DOM FLOW
UNTANGLING THE DOM FOR EASY BUGS
Ahamed Nafeez (@skeptic_fx)

Security Engineer with interest in browsers

Speaker at BlackHat Asia, Hack In The Box, nullc0n, c0c0n.
Overview

Modern web apps and their problems w.r.t pen tests

Hookish! tool and how it works

Dom Flow and its techniques

Few JavaScript / DOM nuances and how to catch them
Today’s web apps

- React
- AngularJS
- Knockout
- Meteor
Today’s state

Classic XSS is already fading away

Static analysis is becoming harder for client side JS code

Frameworks are getting more complex (JSX?)
DOM XSS / Javascript injection

XSS triggered due to client side code (Mostly..)

Most generic class of vulnerability on browser.

**Sources** - Entry point for untrusted data

**Sinks** - Executes untrusted data
The hello world of DOM XSS

https://damnvulnerable.me/domxss/
location_hash_to_window_eval#firstName

```javascript
var hash = document.location.hash //source

firstName = hash.slice(1)

document.write(firstName) //sink
```
## Common Sources / Sinks

### Sources

<table>
<thead>
<tr>
<th></th>
<th>URL</th>
<th>Cookie</th>
<th>referrer</th>
<th>name</th>
<th>postMessage</th>
<th>WebStorage</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTML</td>
<td>1356796</td>
<td>1535299</td>
<td>240341</td>
<td>35466</td>
<td>35103</td>
<td>16387</td>
<td>3219392</td>
</tr>
<tr>
<td>JS</td>
<td>22962</td>
<td>359962</td>
<td>511</td>
<td>617743</td>
<td>448311</td>
<td>279383</td>
<td>1728772</td>
</tr>
<tr>
<td>URL</td>
<td>3798228</td>
<td>2556709</td>
<td>313617</td>
<td>83218</td>
<td>18919</td>
<td>28052</td>
<td>6798743</td>
</tr>
<tr>
<td>CSS</td>
<td>220300</td>
<td>10227050</td>
<td>25062</td>
<td>1328634</td>
<td>2554</td>
<td>5618</td>
<td>11809218</td>
</tr>
<tr>
<td>WebStorage</td>
<td>41739</td>
<td>65772</td>
<td>1586</td>
<td>434</td>
<td>194</td>
<td>105440</td>
<td>215165</td>
</tr>
<tr>
<td>postMessage</td>
<td>451170</td>
<td>77202</td>
<td>696</td>
<td>45220</td>
<td>11053</td>
<td>117575</td>
<td>702916</td>
</tr>
<tr>
<td>Total</td>
<td>5891195</td>
<td>14821994</td>
<td>581813</td>
<td>2110715</td>
<td>516134</td>
<td>552455</td>
<td>24474306</td>
</tr>
</tbody>
</table>

### Sinks


Sebastian Lekies, Ben Stock, Martin Johns
Everyone (Frameworks, Developers, ...) use ‘strings’ in a way that directly or indirectly turns into code.

The DOM specification is rich in doing that.
Direct

eval()

setTimeout

Function(x)()

execScript(x)
Indirect

jQuery’s - $(x)

document.write

Element.setAttribute(x)

Element.innerHTML=x
jQuery - $(x)

$(`#id'), $(`.class'), $(`a`) - Acts as a query selector

$(`<img src="1.png">') - Creates a new IMG element
So why is it hard to pen test them?
Usually they look like this!
Existing tools (DOM XSS)

Dominator Pro - Dynamic taint tracking using Firefox.

Plethora of static analysis tools - Regex pattern match, Parse JS code and analyse.
What can we look for?
All cases of DOM Injection

DOM XSS / Javascript injection

DOM based open redirection

Second order DOM injection (XHR, WebSocket)

WebStorage manipulation
Quirky DOM behaviour

Globally exposed variables in the DOM

DOM Clobbering

Usage of certain methods which could have unforeseen security implications
DamnVulnerable.me is a webapp that is deliberately vulnerable to DOM based attacks.

Its goal is to provide a platform to learn, test and practice DOM based bugs and other exotic cases.
How Hookish works

Inject DomHooks for sources and sinks

Wait for page to load

Track all sources and sinks
Injecting DomHooks

WebApp

DOM

WebApp’s JS

Hookish!

DomHooks

- Ask questions
  Give me all global variables

- Register hooks
document.write()
domhooks.js

Standalone library which selectively registers required DOM properties & methods.

https://github.com/skepticfx/hookish/blob/master/src/js/domHooks.js

Can be used in other tools for performance analysis, hardening DOM, DOM based IDS etc.
DomFlow

Data → Source → Tainted Data

Add source specific flag.
location_hash_12321

Sink → Filter n → Filter 1

Transform, SubString, Change App Logic etc

Look for relevant flags
DomFlow- cookie to innerHTML

Every time a cookie is accessed, the data is tagged with a unique flag - `doc_cookie_12391`

This data may go through various transformations.

When a registered innerHTML receives data with this tag, it marks that as a possible DOM XSS.
Overriding filters

Example: XHR to innerHTML

XHR responses are usually JSON content

```javascript
JSON.parse({'data1': 'value1', 'data2': 'value2'})
```

```javascript
Object.Stringify({'data1': 'value1Flag', 'data2': 'value2Flag'})
```
Boxing strings in JS

```javascript
var str = "hello"

typeof str; // string

str.flag = true;

// JS propagates this string flag in most cases
```
Navigating across the flows
Getting the stack trace in V8 Engine

Dynamically throw the error and filter to remove Hookish! specific stacks

Easily integrates with Chrome’s dev tools and helps analyse vulnerable lines of code
Tracking status of all hooks

[domstorm.skepticfx.com](http://domstorm.skepticfx.com)

Hooking Storage objects,
[http://domstorm.skepticfx.com/modules?id=529d4f84090faf000000000002](http://domstorm.skepticfx.com/modules?id=529d4f84090faf000000000002)
Second order DOM injection

DOM injection where the sources doesn’t flow directly.

Rather, they are fetched from a persistent storage at some point.

XHR/WS response flowing in to sinks
Four Scenarios

The following 4 scenarios talks about bugs/special cases that are often missed while security testing a web app.

Hookish! tool is built to easily find / analyse such bugs
1. Do you check how XHR responses are handled in your application?

Most common issue which pen testers miss / scanners usually ignore.

The choke point is how you treat these data before populating into the DOM (regardless of how you store untrusted input)
XHR response - innerHTML

```javascript
var response = JSON.parse(xhr.responseText);

var description = response.description;

var div = document.getElementById('vulnerableDiv');

div.innerHTML = description;
```
2. DOM Clobbering using Global Variables

Consider an IFrame sandbox which executes arbitrary code.

Exposed global variables can change logic in parent window.
Classic Iframe sandboxing

Trusted Parent window

Untrusted but sandboxed IFrame child

<iframe sandbox="allow-scripts"></iframe>

Defaults to origin ‘null’
About this sandbox

IFrame sandboxes have ‘null’ origin.

The JS in sandboxed IFrame should not interact with the parent Window’s DOM.

Spot the bug and break out of this sandbox

https://damnvulnerable.me/misc/insecure_global_variable
Setting global variables using `window.name`

DOM sets the name of iframe windows to the window object (DOM CLOBBERING)

- Trusted Parent window
  - window name is `SECURE_FLAG`

- Untrusted but sandboxed IFrame child
  - `<iframe sandbox="allow-scripts"></iframe>`
  - `<script>
      name=`SECURE_FLAG`
    </script>`
This sets the global variable SECURE_FLAG in the parent window’s DOM and bypasses the security check.
3. Redirect parent window while opening links in new tab

https://hackerone.com/reports/23386

Works on Chrome and Firefox.
Opening links in new tab

Parent window
<a href="website.com" target="_blank"> </a>

New tab (Can be malicious)
window.opener.location.reload(‘phishing-page.com’)
window.opener should be null always and should not be accessible by another Cross-Domain window
Finding anchor tags with `target=_blank`

Easy to find on static HTML pages.

In modern apps, usually anchor tags are dynamically inserted into the DOM.

Hookish! finds these after the DOM is rendered and all anchor tags are populated.

Not a serious issue most of the times, but depends on where you have these new links.
4. Custom templating engines

var data = {‘name’: ‘mark’, ‘age’: ‘23’}

Welcome to this page, <%= data.mark %>
How would some one write a templating engine using JavaScript?

1. Load the template data object and encode it.

2. Find the template pattern

3. Use string.replace(pattern, matching_data)
A simple templating code

```javascript
var inputHTML = "<img src='PLACEHOLDER'>";

function doTemplating(){

    var input = document.getElementById('id_input').value;

    input = filterInput(input);

    var finalHTML = inputHTML.replace("PLACEHOLDER", input);

    console.log(finalHTML);

    document.write('Your input: </br>' + input);

    document.write(finalHTML);
}
```
The bypass

```
$` onerror=alert(1);//
```
String.prototype.replace

ECMAScript’s String.replace is the culprit

http://www.ecma-international.org/ecma-262/5.1/#sec-15.5.4.11

<table>
<thead>
<tr>
<th>Characters</th>
<th>Replacement text</th>
</tr>
</thead>
<tbody>
<tr>
<td>$$$</td>
<td>$</td>
</tr>
<tr>
<td>$&amp;</td>
<td>The matched substring.</td>
</tr>
<tr>
<td>$`</td>
<td>The portion of string that precedes the matched substring.</td>
</tr>
<tr>
<td>$'</td>
<td>The portion of string that follows the matched substring.</td>
</tr>
<tr>
<td>$n</td>
<td>The \n^{th} capture, where ( n ) is a single digit in the range 1 to 9 and $n is not followed by a decimal digit. If ( n \leq m ) and the ( n )th capture is undefined, use the empty String instead. If ( n &gt; m ), the result is implementation-defined.</td>
</tr>
<tr>
<td>$nn</td>
<td>The \nn^{th} capture, where ( nn ) is a two-digit decimal number in the range 01 to 99. If ( nn \leq m ) and the ( nn )th capture is undefined, use the empty String instead. If ( nn &gt; m ), the result is implementation-defined.</td>
</tr>
</tbody>
</table>
Work in progress

Patching chromium to have V8 level tainting and enable overriding of Objects that are not possible now.
Thanks

More questions

@skeptic_fx