnerable code snippets that did not provide an easy way to be called. Conclusively, while a vulnerability is present, an exploit might not always be possible.

DOM-01-001 Incorrect fallback handling leads to script termination *(Info)*

An artificial mistake was made in the fallback handling. This breaks DOMPurify on some legacy browsers, namely MSIE 7 or below.

```
if (window.toStaticHTML !== 'undefined' && typeof dirty === 'string') {
        return window.toStaticHTML(dirty);
    }
return dirty;
```

The highlighted expression meant to determine if the user agent provides the *toStaticHTML* method. If so, DOMPurify will fallback to it. However, there was supposed to be a preceding *typeof* operator, otherwise the expression will always be true. As a result, legacy browsers that do not support *toStaticHTML* method will fail to execute the code and yield a *ReferenceError*.

DOM-01-003 Missing clobber-check for elements with name attribute *(Low)*

DOMPurify by default prevents DOM clobbering attack. However, there is a missing check for certain elements that allows clobbering via the name attribute.

```
if(SANITIZE_DOM) {
    if(tmp.name === 'id'
        && (tmp.value in window || tmp.value in document)) {
        clobbering = true;
    }
    if(tmp.name === 'name' && tmp.value in document){
        clobbering = true;
    }
}
```

Normally, only the elements with an id attribute are able to clobber *window* properties. While this holds true, some elements can achieve the same goal using the name attribute. For example, the *form* element. As seen from the above code snippet, such check is performed on the id attribute but not the name attribute. Yet, unlike *document*, clobbering *window* properties is hard, if not impossible. This is because overriding *window* properties is disallowed. There are some scenario, though, where attackers can take advantage of it.

Proof-of-Concept

Consider the following hypothetical situation:

```
<iframe name='foo' src='bar'></iframe>
...
<script>if (foo.password.value == '...') {...}</script>
```

Assume that there is a check that extracts the value of an iframe and compare it to some secret value so that it performs certain action, the attacker can make use of the missing clobber-check to inject a crafted *form* element and replace the original one.

```
elem.innerHTML += DOMPurify.sanitize('<form name=foo><input name=password value=...>',
{SANITIZE_DOM: false});
```

After the *form* element is injected, the attacker can then navigate the iframe and change its name to something else. Eventually, the *form* element successfully clobbers the intended one.

DOM-01-005 Week validation on custom data attribute names *(Low)*

DOMPurify allows custom data attributes. Although it validates the the data attribute names, the validation is insufficient and allows attacker to terminate the script execution.

```
...
ALLOW_DATA_ATTR && tmp.name.match(/^data-[\w-]+/i)
```

The code snippet above validates custom data attribute names. However, such check is inadequate. For example, *<a data-foo”>* can bypass the check and cause DOMPurify to fail.

```
DOMException: Failed to execute 'setAttribute' on 'Element': 'data-foo”' is not a valid
attribute name.
```

Resolution

One possible fix is implement a strict validation according the the specification. Nevertheless, browsers are tolerant on this manner. For example, the specification disallows colon (:) in the attribute name, but some browsers actually allow it. Therefore, the finalized fix is to catch the exception and do nothing in case of invalid attribute names.

Conclusion

DOMPurify has proved itself to be a secure, simple to use, and compatible XSS sanitizer. While several issues were identified during the penetration test, no direct bypass was found on modern browsers. It is commendable that the DOMPurify team and the community are professional in the implementation of DOMPurify, especially the high quality of the source code.

I would like to thank the DOMPurify team for creating this project and the support during the assignment.