Unraveling some of the Mysteries around DOM-based XSS

Dave Wichers
Aspect Security, COO
OWASP Board Member
OWASP Top 10 Project Lead

dave.wichers@aspectsecurity.com
COO / Cofounder – Aspect Security

- 20+ Years in Application Security Consulting
- Code Review / Pen Testing 100s of Applications
- Taught 100s of Secure Coding Courses
- Helped numerous organizations improve their ability to produce secure software
- Board Member, OWASP Top 10 Lead, ASVS Coauthor
  - Open Web Application Security Project (OWASP)

Aspect Security (www.aspectsecurity.com)

- Focused exclusively on Application Security
- Code Review / Pen Testing of Applications
- Secure Coding Training (over a dozen courses including secure mobile)
- Secure Development Lifecycle consulting
  - Tools, Techniques, Processes for Developing Secure Code
- Contrast – Intrinsic Application Security Testing (IAST) for JavaEE

https://www.aspectsecurity.com/Contrast
Search-field input is often reflected back to user.

\[\text{<script>alert(document.cookie)</script>}\]

Site reflects the script back to user where it executes and displays the session cookie in a pop-up.
Types of Cross-Site Scripting (XSS)

<table>
<thead>
<tr>
<th>Cross-Site Scripting Vulnerabilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type 2: Stored XSS (aka Persistent)</td>
</tr>
<tr>
<td>Type 1: Reflected XSS (aka non-Persistent)</td>
</tr>
<tr>
<td>Type 0: DOM-Based XSS</td>
</tr>
</tbody>
</table>

Sources: [https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)](https://www.owasp.org/index.php/Cross-site_Scripting_(XSS))  
[http://projects.webappsec.org/w/page/13246920/Cross_Site_Scripting](http://projects.webappsec.org/w/page/13246920/Cross_Site_Scripting)

“There’s also a third kind of XSS attacks - the ones that do not rely on sending the malicious data to the server in the first place!” Amit Klein – Discoverer of DOM-Based XSS

XSS – Illustrated

1: DOM-Based Reflected XSS

2: DOM-Based Stored XSS

3: Reflected XSS (via AJAX Request/Response)

4: Stored XSS (via AJAX Message Retrieval)

JavaScript Form

Sent Message: Bob
Message Body: What’s for dinner?
<script>alert(1)</script>

User Input

Store Message on Server

HTML 5 Local Storage
PROBLEM: Current terms are overlapping and difficult to understand

SOLUTION: Define types of XSS based on where the dangerous sink is. This is the right approach because it’s very clear and helps the developer know where the problem that needs fixing is.

- **Server XSS** – Untrusted data is included in generated HTML.
  - Best defense is context-sensitive output encoding.

- **Client XSS** – Untrusted data is added to DOM through unsafe JavaScript call.
  - Best defense is using safe JavaScript APIs.

Either can be Stored or Reflected

Untrusted data can come from client or server
### DOM-Based XSS

- **DOM-Based XSS** is simply Client XSS where the data source is from the client rather than the server.

- **Stored vs. Reflected** only affects the likelihood of successful attack, not nature of vulnerability or defense.
Logic is Migrating from Server to Client...

Server-side vs. Client-side LoC in popular web applications in 2005 and in 2010

Source: IBM
And How Vulnerable are Apps Today?

Using IBM’s JavaScript Security Analyzer (JSA), IBM tested Fortune 500 + Top 178 Sites and found

40%

Vulnerable to Client-side JavaScript vulnerabilities,

90%

of which was caused by 3rd party JavaScript Libraries

Source: IBM
What’s the Risk of Client/DOM-Based XSS?

- XSS Risk from OWASP Top 10

- Stored XSS attack more likely to succeed than reflected but impact is the same

- Risks are the SAME for Server and Client XSS

- Detectability is lower for Client XSS as it’s harder for attackers (and defenders) to find
For: http://www.vulnerable.site/welcome.html

```html
<html>
<title>Welcome!</title>

Hi

<script>
var pos=document.URL.indexOf("name=")+5;

document.write(document.URL.substring(pos,document.URL.length));

</script>

Welcome to our system ...
</html>
```

Notice that both data source and dangerous sink are on the client.

src: http://projects.webappsec.org/w/page/13246920/Cross Site Scripting
Why is finding Client XSS So Hard?

- **Document Object Model**

- Existing JavaScript can update the DOM and new data can also contain JavaScript

- It's like trying to find code flaws in the middle of a dynamically compiled, running, self modifying, continuously updating engine while all the gears are spinning and changing.

- **Self modifying code** has basically been banned in programming for years, and yet that’s exactly what we have in the DOM.

“Manual code review is hell – have you seen JavaScript lately?” Ory Segal, IBM
How do we make detecting/avoiding Client XSS more of a Science?

Better Understanding of

• Dangerous Sources
• Propagators (not covered here)
• Unsafe Sinks
• Defense Techniques
Dangerous Sources (of Browser Input)

#1 New page is created/updated in DOM.
Scripts can access any of this data.

#2
```
function handleReloadContents() {
    if (httpObj.readyState == 4 || httpObj.readyState=="complete") {
        var result = document.getElementById("mainBody");
        result.innerHTML = httpObj.responseText;
    }
}
```

#3 Request/Window Attributes
- `document.*`
- `window.*`

#4 Directly from the user.
e.g., `onenter()`, `onclick()`, submit button ...

#5 JavaScript
- CSS

3rd Party Server
- 3rd Party Content in Same Origin
- Standard HTML / JavaScript / CSS Response
- AJAX (XHR) Response

New page is created/updated in DOM.
Scripts can access any of this data.

Directly from the user.
e.g., `onenter()`, `onclick()`, submit button ...

#1 New page is created/updated in DOM.
Scripts can access any of this data.

#2
```
function handleReloadContents() {
    if (httpObj.readyState == 4 || httpObj.readyState=="complete") {
        var result = document.getElementById("mainBody");
        result.innerHTML = httpObj.responseText;
    }
}
```

#3 Request/Window Attributes
- `document.*`
- `window.*`

#4 Directly from the user.
e.g., `onenter()`, `onclick()`, submit button ...

#5 JavaScript
- CSS
Dangerous Request/Window Attributes

- Page: [https://code.google.com/p/domxsswiki/wiki/Sources](https://code.google.com/p/domxsswiki/wiki/Sources)
- For: Browsers: IE 8, Firefox 3.6.15 – 4, Chrome 6.0.472.53 beta, Opera 10.61 (Safari data TBD)
- Describes return values for document.URL / documentURI / location.*
  ([https://code.google.com/p/domxsswiki/wiki/LocationSources](https://code.google.com/p/domxsswiki/wiki/LocationSources))

```
scheme://user:pass@host/path/to/page.ext/Pathinfo;semicolon?search.location=value#hash=value&hash2=value2
```

Example: `http://host/path/to/page.ext/test<a""%0A`=%20+%20>;test<a""%0A`=%20+%20>;<test<a""%0A`=%20+%20>;<#test<a""%0A`=%20+%20>;

document.url output:
```
http://host/path/to/page.ext/test%3Ca%22'%0A%60=%20+%20%3E;test%3Ca%22'%0A%60=%20+%20%3E?test%3Ca""%0A`=%20+%20>;#test%3Ca""%0A`=%20+%20>;
```

- Similar info for other direct browser data sources including
  - window.name ([https://code.google.com/p/domxsswiki/wiki/TheWindowNameSource](https://code.google.com/p/domxsswiki/wiki/TheWindowNameSource))
Some Dangerous JavaScript Sinks

Direct execution

• eval()
• window.execScript()/function()/setInterval()/setTimeout()
• script.src(), iframe.src()

Build HTML/Javascript

• document.write(), document.writeln()
• elem.innerHTML = danger, elem.outerHTML = danger
• elem.setAttribute("dangerous attribute", danger) – attributes like: href, src, onclick, onload, onblur, etc.

Within execution context

• onclick()
• onload()
• onblur(), etc.

Gleaned from: https://www.owasp.org/index.php/DOM_based_XSS_Prevention_Cheat_Sheet
Some Safe JavaScript Sinks

Setting a value
- `elem.innerText(danger)`
- `formfield.val(danger)`
- `node.textContent(danger)`
- `document.createTextNode(danger)`

Safe JSON parsing
- `JSON.parse(danger)`
- rather than `eval()`
Popular JavaScript Library #1: jQuery

Note: This may not be the whole list!!

### Dangerous jQuery 1.7.2 Data Types

<table>
<thead>
<tr>
<th>CSS</th>
<th>Some Attribute Settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTML</td>
<td>URL (Potential Redirect)</td>
</tr>
</tbody>
</table>

### jQuery methods that directly update DOM or can execute JavaScript

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(() or jQuery()</td>
<td>.attr()</td>
</tr>
<tr>
<td>.add()</td>
<td>.css()</td>
</tr>
<tr>
<td>.after()</td>
<td>.html()</td>
</tr>
<tr>
<td>.animate()</td>
<td>.insertAfter()</td>
</tr>
<tr>
<td>.append()</td>
<td>.insertBefore()</td>
</tr>
<tr>
<td>.appendTo()</td>
<td>Note: .text() updates DOM, but is safe.</td>
</tr>
</tbody>
</table>

### jQuery methods that accept URLs to potentially unsafe content

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jQuery.ajax()</td>
<td>jQuery.post()</td>
</tr>
<tr>
<td>jQuery.get()</td>
<td>load()</td>
</tr>
<tr>
<td>jQuery.getScript()</td>
<td></td>
</tr>
</tbody>
</table>
jQuery Security Tests

The following frames run a series of tests to verify the security of different elements of jQuery v1.7.2. These tests are focused on DOM-Based XSS attacks and are intended to identify where client-side security controls are needed. Failed tests indicate that the element being tested should be considered unsafe when used with tainted data and client-side output encoding should be used to sanitize any potential attacks.

Methods

About jQuery Methods

The following tests each of the jQuery methods to determine which method signatures are unsafe.

1. $(String) or jQuery(String) - Test if an attacker can execute arbitrary code if tainted data is injected into the $(String) method. $(String) passed to method: jQuery(String) or $(String)
   
   **Expected:** ""
   
   **Result:** ""FAIL"
   
   **Diff:** """"FAIL"
   
   **Source:**
   
   at checkFail (file:///Z:/dwicher/Documents/p42/aspect/Customer/Vanguard/ISCI/2012-05-JQuery%20ISCI/working/Matts%20new%20jQuery%20test%20suite/jQSecurity/tests/core.js:34:5)
   
   at file:///Z:/dwicher/Documents/p42/aspect/Customer/Vanguard/ISCI/2012-05-JQuery%20ISCI/working/Matts%20new%20jQuery%20test%20suite/jQSec
   
   urity/tests/core.js:38:29

2. $(Element).addClass(String) - Test if an attacker can execute arbitrary code if tainted data is injected into the addClass() method. $(Element).addClass(String) passed to method: $(Element).addClass(String)

3. $(Element).append(String) - Test if an attacker can execute arbitrary code if tainted data is injected into the append() method. $(Element).append(String) passed to method: $(Element).append(String)
Example

- jQuery(danger) or $(danger)
  - Immediately evaluates the input!!
  - E.g., $('"<img src=x onerror=alert(1)>"')

Same safe examples

- .text(danger), .val(danger)

Some serious research needs to be done to identify all the safe vs. unsafe methods

- There are about 300 methods in jQuery

We’ve started a list at:
http://code.google.com/p/domxsswiki/wiki/jQuery
What about other Popular JavaScript Libraries?

- YUI Library
- MooTools
- Ext JS 3.4
- script.aculo.us
- dojō
- <swfobject>
## Defense Techniques
### Server XSS vs. Client XSS

<table>
<thead>
<tr>
<th>Technique</th>
<th>Server XSS</th>
<th>Client XSS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avoid untrusted data</td>
<td>Don’t include in response</td>
<td>Don’t accept as input</td>
</tr>
<tr>
<td>Context Sensitive Output Escaping</td>
<td>Server-Side</td>
<td>Client-Side</td>
</tr>
<tr>
<td>Input Validation (Hard!)</td>
<td>Server-Side</td>
<td>Client-Side</td>
</tr>
<tr>
<td>Avoid JavaScript Interpreter</td>
<td>Can’t. Page always interpreted</td>
<td>Avoid unsafe JavaScript / JS Library methods</td>
</tr>
</tbody>
</table>
Primary XSS Defense: Context Sensitive Output Escaping

**HTML Element Content**
(e.g., `<div>` some text to display `</div>`)

**HTML Attribute Values**
(e.g., `<input name='person' type='TEXT' value='defaultValue'>`)

**JavaScript Data**
(e.g., `<script>` some javascript `</script>`)

**HTML Style Property Values**
(e.g., `.pdiv a:hover {color: red; text-decoration: underline}`)

**URI Attribute Values**
(e.g., `<a href='javascript:toggle('lesson')'>`

#1: ( &, <, >, " ) → &entity; ( ' , / ) → &xHH;
ESAPI: `encodeForHTML()`

#2: All non-alphanumeric < 256 → &xHH
ESAPI: `encodeForHTMLAttribute()`

#3: All non-alphanumeric < 256 → \xHH
ESAPI: `encodeForJavaScript()`

#4: All non-alphanumeric < 256 → \HH
ESAPI: `encodeForCSS()`

#5: All non-alphanumeric < 256 → %HH
ESAPI: `encodeForURL()`

See: [www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet](http://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet) for more details
## Client-Side Context Sensitive Output Escaping

<table>
<thead>
<tr>
<th>Context</th>
<th>Escaping Scheme</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTML Element</td>
<td>( &amp;, &lt;, &gt;, &quot; ) → &amp;entity; ( ', / ) → &amp;#xHH;</td>
<td><code>$ESAPI.encoder().encodeForHTML()</code></td>
</tr>
<tr>
<td>HTML Attribute</td>
<td>All non-alphanumeric &lt; 256 → &amp;#xHH</td>
<td><code>$ESAPI.encoder().encodeForHTMLAttribute()</code></td>
</tr>
<tr>
<td>JavaScript</td>
<td>All non-alphanumeric &lt; 256 → \xHH</td>
<td><code>$ESAPI.encoder().encodeForJavaScript()</code></td>
</tr>
<tr>
<td>HTML Style</td>
<td>All non-alphanumeric &lt; 256 → \HH</td>
<td><code>$ESAPI.encoder().encodeForCSS()</code></td>
</tr>
<tr>
<td>URI Attribute</td>
<td>All non-alphanumeric &lt; 256 → %HH</td>
<td><code>$ESAPI.encoder().encodeForURL()</code></td>
</tr>
</tbody>
</table>


Identical encoding methods also built into a jQuery-encoder: [https://github.com/chrisisbeef/jquery-encoder](https://github.com/chrisisbeef/jquery-encoder)

Note: Nested contexts like HTML within JavaScript, and decoding before encoding to prevent double encoding are other issues not specifically addressed here.
Client-Side Input Validation

- Input Validation is HARD
- We recommend output escaping instead
- But if you must, it usually looks something like this:

```javascript
function check5DigitZip(value) {
    var re5digit=/^\d{5}$/ //regex for 5 digit number
    if (value.search(re5digit)==-1) //if match failed
        return false;
    else return true;
}
</script>

Example inspired by:
http://www.javascriptkit.com/javatutors/re.shtml
Avoid JavaScript Interpreter

This is what I recommend for Client XSS

Trick is knowing which calls are safe or not

We covered some examples of safe/unsafe sinks but serious research needs to be done here
DOM-Based XSS While Creating Form

Vulnerable Code:

```javascript
var html = ['<form class="config">',
    '<fieldset>',
    '<label for="appSuite">Enter URL:</label>',
    '<input type="text" name="appSuite" id="appSuite" value="', options.appSuiteUrl, '">
    ',
    '</fieldset>',
    '</form>'];

dlg = $(html).appendTo($('body'));
...
```

DOM Result: <input type="text" name="appSuite" id="appSuite"
value="http://a.com/foo?" onblur="alert(123)"/>

Attack URL Value: http://a.com/foo?" onblur="alert(123)"
Fix #1: Input Validation

Fix #1:

\[
\text{regexp} = \begin{array}{l}
/\http(s)://(/\w+:{0,1}\w*@)?(\S+)\:(?:[0-9]+)?(/\|\|(/\w#!:.?+=&%@\-\/)\|))?/;
\end{array}
\]

buttons: { 'Set': function () {
    var u = $.trim(appSuite.val());
    if (!regexp.test(u) || u.indexOf('"') >= 0) {
        Util.ErrorDlg.show('Please enter a valid URL.');
        return;
    } ...  

Note: This is client-side input validation. However, in this particular instance, the data was sent to the server and then back to the client. As such, this defense could be bypassed. But the next one couldn’t...
Fix #2:

```javascript
var html = ['<form class="config">',
    '<fieldset>',
    '<label for="appSuite"> Enter URL:</label>',
    '<input type="text" name="appSuite"
        id="appSuite"/>',
    '</fieldset>',
    '</form>'];

dlg=$(html).appendTo($('body')); // No user input in HTML to be interpreted

appSuite.val(options.appSuiteUrl); // Set value safely
...
```
namespace.events = {
    spyOn: function(selector, eName) {
        var handler = function(e) {
            data.spiedEvents[[selector, eName]] = e;
        };

        $(selector).bind(eName, handler);
        data.handlers.push(handler);
    }, ...
}

Passing data as a selector to the $( ) function is VERY common. Problem is that $( ) is also a JavaScript interpreter, not just a way of getting references to things.
Current Defense: a) Validate selector to make sure its safe, or
    b) JavaScript encode any < it may contain.
Future: Maybe I can convince jQuery to make a safe selector handler method.
<table>
<thead>
<tr>
<th>Techniques for Finding Client XSS #1</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Test like normal XSS in obvious inputs</strong></td>
</tr>
<tr>
<td>• Step 1: Enter test script: dave&lt;script&gt;alert(1)&lt;/script&gt;</td>
</tr>
<tr>
<td>• Step 2: Inspect response and DOM for ‘dave’</td>
</tr>
<tr>
<td>• Step 3: If found, determine if encoding is done (or not needed)</td>
</tr>
<tr>
<td>• Step 4: Adjust test to actually work if necessary</td>
</tr>
<tr>
<td>• E.g., dave&quot; /&gt;&lt;script&gt;alert(1)&lt;/script&gt;</td>
</tr>
<tr>
<td>• dave&quot; onblur=&quot;(alert(2))</td>
</tr>
</tbody>
</table>

Tools: Normal manual Pen Test Tools like WebScarab/ZAP/Burp can be used here
Automated scanners can help, but many have no Client/DOM-Based XSS test features
### Inspect JavaScript loaded into DOM

1. **Step 1:** Look for references to user controlled input
   - Remember 5 browser sources referenced previously?
2. **Step 2:** Follow data flow to dangerous sinks
   - Lots of dangerous sinks mentioned before
3. **Step 3:** If data is properly validated or encoded before reaching dangerous sink (it’s safe)
   - Validation/encoding could occur server or client side

**NOTE:** THIS IS **REALLY** HARD!!

Browser Plugins REALLY USEFUL: Firebug, Chrome Developer Tools
Free Tools: DOMinator, DOM Snitch, Ra.2 try to automate this type of analysis
IBM’s AppScan does this too
XSS Flaws Aren’t Always Easily Exploited

- Scenario 1: Reflected XSS protected with CSRF Token
  - Attacker workaround: Clickjacking vulnerable page
- Scenario 2: DOM-Based XSS starting with user input to form
  - Can’t force user to fill out form right? Yes – Clickjacking
  - Or, if Client XSS, but data passes through server:
    - Force the request to the server, instead of filling out the form. Works for Stored XSS, but not Reflected XSS, since XHR won’t be waiting for response.
Its not just Client/DOM-Based XSS

Unchecked Redirect

- `window.location.href = danger, window.location.replace()`

HTML 5 Shenanigans

- Client-side SQL Injection
- ‘Pure’ DOM-Based Stored XSS (Discussed before)
- Local storage data left and data persistence (super cookies)
- Notification API Phishing, Web Storage API Poisoning, Web Worker Script URL Manipulation, (all coined by IBM)
- Web Sockets ???

Lots more ... 😞
DOMinador – by Stefano DiPaola

- Firefox Plugin
- Works by adding taint propagation to strings within the browser
- Update just released
  - Adds support for some HTML5 features like cross domain requests, new tags, etc.
DOM Snitch

- Experimental tool from Google (Dec, 2011)
- **Real-time:** Flags DOM modifications as they happen.
- **Easy:** Automatically flags issues with details.
- Really Easy to Install
- Really Easy to Use

http://code.google.com/p/domsnitch/

---

<table>
<thead>
<tr>
<th>Id</th>
<th>URL</th>
<th>Type</th>
<th>Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>23+</td>
<td><a href="http://www.facebook.com/dave.wichers?ref=tn_tnmn">http://www.facebook.com/dave.wichers?ref=tn_tnmn</a></td>
<td>HTTP headers</td>
<td>Hide/Star this record</td>
</tr>
<tr>
<td>29+</td>
<td><a href="http://www.facebook.com/dave.wichers?sk=notes">http://www.facebook.com/dave.wichers?sk=notes</a></td>
<td>HTTP headers</td>
<td>Hide/Star this record</td>
</tr>
<tr>
<td>92+</td>
<td><a href="http://www.facebook.com/dave.wichers?sk=notes">http://www.facebook.com/dave.wichers?sk=notes</a></td>
<td>HTTP headers</td>
<td>Hide/Star this record</td>
</tr>
<tr>
<td>122+</td>
<td><a href="http://www.facebook.com/dave.wichers?sk=notes">http://www.facebook.com/dave.wichers?sk=notes</a></td>
<td>HTTP headers</td>
<td>Hide/Star this record</td>
</tr>
<tr>
<td>134+</td>
<td><a href="http://www.facebook.com/dave.wichers?sk=notes">http://www.facebook.com/dave.wichers?sk=notes</a></td>
<td>Untrusted code</td>
<td>Hide/Star this record</td>
</tr>
</tbody>
</table>

**Security notes:**

Loading of scripts from an untrusted origin.

**Global ID:**

http://www.facebook.com/dave.wichers#SCRIPT

**Document URL:**

http://www.facebook.com/dave.wichers?sk=notes

**Data used:**

**URL:**

http://static.wk.fbcdn.net/rsrc.php/v1/yP/s/jwR4w5nQCdL.js

**HTML:**

```html
<script src="http://static.wk.fbcdn.net/rsrc.php/v1/yP/s/jwR4w5nQCdL.js" type="text/javascript" async=""></script>
```
Free - Open Source Detection Tools cont’d

• Nishant Das Patnaik/Security Engineer & Sarathi Sabyasachi Sahoo/Software Engineer, Yahoo, India
• FireFox added on, first discussed Feb, 2012
  • Downloads added to Google project Apr 5, 2012
• Large database of DOM-Based XSS injection vectors.
• Fuzzes sources with these attacks and flags sinks where the attacks actually execute.
• Intended to be mostly point and click
• http://code.google.com/p/ra2-dom-xss-scanner/
DOM XSS Scanner – from Ramiro Gómez

- Online service
- Just type in your URL and hit go
- Simplistic string matching source and sink detector
- Purely a human aide

http://www.domxssscannerc.com/

Results from scanning URL: http://js.revsci.net/gateway/gw.js?csid=A09
Number of sources found: 7
Number of sinks found: 0

```javascript
{this._rsima=ra;}this.DM_tag=function(){var Ra;if(this._rsioa==0)if(typeof(DM_prepClient)=="function")DM_prepClient(this._rsiaa,th;
Sa=this._rsiya();if(this._rsiia="gif"){Ra=new Image(2,3);Ra.src=Sa;this._rsina[this._rsina.length]=Ra;}else if(th
{if(this._rsifa==1)document.write("<script language=""JavaScript"

type="text/javascript" type=""+Sa+""+/
```
IBM’s JavaScript Security Analyzer (JSA)

- Built into AppScan
- Crawls target site
- Copies ALL JavaScript
- Then does source code analysis on it

JavaScript Vulnerability Types

- DOM-based XSS
- Phishing through Open Redirect
- HTML5 Notification API Phishing
- HTML5 Web Storage API Poisoning
- HTML5 Client-side SQL Injection
- HTML5 Client-side Stored XSS
- HTML5 Web Worker Script URL Manipulation
- Email Attribute Spoofing
Web Vulnerability Scanner (WVS)

- has Client Script Analyzer (CSA) for detecting DOM-Based XSS

http://www.acunetix.com/blog/web-security-zone/articles/dom-xss/

Commercial Edition

Any other commercial tools??
Conclusion

- Client/DOM-Based XSS is becoming WAY more prevalent
- It's generally being ignored
- We need to KNOW what JavaScript APIs are safe vs. unsafe
- We need more systematic techniques for finding Client XSS flaws
- We need better guidance on how to avoid / fix such flaws