Unraveling some of the Mysteries around DOM-based XSS

Dave Wichers
Aspect Security, COO
OWASP Boardmember
OWASP Top 10 Project Lead
OWASP ASVS Coauthor

ASPECT SECURITY
Application Security Experts

dave.wichers@aspectsecurity.com

Cross-Site Scripting (XSS)

Types of Cross-Site Scripting Vulnerabilities (per OWASP and WASC)
Type 2: Stored XSS (aka Persistent)
Type 1: Reflected XSS (aka non-Persistent)
Type 0: DOM-Based XSS
Sources: https://www.owasp.org/index.php/Cross-site_Scripting_(XSS) 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

"DOM-based vulnerabilities occur in the content processing stage performed on the client, typically in client-side JavaScript." – http://en.wikipedia.org/wiki.Cross-site_scripting

XSS Categories

Really its more like this

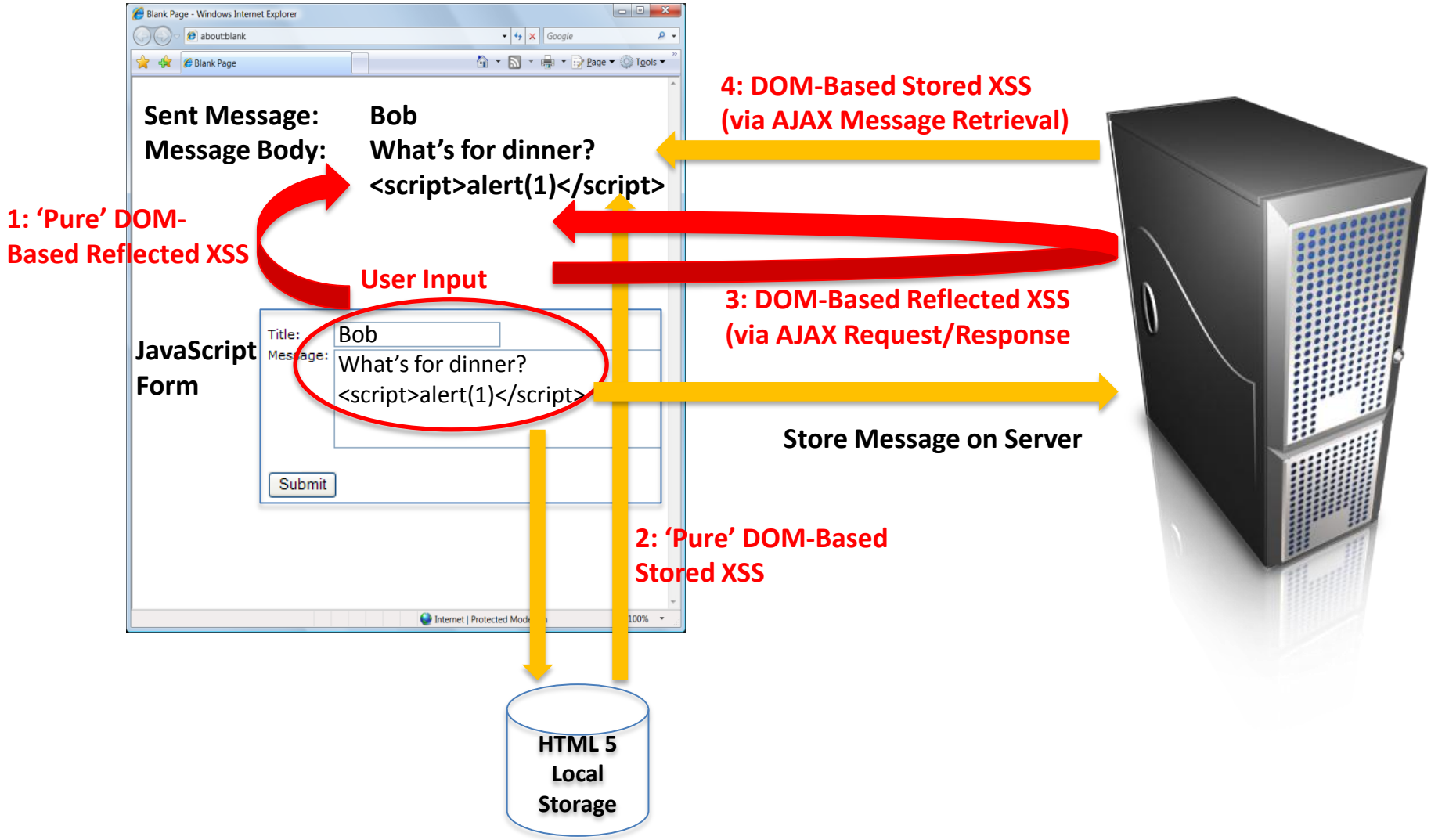
(Traditional) Stored XSS	**DOM-Based Stored XSS**
(Traditional) Reflected XSS	DOM-Based Reflected XSS

XSS Categories – More Details

Or maybe like this

(Traditional) Stored XSS	DOM-Based Stored XSS (data from server)	'Pure' DOM-Based Stored XSS (data from HTML5 Local storage)
(Traditional) Reflected XSS	DOM-Based Reflected XSS (data from server)	'Pure' DOM-Based Reflected XSS (data from DOM)

DOM-Based XSS - Viewed Another Way



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?

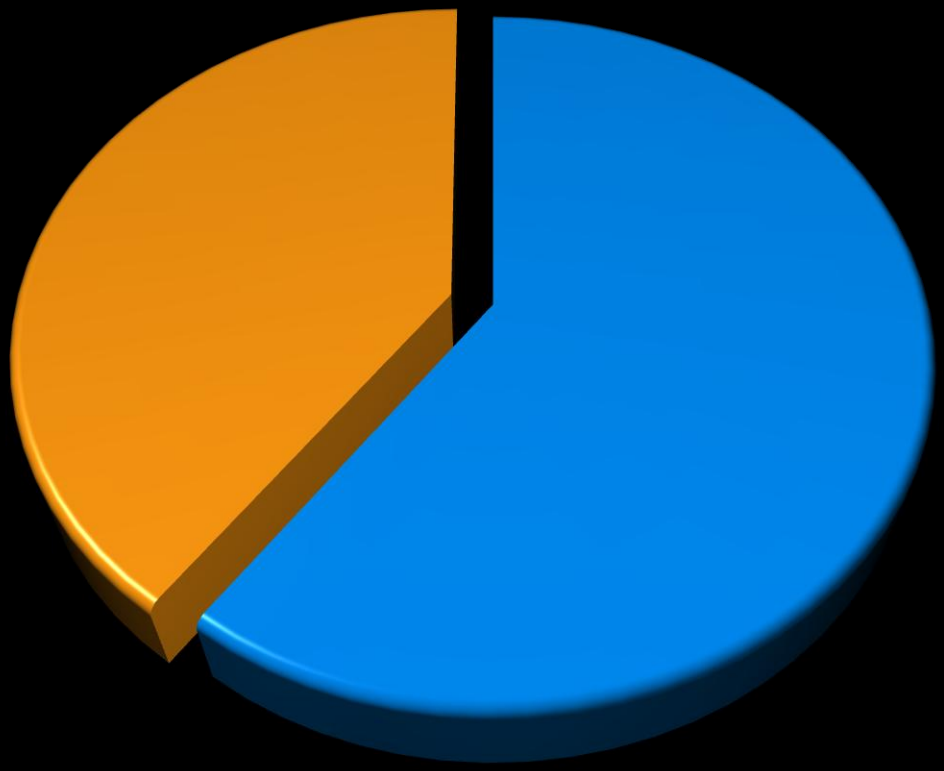
Using IBM's JavaScript Security Analyzer (JSA), IBM tested Fortune 500 + Top 178 Sites and found

40%

Vulnerable to Client-side JavaScript vulnerabilities,

90%

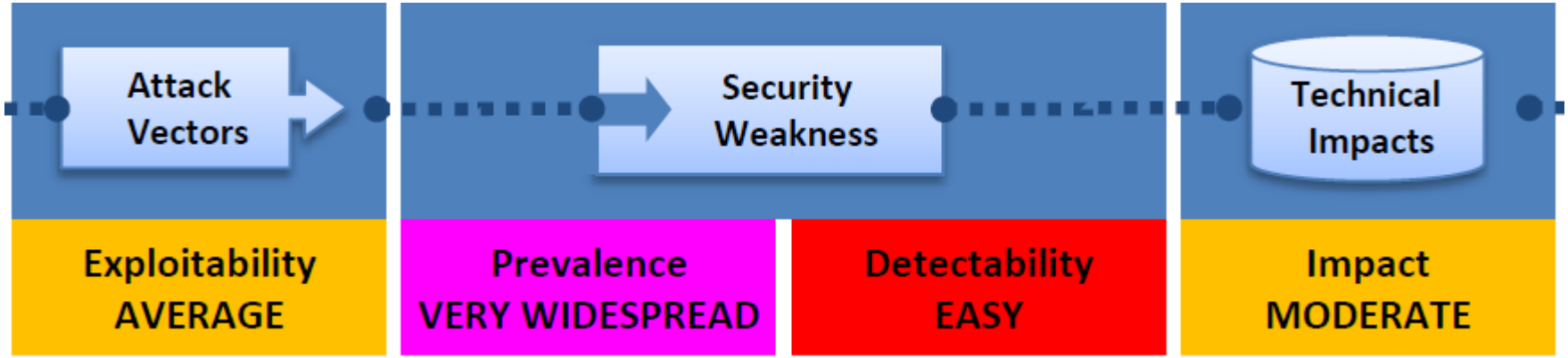
of which was caused by 3rd party JavaScript Libraries



Source: IBM

What's the Risk of 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 Traditional and DOM-Based XSS
  - Detectability is lower for DOM-Based XSS as its harder for attackers (and defenders) to find

DOM-Based XSS – The Classic Example

For: http://www.vulnerable.site/welcome.html#name=Joe<script>alert(1)</script>?name=Joe

```
<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>
```

src: http://projects.webappsec.org/w/page/13246920/Cross Site Scripting

Why is finding DOM-Based XSS So Hard?

- Document Object Model
  - "…convention for representing and interacting with objects in HTML, XHTML and XML documents.[1] Objects in the DOM tree **may be addressed and manipulated by using methods** on the objects." (http://en.wikipedia.org/wiki/Document_Object_Model)
  - Existing JavaScript can update the DOM and new data can also contain JavaScript
- Its 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

How do we make detecting/avoiding DOM-Based XSS more of a Science?

Better Understanding of

- Dangerous Sources
- Propagators (not covered here)
- Unsafe Sinks
- Defense Techniques

Dangerous Sources (of Browser Input)

Blank Page - Windows Internet Explorer

about:blank

Blank Page

#5 JavaScript
CSS

3rd Party Content in Same Origin

3rd Party Server

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

#3 document.*
window.*

Request/Window Attributes

Server

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

Standard HTTP Response

XHR Response

#2

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

Internet | Protected Mode: On

100%

Dangerous Request/Window Attributes

- Page: 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)

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>;test<a"'%0A`= +%20>?test<a"'%0A`= +%20>;#test<a"'%0A`= +%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<a"'%0A`=%20+%20>;#test<a"'%0A`=%20+%20>;

- Similar info for other direct browser data sources including
  - document.cookie (https://code.google.com/p/domxsswiki/wiki/TheCookiesSources)
  - document.referer (https://code.google.com/p/domxsswiki/wiki/TheReferrerSource)
  - window.name (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)

Safe JSON parsing

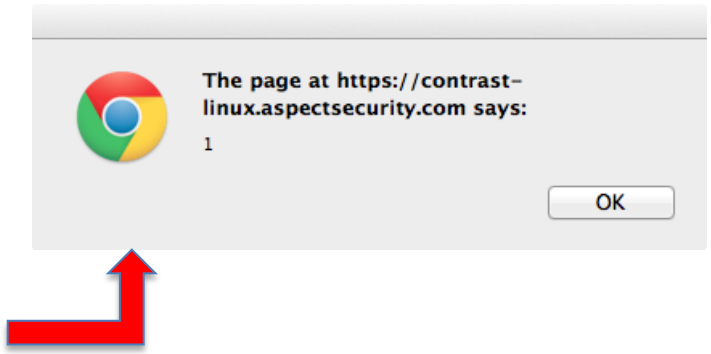
- JSON.parse() (rather than eval())

Popular JavaScript Library #1: jQuery

jQuery Methods That Directly Update the DOM	
.after()	.prependTo()
.append()	.replaceAll()
.appendTo()	.replaceWith()
.before()	.unwrap()
.html()	.wrap()
.insertAfter()	.wrapAll()
.insertBefore()	.wrapInner()
.prepend()	Note: .text() updates DOM, but is safe.
These are all the DOM Insertion and Replacement methods in jQuery.	

Don't send unvalidated data to these methods, or properly escape the data before doing so.

jQuery – But there's more…



- More danger
  - jQuery(danger) or $(danger)
    - This immediately evaluates the input!!
    - E.g., $("<img src=x onerror=alert(1)>")
  - jQuery.globalEval()
  - All event handlers: .bind(events), .bind(type, [,data], handler()), .on(), .add(html),
- 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?

Home / Products

Ext JS 3.4

<swfobject>

XSS Prevention Techniques
Traditional vs. DOM-Based

Traditional XSS	DOM-based XSS
Avoid including unsafe input in response	Same
Server Side Context Sensitive Output Escaping	Client Side Context Sensitive Output Escaping
Server Side Input Validation of the Request (HARD)	Same, plus: Client Side Input Validation of the Response
No equivalent (page is always interpreted)	Avoid JavaScript Interpreter (i.e., avoid unsafe sinks)

Primary XSS Defense: Context Sensitive Escaping

Blank Page - Windows Internet Explorer

about:blank

Blank Page

HTML Element Content
(e.g., some text to display)

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')")

Internet | Protected Mode: On

#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 for more details

Client Side Context Sensitive Output Escaping

Context	Escaping Scheme	Example
HTML Element	(&, <, >, ") → &entity; (', /) → &#xHH;	$ESAPI.encoder(). encodeForHTML()
HTML Attribute	All non-alphanumeric < 256 → &#xHH	$ESAPI.encoder(). encodeForHTMLAttribute()
JavaScript	All non-alphanumeric < 256 → \xHH	$ESAPI.encoder(). encodeForJavaScript()
HTML Style	All non-alphanumeric < 256 → \HH	$ESAPI.encoder(). encodeForCSS()
URI Attribute	All non-alphanumeric < 256 → %HH	$ESAPI.encoder(). encodeForURL()

ESAPI for JavaScript Library Home Page: https://www.owasp.org/index.php/ESAPI_JavaScript_Readme
Identical encoding methods also built into a 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:

```
<script>
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
- Trick is knowing which calls are safe or not
  - We covered examples of safe/unsafe sinks already

DOM-Based XSS While Creating Form

Attack URL Value: http://a.com/foo?" **onblur="alert(123)**

Vulnerable Code:

```
var html = ['<form class="config">',
                '<fieldset>',
                '<label for="appSuite">Enter URL:</label>',
                '<input type="text" name="appSuite" id="appSuite"
                   value="', options.appSuiteUrl || '', '" />',
                '</fieldset>',
            '</form>'].join(''), dlg = $(html).appendTo($('body'));
        ...
```

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

Fix #1:

```
regexp = /http(s):\/\/(\w+:{0,1}\w*@)?(\S+)(:[0-9]+)?(\/|\/([\w#!:.?+=&%@!\-
\/]))?/;
buttons: { 'Set': function () {
           var u = $.trim(appSuite.val());
           if (!regexp.test(u) || u.indexOf('"') >= 0) {
               Util.ErrorDlg.show('Please enter a valid URL.');
               return;
           } ...
```

Fix #2 – Safe construction of the form

Vulnerable Code:

```
var html = ['<form class="config">',
                '<fieldset>',
                '<label for="appSuite">Enter URL:</label>',
                '<input type="text" name="appSuite" id="appSuite"
                   value="', options.appSuiteUrl || '', '" />',
                '</fieldset>',
            '</form>'].join(''), dlg = $(html).appendTo($('body'));
        ...
```

Fix #2:

```
var html = ['<form class="config">',
                '<fieldset>',
                '<label for="appSuite"> Enter URL:</label>',
                '<input type="text" name="appSuite" id="appSuite" />',
                '</fieldset>',
            '</form>'].join(''), dlg = $(html).appendTo($('body'));
             appSuite.val(options.appSuiteUrl || '');
        ...
```

Techniques for Finding DOM-Based XSS #1

Test like normal XSS in obvious inputs

- Step 1: Enter test script: dave<script>alert(1)</script>
- Step 2: Inspect response and DOM for 'dave'
- Step 3: if found, determine if encoding is done (or not needed)
- Step 4: adjust test to actually work if necessary
  - E.g., dave" /><script>alert(1)</script>
  - dave" onblur="(alert(2))

Tools: Normal manual Pen Test Tools like WebScarab/ZAP/Burp can be used here
Automated scanners can help, but many have no DOM-Based XSS specific test features
More tips at: https://www.owasp.org/index.php/Testing_for_DOM-based_Cross_site_scripting_(OWASP-DV-003)

Techniques for Finding DOM-Based XSS #2

Inspect JavaScript loaded into DOM

- Step 1: look for references to user controlled input
  - Remember 5 browser sources referenced previously?
- Step 2: follow data flow to dangerous sinks
  - Lots of dangerous sinks mentioned before
- Step 3: if data is properly validated or encoded before reaching dangerous sink (its 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

Unexploitable XSS ?? Not Always …

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 DOM-Based, but data passes through server:
    - Force the request to the server, instead of filling out the form. Works for per user Stored XSS, but not Reflected XSS, since XHR won't be waiting for response.

Its not just 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 … ☹

Free - Open Source Detection Tools

DOMinator – by Stefano DiPaola

- Firefox Plugin (to OLD version of FF)
- Works by adding taint propagation to strings within the browser
- Difficult to install, run, and understand output, <u>but very promising approach</u>
- Update coming out soon (April, 2012)
  - Updated to Firefox 8.0.1
  - Adds support for some HTML5 features like cross domain requests, new tags, etc.
- http://code.google.com/p/dominator/

Free - Open Source Detection Tools cont'd

- 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/

Id	URL	Type	
23+	http://www.facebook.com/dave.wichers?ref=tn_tnmn	HTTP headers	Hide Star this record
29+	http://www.facebook.com/dave.wichers?sk=notes	HTTP headers	Hide Star this record
92+	http://www.facebook.com/dave.wichers?sk=notes	HTTP headers	Hide Star this record
122+	http://www.facebook.com/dave.wichers?sk=notes	Untrusted code	Hide Star this record
134+	http://www.facebook.com/dave.wichers?sk=notes	Untrusted code	Hide Star this record

Show similar records | Export record

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.ak.fbcdn.net/rsrc.php/v1/yP/r/ezRr4usBCUr.js
```

HTML:

```
<script src="http://static.ak.fbcdn.net/rsrc.php/v1/yP/r/ezRr4usBCUr.js" type="text/javascript" async=""></script>
```

Free - Open Source Detection Tools cont'd

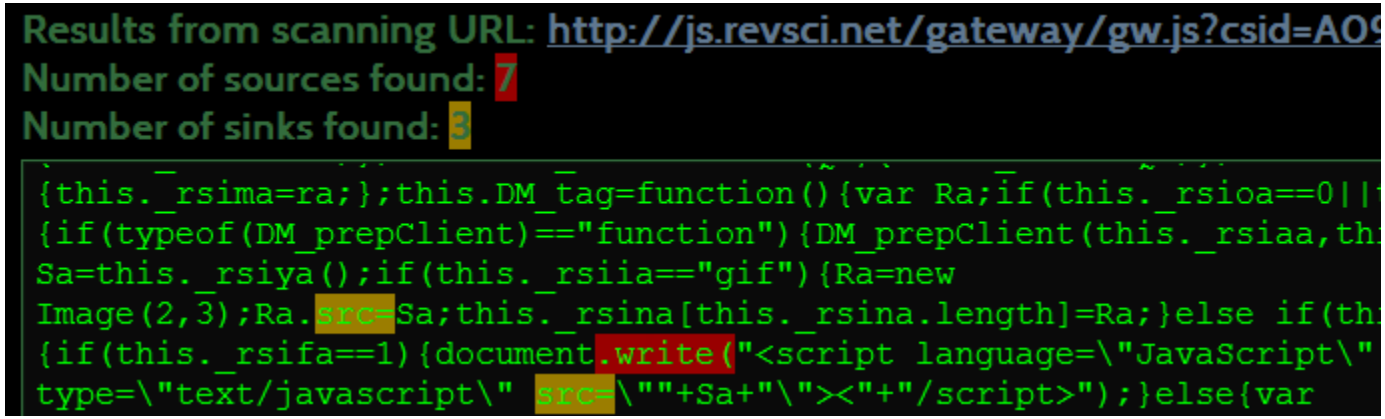
Ra.2 ra2-dom-xss-scanner

Ra.2 - Blackbox DOM XSS Scanner

- Nishant Das Patnaik/Security Engineer & Sarathi Sabyasachi Sahoo/Software Engineer, Yahoo, India
- FireFox added on, first discussed Feb, 2012
  - Downloads added to Google project 6 hours ago ☺
- 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/

Free - Open Source Detection Tools cont'd

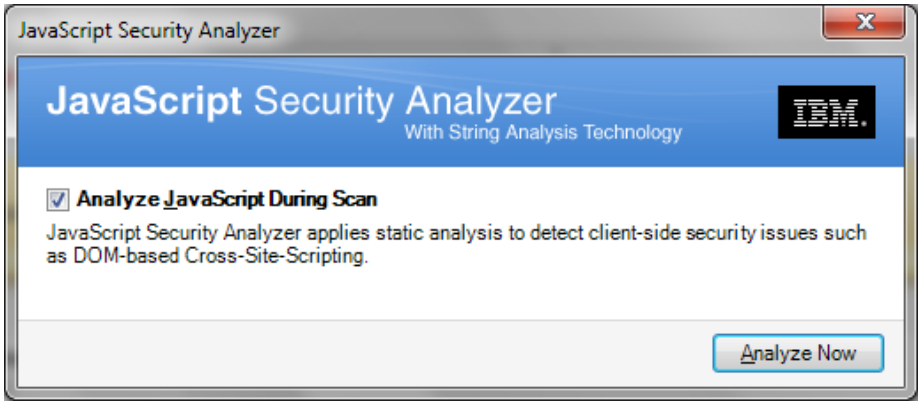
- DOM XSS Scanner – from Ramiro Gómez
  - Online service
  - Just type in your URL and hit go
  - Simplistic string matching source and sink detector
  - Purely a human aide

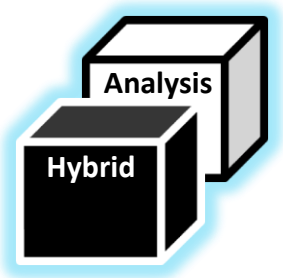


Commercial Tools

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

Commercial Tools cont'd

- acunetix 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/
- DOMinater Commercial Edition (future)
- Any other commercial tools??

Conclusion

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