Client-Side Protection Against DOM-based XSS Done Right (TM)

Ben Stock, Sebastian Lekies, Martin Johns
About us

• Ben Stock, Sebastian Lekies, Martin Johns
• Security Researcher at Uni Erlangen, Uni Bochum and SAP
• More and stuff at http://kittenpics.org

About this talk

• Results of a practical evaluation of client-side XSS filtering
• Presentation of numerous bypasses for Chrome's XSSAuditor
• New concept to combat client-side XSS
Cross-Site Scripting

a.k.a. XSS (duh)
The Same-Origin Policy

• **Question:** why can’t attacker.org read the visitors emails from GMail?

• **Answer:** Same-Origin Policy
  • Application boundaries by origin: **protocol**, **domain** and **port**
  • Attacker's code runs in different **origin**
Bypassing the Same-Origin Policy

• Applications process user-provided data
  • May be stored or echoed back

• Data `<script>alert(1)</script>` is actually Code
  • .. interpreted by the victim's browser, executed in the origin of vulnerable application

• Attacker's script code is executed on flawed site
  • ➔ Cross-Site Scripting!

• ➔ We can read your GMails 😊
XSS – what an attacker can do

• Open an alert box!

• Hijack a session
  • Oldest trick in the book: steal their cookies
  • Control victim's browser as he wishes

• Alter content
  • Display fake content or spoof login forms

• Steal your password manager’s passwords
  • See our BlackHat EU Talk for more information 😊

Do everything with the Web app, that you could do – under your ID
Types of XSS

Reflected

```php
<?php
echo "Hello ", @$_GET['name'];
?>
```

```html
<script>
    var name = location.hash.slice(1));
    document.write("Hello "+ name);
</script>
```

Stored

```php
<?php
    $res = mysql_query("INSERT...", @$_GET['message'])
    $res = mysql_query("SELECT...");
    $row = mysql_fetch_assoc($res);
    echo $row['message'];
?>
```

```php
<?php
    $res = mysql_query("INSERT...", @$_GET['message']);
    $res = mysql_query("SELECT...");
    $row = mysql_fetch_assoc($res);
    echo $row['message'];
?>
```

```html
<script>
    var html = location.hash.slice(1));
    localStorage.setItem("message", html);
    var message = localStorage.getItem("message");
    document.write(message);
</script>
```
DOM-based / Client-Side XSS

• **Flaws in client-side code**
  • Data from attacker-controlled source flows to security-sensitive sink
  • Eventually, attacker-controlled data is interpreted as code

• **Detection of client-side XSS**
  • Dynamic analysis: use taint tracking
    • Commercial product DOMinator
  • Static analysis: no idea, we don't do static analysis 😊
Stopping XSS attacks

• **If you are the application’s owner:**
  • Don’t use user-provided data in an unencoded/unfiltered way
  • Use secure frameworks or other magic
  • Use Content Security Policy, sandboxed iframes, ...
Stopping XSS attacks

• If you are the application’s owner:
  • Don’t use user-provided data in an unencoded/unfiltered way
  • Use secure frameworks or other magic
  • Use Content Security Policy, sandboxed iframes, …

• If you are the application's user:
  • Turn of JavaScript
  • Use client-side XSS filter
    • NoScript for Firefox
    • IE ships one
    • Chrome (the "XSS Auditor")
Quick digression: finding a lot of DOM XSS vulns
Finding and exploiting DOMXSS vulnerabilities automatically at scale

- **byte-level taint tracking in Chromium**
  - each character in a string has its source information attached to it

- **Chrome crawling extension**
  - also the interface between taint engine and central server

- **An exploit generator**
  - Taint information + HTML/JavaScript syntax rules
  - Generates exploits automatically
Results (many many cats XSS)

• Ran experiment against Alexa Top 10k
  • Found a total of 1,602 unique vulnerabilities
  • .. On 958 domains

• Auditor turned off at that point
  • Vulnerability exists even if caught

• Reran experiment with Auditor
  • Auditor did not catch all exploits
  • Conducted in-depth analysis into the WHY
Bypassing the XSS Auditor
How the XSS Auditor works

- **HTTP response is parsed**
- **Auditor invoked if dangerous HTML construct is encountered**
  - Only during initial parsing process
  - Only if certain chars are in the request ("", "", "", '''
- **HTTP request is checked for existence of construct**
  - Matching algorithm depends on HTML construct
- **If match is found, payload is "neutered"**
Auditor Matching Rules (simplified)

Inline Scripts

- `<script>alert(1)</script>`

**Matching rule**

- Check whether **content** of script is contained in the request
- ... skipping initial comments and whitespaces
- ... only up to 100 characters
- ... stops if "terminating character" is encountered (#, ?, //, ..)
Auditor Matching Rules (simplified)

**HTML attributes**

- **Event handlers**
  ```html
  <img onerror="alert(1)" src="/doesnot.exist">
  ```

- **Attributes with JavaScript URLs**
  ```html
  <iframe src="javascript:alert(1)"></iframe>
  ```

- **For each parsed attribute**
  - ... check if the attribute contains a **JavaScript URL**
  - ... or whether the attribute is an **event handler**
  - If so, check if the **complete attribute** is contained in the request
Auditor Matching Rules (simplified)
Referencing external content

- `<script src="//attacker.org/script.js"></script>`
- `<embed src="//attacker.org/flash.swf"></embed>`

**Matching rule**
- ... check if *tag name*
- ... and the *complete attribute* is contained in the request
How the XSS Auditor works

- HTTP response is parsed
- Auditor invoked if dangerous
- HTML construct is encountered
  - Only during initial parsing process
  - Only if certain characters are present (' and ')
- HTTP request is checked for existence of construct
  - Matching algorithm depends on the construct
- If match is found, payload is "neutered"
How to bypass the XSS Auditor

• HTTP response is parsed
• Auditor invoked if dangerous HTML construct is encountered
  - Only during initial parsing process
  - Only if certain characters are present in the request (<, >, " and ")
• HTTP request is checked for existence of construct
  - Matching algorithm depends on HTML construct
  - If match is found, payload is "neutered"
How to bypass the XSS Auditor

• HTTP response is parsed
• Auditor invoked if dangerous HTML construct is encountered
  • Only during initial parsing process
  • Only if certain characters are present in request (<, >, " and ")
• HTTP request is checked for existence of construct
  • Matching algorithm depends on HTML construct
• If match is found, payload is "neutered"
How to bypass the XSS Auditor

- **HTTP response is parsed**
- **Auditor invoked if dangerous HTML construct is encountered**
  - Only during initial parsing process
  - Only if certain characters are in the request (<, >, " and ")
- **HTTP request is checked for existence of construct**
  - Matching algorithm depends on HTML construct
- **If match is found, payload is "neutered"**
Avoiding Auditor Invocation
Bypassing Auditor Invocation

- Filter works only for injected HTML
  - not for injected JavaScript
  - eval, setTimeout, ...

- Parsing document fragments
  - innerHTML, insertAdjacentHTML, ...
  - Auditor is off for performance

- Unquoted attribute injection (no <,>," or ")
Bypassing Auditor Invocation (cntd.)

• Various injection techniques do not require HTML
  1. DOM bindings
     • e.g., assigning `script.src`
     • injection into already parsed DOM
  2. Second-order flows
     • e.g. cookies or Web Storage
     • injection vector cannot be found in the request
  3. Alternative data sources
     • e.g. `postMessages`
     • Attack vector enters the page through non-request channel
String-matching issues

Create situations, in which the injected vector does not match the parsed JavaScript
Partial Injections

- Hijack an existing tag, attribute or text

```javascript
document.write("<scr"+"ipt>var urlhash=''"+location.hash.slice(1)+"'</scr"+"ipt>");
```

- `http://vuln.com/partial.html#someValue'; alert(1); //`

```javascript
<script>var urlhash='someValue'; alert(1); //</script>
```
Trailing Content

• **Use existing content to fool Auditor**
  • ... while still resulting in valid JavaScript
  • where "valid" means "will not cause compile-time errors"

```javascript
var width = location.hash.slice(1);
document.write("<img src='img.jpg' width='" + width + "px'/>");
```

• **http://vuln.com/trailing.html#' onload='alert(1);**

```html
<img src='img.jpg' width='' onload='alert(1);px'/>
```

• **Other bypasses**
  • using trailing slashes (Auditor stops search after second slash)
  • Trailing SVG (using semicolon)
Double Injections

- User input used more than once

```javascript
var urlhash = location.hash;
document.write("<img src='1.jpg?hash="+urlhash+"'/>
<img src='2.jpg?hash="+urlhash+"'/>");
```

...double.html#"></script>...void("
Double Injections

```javascript
var urlhash = location.hash;
document.write("<img src='1.jpg?hash="+urlhash+"'/>
<img src='2.jpg?hash="+urlhash+"'/>");

<img src='1.jpg?hash=#foo'/><img src='2.jpg?hash=#foo'/>
Double Injections

var urlhash = location.hash;
document.write("<img src='1.jpg?hash="+urlhash+'"/>
<img src='2.jpg?hash="+urlhash+'"/>");
Bypasses in the wild

• Using our existing infrastructure, we found
  • ... 1,602 DOM-based XSS vulnerabilities
  • ... on 958 domains

• We enhanced our exploit generator to target bypassable vulnerabilities
  • Not targeting DOM bindings, second-order flows or alternative attacks

• Result: 776 of 958 domains susceptible to Auditor bypasses
Doing it the right way
The Auditor's problems

• Problem #1: approximation of data flow
  • string matching

• Problem #2: HTML parser
  • after all, XSS is JavaScript injection

• Problem #3: Never designed to tackle client-side XSS
  • let's fix that
Our proposed solution

• Approximation unnecessarily imprecise for local flows
  • we can use taint tracking instead

• Position inside JavaScript parser
  • after all, XSS is JavaScript injection

• XSS: data is interpreted as code
  • "data" in JavaScript: Literals (Numeric, String, Boolean)

• ➔ Only allow tainted data to generate Literals
Example

 userdata

Declaration

 Identifier: a

 StringLiteral: 'userdata'

```
var userInput = location.hash.slice(1)
eval("var a='" + userInput + "';")
```

```
var a='userdata';
```
Example
userdata';alert(1);//

Declaration
  Identifier: a
  StringLiteral: 'userdata'

ExpressionStmt
  Type: CallExpression
  Callee:
    Identifier: alert
    Arguments:
      Literal: 1
Block policies

• No tainted value may generate anything other than a Literal in the JavaScript engine

• No element that references external resources may have a tainted origin
  • enforced in HTML parser and DOM bindings
  • single exception: same origin as including page
Evaluation
False positives

- Compatibility crawl of Alexa Top10k with policies in place
  - 981,453 URLs, 9,304,036 frames

<table>
<thead>
<tr>
<th>Blocking component</th>
<th>documents</th>
</tr>
</thead>
<tbody>
<tr>
<td>JavaScript</td>
<td>5,979</td>
</tr>
<tr>
<td>HTML</td>
<td>8,805</td>
</tr>
<tr>
<td>DOM API</td>
<td>182</td>
</tr>
<tr>
<td><strong>Sum</strong></td>
<td><strong>14,966 (0.016%)</strong></td>
</tr>
</tbody>
</table>
False positives

• Compatibility crawl of Alexa Top10k with policies in place
  • 981,453 URLs, 9,304,036 frames

<table>
<thead>
<tr>
<th>Blocking component</th>
<th>documents</th>
<th>domains</th>
</tr>
</thead>
<tbody>
<tr>
<td>JavaScript</td>
<td>5,979</td>
<td>50</td>
</tr>
<tr>
<td>HTML</td>
<td>8,805</td>
<td>73</td>
</tr>
<tr>
<td>DOM API</td>
<td>182</td>
<td>60</td>
</tr>
<tr>
<td>Sum</td>
<td>14,966 (0.016%)</td>
<td>183 (1.83%)</td>
</tr>
</tbody>
</table>
False positives

- Compatibility crawl of Alexa Top10k with policies in place
  - 981,453 URLs, 9,304,036 frames

<table>
<thead>
<tr>
<th>Blocking component</th>
<th>documents</th>
<th>domains</th>
<th>exploitable domains</th>
</tr>
</thead>
<tbody>
<tr>
<td>JavaScript</td>
<td>5,979</td>
<td>50</td>
<td>22</td>
</tr>
<tr>
<td>HTML</td>
<td>8,805</td>
<td>73</td>
<td>60</td>
</tr>
<tr>
<td>DOM API</td>
<td>182</td>
<td>60</td>
<td>8</td>
</tr>
<tr>
<td><strong>Sum</strong></td>
<td><strong>14,966 (0.016%)</strong></td>
<td><strong>183 (1.83%)</strong></td>
<td><strong>90</strong></td>
</tr>
</tbody>
</table>
Performance

![Graph showing performance metrics for different browsers and benchmarks.](image)
What to take away?

• **XSS still is a problem**
  • DOM-based XSS on about 10% of the Alexa Top 10k domains

• **Browsers deploy countermeasure to protect users**
  • Chrome arguably best filter

• **Security analysis of the Auditor shows that**
  • ... there are many bypasses, related to both
  • ... invocation and
  • ... string-matching issues

• **We propose new approach to client-side XSS filters**
  • using exact taint information
  • low false positives, some overhead (improvable)
Thank you
visit us at kittenpics.org

<table>
<thead>
<tr>
<th>Ben Stock</th>
<th>Sebastian Lekies</th>
<th>Martin Johns</th>
</tr>
</thead>
<tbody>
<tr>
<td>@kcotsneb</td>
<td>@sebastianlekiess</td>
<td>@datenkeller</td>
</tr>
</tbody>
</table>