Reliable Mitigation of DOM-based XSS

Tobias Mueller

2014-09-07
about:me

- MSc. cand. Dipl. Inf.
- presenting results of diploma thesis / USENIX paper
- ~ 45 min. presentation
- ask immediately, Q&A afterwards
Is it possible to reliably defend against DOM-based XSS without breaking the Web?

- recognise new code from attacker-provided strings
- modified V8’s scanner, WebKit’s strings, and the bindings for Chromium
- evaluated protection, compatibility, and speed
→ yes, with some exceptions
1 Intro
   - Motivation

2 Cross-Site Scripting (XSS)
   - Reflected XSS
   - Stored XSS
   - DOM-based XSS
   - Protection

3 Implementation
   - Taint Tracking
   - Compilation
   - Chromiums Architecture
   - V8

4 Evaluation
   - Protection
   - Compatibility
   - Execution Speed

5 Q&A
Motivation

10,000 feet
Severity of XSS

- 2004: OWASP Top 4
- 2007: OWASP Top 1
- 2010: OWASP Top 2
- 2013: OWASP Top 3
Severity of XSS - 10% of CVEs are XSS

New CVE-ID Format as of January 1, 2014 — learn more

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVE-2014-4017</td>
<td>Cross-site scripting (XSS) vulnerability in the Conversion Ninja plugin for WordPress allows attackers to inject arbitrary web script or HTML via the id parameter to lp/index.php.</td>
</tr>
<tr>
<td>CVE-2014-3974</td>
<td>Cross-site scripting (XSS) vulnerability in filemanager.php in AuraCMS 3.0 and earlier allows attackers to inject arbitrary web script or HTML via the viewdir parameter.</td>
</tr>
<tr>
<td>CVE-2014-3966</td>
<td>Cross-site scripting (XSS) vulnerability in Special:PasswordReset in MediaWiki before 1.19. before 1.21.10, and 1.22.x before 1.22.7, when wgRawhtml is enabled, allows remote attackers to execute arbitrary web script or HTML via an invalid username.</td>
</tr>
</tbody>
</table>
Severity of XSS

XSS is very common and dangerous
Severity of XSS - 2 mio user records

One of the Forum administrators quickly looked at the announcement page, saw nothing wrong and replied to the private message from the attacker saying so. 31 seconds after the Forum administrator looked at the announcement page (and before the administrator even had time to reply to the private message), the attacker logged in as that Forum administrator.

Based on the above and conversations with the vBulletin support staff, we believe the attacker added an XSS attack in the announcement they posted which sent the cookies of any visitor to the page to the attacker.

Once the attacker gained administrator access in the Forums they were able to add a hook through the administrator control panel. Hooks in vBulletin are arbitrary PHP code which can be made to run on every page load. The attacker installed a hook allowing them to execute arbitrary PHP passed in a query string argument. They used this mechanism to explore the environment and also to upload and install two widely available PHP shell kits. The attacker used these shell kits to upload and run some custom PHP code to dump the ‘user’ table to a file on disk which they then downloaded.
Overview of section 2

1. Intro
   - Motivation

2. Cross-Site Scripting (XSS)
   - Reflected XSS
   - Stored XSS
   - DOM-based XSS
   - Protection

3. Implementation
   - Taint Tracking
   - Compilation
   - Chromiums Architecture
   - V8

4. Evaluation
   - Protection
   - Compatibility
   - Execution Speed

5. Q&A
Cross-Site Scripting (XSS)
Code Execution in the victim’s browser

- (JavaScript) Code execution
- use all browser APIs
- use Web app in the name of the user
- obtain credentials
- spy on behaviour
Reflected XSS

```php
// returning unsanitised data
echo $_GET['bar'];
?
```

Attack: http://foo/?bar=<script>alert("xss")</script>
Reflected XSS

2. Browser sends a GET request to `http://vuln/?param=XSS`.
3. Web Server responds with an HTML page containing an XSS payload.
4. Victim's browser receives the payload and executes it.
Reflected XSS

Introduces XSS Implementation Evaluation Q&A

Reflected XSS  Stored XSS  DOM-based XSS  Protection

Reflected XSS

Victim

Browser

Attacker

Web Server

http://vuln/?param=XSS

GET vuln/?param=XSS

<html>...</html>...XSS...<html/

Sensitive Data
Reflected XSS

1. The attacker sends a request to the vulnerable web server with a URL containing a parameter that includes an XSS payload:
   
   \[ \text{http://vuln/?param=XSS} \]

2. The web server processes the request and returns the webpage with the payload, which is reflected on the webpage:
   
   \[ \langle \text{html} \rangle \ldots \langle \text{XSS} \rangle \ldots \langle /\text{html} \rangle \]

3. The payload is reflected back to the victim's browser as part of the HTML content:

4. The victim's browser receives the reflected payload and renders it, causing the XSS attack to occur.

Sensitive Data: The敏感数据 is passed from the attacker to the victim via the reflected XSS payload.
Reflected XSS

1. Victim sends sensitive data to the Web Server.
2. Web Server receives the request and sends a response to the Victim.
3. Victim receives the response containing `<html>`...`XSS`...`</html>`.
4. Victim sends the response to the Attacker.

http://vuln/?param=XSS
Stored XSS

<?php // store.php
store_in_db ('some_key', $_POST['bar']);
?>

<?php // retrieve.php
// returning unsanitised data
echo get_from_db ('some_key');
?>

Attack:

1. POST http://foo/?bar=<script>alert(1)</script>
Stored XSS

1. POST vuln/?param=XSS
3. GET /
4. <html>...XSS...</html>
5. Sensitive Data
Stored XSS

Victim → Browser

Attacker → Web Server

1. POST vuln/?param=XSS
3. GET /
4. <html>...XSS...</html>
5. Sensitive Data

Reflected XSS, Stored XSS, DOM-based XSS, Protection
Stored XSS

1. POST vuln/?param=XSS
3. GET /
4. "<html>...XSS...</html>"
5. Sensitive Data
Stored XSS

- Victim
  - Web Server
  - Browser
  - Attacker

- Web Server
  - Attacker
  - Browser
  - Victim

1. POST `vuln/?param=XSS`
3. GET `/`
4. `<html>...XSS...</html>`
5. Sensitive Data
Stored XSS

1. Attacker
   - POST vuln/?param=XSS

2. Web Server
   - http://vuln/
   - Sensitive Data

3. Browser
   - GET /
   - <html>...
   - XSS
   - </html>

4. Victim
   - Browser

5. Attacker
   - POST vuln/?param=XSS

Sensitive Data
DOM-based XSS

<HTML>
<TITLE>Welcome!</TITLE>
Hi
<SCRIPT>
var pos = document.URL.indexOf("name=")+5;
document.write(document.URL.substring(pos,document.URL.length));
</SCRIPT>
<br/>
Welcome to our system...
</HTML>

Attack:
http://vuln/welcome.html#name=<script>alert(1)</script>
DOM-based XSS

- Neither Stored- nor Reflected-XSS !!111elfeins

- Client-side vulnerability

- Read from (attacker controlled) properties of the loaded document
  - `document.location, window.name, etc...`

- Write to security sensitive sinks
  - `eval, document.write, etc...`

```
eval(document.location.hash.substring(1))
http://lolcathost:8000/#alert(1)
```
DOM-based XSS

1. Attacker
2. Web Server
3. GET /
4. <html>...
   <script>...
   </html>

Victim

Browser

Sensitive Data

http://vuln/#XSS
DOM-based XSS

1. Attacker sends sensitive data via POST to the web server at http://vuln/#XSS.
2. The web server returns a response containing a script tag.
3. The script tag executes in the browser, injecting malicious code.
4. The victim browser, upon loading the malicious script, executes the attacker's code.

Diagram:
- Victim
- Browser
- Attacker
- Web Server
DOM-based XSS

1. The attacker sends the victim to a vulnerable web page via a malicious link.
2. The web server responds with a web page containing JavaScript code.
3. The victim visits the malicious link.
4. The script runs in the victim's browser, accessing sensitive data.

http://vuln/#XSS

GET /
DOM-based XSS

1. **Attacker** sends **Sensitive Data** to **Victim**.

2. **Victim** visits the malicious website containing the **Sensitive Data**.

3. **Web Server** returns the HTML content to the **Browser**.

4. **Browser** executes the malicious JavaScript contained in the HTML, allowing the attacker to steal the **Sensitive Data**.
DOM-XSS protection mechanisms

- server-side solutions
  - inappropriate as data does not leave the client
- turn off JavaScript . . .
  - breaks the Web
- WebKit’s XSS Auditor
  - “Only” smarter string matching
  - inherent weaknesses, e.g. in WebKit, not V8 → eval
- Block tainted JavaScript code
  - too coarse grained, breaks the Web:
    - `var name=d.URL.substring(d.URL.indexOf("name="))`
  → use knowledge of data flows to only allow data values and forbid code
Interlude: Recap

- XSS is a problem
- DOM-XSS is a client-side problem
- The client is the appropriate place for a fix
- The idea is to observe data flows to allow literals but block new code
Overview of section 3

1 Intro
   - Motivation

2 Cross-Site Scripting (XSS)
   - Reflected XSS
   - Stored XSS
   - DOM-based XSS
   - Protection

3 Implementation
   - Taint Tracking
   - Compilation
   - Chromiums Architecture
   - V8

4 Evaluation
   - Protection
   - Compatibility
   - Execution Speed

5 Q&A
Taint Tracking

- Annotate data and track it throughout
- `perl -T`
- `navigator.taintEnabled()`
Automated data flow detection

WebKit

DOM Tree

Store taint t1 in node document.title

Tainted value t1

Initialize v8 with JS code

V8 Javascript Engine

x = document.location.hash

Pass on taint inside v8

document.title = x

y = document.title

We still know the actual, unsecure source! (document.location.hash)
Compilation

Source Code

Compiler

Lexer

Parser

Code Generator

Program

\[
\text{var } \text{foo } = \text{ "bar" } ;
\]
Compilation

```
var foo = "bar";
VAR ID EQ STR SEMI
```
Compilation

```
var foo = "bar" ;

VAR ID EQ STR SEMI

VariableStatement:
  var VariableDeclarationList
```
Compilation

var foo = "bar" ;

VAR ID EQ STR SEMI

VariableStatement:
  var VariableDeclarationList

sub esp, 4
Architecture

Chromium

∼ 12,000k SLoC

WebKit
HTML Rendering

∼ 800k SLoC C++

V8
JavaScript Execution

∼ 700k SLoC C++
* If it doesn't look good, you will get a pointer to the second container
* back. You may preload that container with an ILLEGAL token.
* The reason for that design is a bit wacky: I believe it more safe
* as the container live on the stack of the caller. So they won't get
* tampered with if they were on the callee's stack and some other functions
* run in between. Although I have no data to back that up.
*/

```cpp
Token::Container* Scanner::CheckTaint(Token::Container& current_container,
                                       Token::Container& illegal_container) {
    Token::Container* return_container_p = nullptr;
    const Token::Value current_token = current_container.value();
    const bool is_tainted = current_container.is_tainted();

    if (is_tainted) {
        OS::Print("Tainted Token in scanner!!1 %s (%d)\n",
                   Token::String(current_token), is_tainted);
        // We check the token's value and decide whether to allow or not
        switch (current_token) {
        case Token::STRING:
        case Token::TRUE_LITERAL:
        case Token::FALSE_LITERAL:
        case Token::NUMBER:
            // It may be useful to allow this to go through untaintedly.
            // We cannot call Token::String(EOS) and we prevent to get in
            // trouble if we wanted to report that token.
        case Token::EOS:
            // We have only so many tokens that we want to allow for now.
            return_container_p = &current_container;
            break;
        default:
            // All others we are replacing with an illegal token.
            return_container_p = &illegal_container;
            break;
        }
    } else {
        return_container_p = &current_container;
    }
    return return_container_p;
```
Overview of section 4

1 Intro
   - Motivation

2 Cross-Site Scripting (XSS)
   - Reflected XSS
   - Stored XSS
   - DOM-based XSS
   - Protection

3 Implementation
   - Taint Tracking
   - Compilation
   - Chromiums Architecture
   - V8

4 Evaluation
   - Protection
   - Compatibility
   - Execution Speed

5 Q&A
Evaluation

- Protection: Test cases and vulnerable top 10000 Web apps (≈ 8% vuln.)
- Compatibility: Test cases and top 10000
- Execution speed: standard benchmarks against baseline
Protection - Setup

**in vitro (test suite)**
- Pre-existing test suite

**in vivo (real world large-scale study)**
- Find existing vulnerability
  - Create witness inputs
  - Test with proposed implementation
  - \# rejections \(\Rightarrow\) protection capability

- Sites with DOM-based XSS vulnerabilities
- Regression test
## Protection - Results

<table>
<thead>
<tr>
<th></th>
<th>Without</th>
<th>XSS Auditor</th>
<th>Taint Aware browser</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exploitable Domains</td>
<td>757</td>
<td>545</td>
<td>0</td>
</tr>
<tr>
<td>Protection Rate</td>
<td>0%</td>
<td>28.01%</td>
<td>100%</td>
</tr>
</tbody>
</table>

**Table:** Protection Capabilities of the XSS Auditor and the taint browser
Compatibility - Setup

1. Obtain list of top 10000 Web sites
2. Browse Web sites
3. Collect block reports
4. #rejects ≈ compatibility
Compatibility - Results

8 (out of 10000) wrongfully blocked Web apps: 
al.com, blogger.com, elpais.com, google.com, ixian.cn, miami.com, 
mlive.com, toyota.jp
Execution Speed - Setup

- Select Browser
  - Chromium Vanilla
  - Chromium Taint
  - Firefox
  - Internet Explorer
  - Opera

- Select Benchmark
  - Kraken
  - Sunspider
  - Dromaeo
  - Octane

- Run Experiment
- Run Benchmark
  - create fresh profile
  - run benchmark
  - collect results

Flow:
- 10 times
- 20 times
Execution Speed - Results - 23% vs. 39%, 49%, and 63%

- **Patched Chrome**: 5.00
- **Firefox**: 1.59
- **Internet Explorer**: 1.12
- **Epiphany**: 1.14
- **Opera**: 1.09
- **Kraken Sunspider Dromaeo Octane v2**: 1.08, 1.36, 1.75, 1.37, 2.10, 0.50, 1.89, 1.48, 1.63, 0.69, 0.93, 1.66, 5.00, 0.91, 2.45, 2.68

---

<table>
<thead>
<tr>
<th>Browser</th>
<th>Kraken</th>
<th>Sunspider</th>
<th>Dromaeo</th>
<th>Octane v2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Patched Chrome</td>
<td>1.59</td>
<td>1.12</td>
<td>1.14</td>
<td>1.09</td>
</tr>
<tr>
<td>Firefox</td>
<td>2.10</td>
<td>1.36</td>
<td>1.75</td>
<td>1.37</td>
</tr>
<tr>
<td>Internet Explorer</td>
<td>1.63</td>
<td>0.91</td>
<td>0.93</td>
<td>1.09</td>
</tr>
<tr>
<td>Epiphany</td>
<td>1.08</td>
<td>0.69</td>
<td>0.93</td>
<td>1.48</td>
</tr>
<tr>
<td>Opera</td>
<td>2.00</td>
<td>0.75</td>
<td>1.66</td>
<td>2.10</td>
</tr>
</tbody>
</table>

**Slowdown factor**

- Kraken: 1.59
- Sunspider: 1.12
- Dromaeo: 1.14
- Octane v2: 1.09
Overview of section 5

1. Intro
   - Motivation

2. Cross-Site Scripting (XSS)
   - Reflected XSS
   - Stored XSS
   - DOM-based XSS
   - Protection

3. Implementation
   - Taint Tracking
   - Compilation
   - Chromiums Architecture
   - V8

4. Evaluation
   - Protection
   - Compatibility
   - Execution Speed

5. Q&A
Client-side protection mechanism against DOM-XSS
Thorough evaluation of the proposed implementation
Review of existing XSS protection mechanisms

to be read in
“Precise Client-side Protection against DOM-based XSS”,

Questions?