Ajax

- Fancier and easier-to-use web applications using:
  - Asynchronous
  - JavaScript
  - And
  - XML
- Matter of degree, not kind
Success is foreseeing failure

– Henry Petroski
Cross-Site Scripting

```c:if
test="${param.sayHello}">
Hello ${param.name}!
```

“We never intended the code that's in there to actually be production-ready code.”

- Ryan Asleson
<table>
<thead>
<tr>
<th><strong>Reliving Past Mistakes</strong></th>
</tr>
</thead>
</table>

- Cross-site scripting looks more and more like buffer overflow

<table>
<thead>
<tr>
<th><strong>Buffer Overflow</strong></th>
<th><strong>Cross-site Scripting</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Allows arbitrary code execution</td>
<td>Allows arbitrary code execution</td>
</tr>
<tr>
<td>Easy mistake to make in C/C++</td>
<td>Easy mistake to make</td>
</tr>
<tr>
<td><strong>Exploit is hard to write</strong></td>
<td><strong>Exploit is easy to write</strong></td>
</tr>
<tr>
<td>Well known problem for decades</td>
<td>Well known problem for a decade</td>
</tr>
</tbody>
</table>
What’s Wrong with Ajax?

- Today’s rage or tomorrow’s security disaster?
- Could more JavaScript possibly be better?
- Sample of the almost 400 JavaScript CVE entries:

**CVE-2007-1794**: The Javascript engine in Mozilla 1.7 and earlier... can allow remote attackers to execute arbitrary code.

**CVE-1999-0793**: Internet Explorer allows remote attackers to read files by redirecting data to a Javascript applet.

**CVE-1999-0790**: A remote attacker can read from a Netscape user's cache via JS

**CVE-1999-0347**: Internet Explorer 4.01 allows remote attackers to read local files and spoof web pages via a "%01" character in an "about:" Javascript URL, which causes Internet Explorer to use the domain specified
Overview

- Introduction
- Ajax for...
  - Developers
  - Hackers
- Risks
  - Old
  - New
- State of the frameworks
- Automated protections
- The future of Ajax
AJAX FOR DEVELOPERS
Increased complexity makes frameworks attractive.

Popular toolkits for Ajax development include:
- Google Web Toolkit (GWT)
- Direct Web Remoting (DWR)
- Microsoft ASP.NET AJAX (Codename Atlas)
- Client-side libraries (e.g. Prototype and Dojo)

Also popular: ad-hoc Ajax (roll-your-own)
Back-of-the-Napkin Analysis

- Ad-hoc
- Atlas
- DWR
- GWT
**Ajax - The Case of the Vanishing “X”**

- XML being replaced by more JavaScript/JSON

```xml
<book>
    <title>JavaScript, the Definitive Guide</title>
    <publisher>O'Reilly</publisher>
    <author>David Flanagan</author>
    <cover src="/images/cover_defguide.jpg" />
    <blurb>elit.</blurb>
</book>
```

```json
{  "book": {  
      "title":"JavaScript, the Definitive Guide",
      "publisher":"O'Reilly",
      "author":"David Flanagan",
      "cover": "/images/cover_defguide.jpg",
      "blurb": "elit."
  }
}
```
AJAX FOR HACKERS
Ajax For Malware

- Exploit writers buy JavaScript books too
- Web 2.0 exploits for Web 1.0 vulnerabilities
  - MySpace worm
  - Port scanning behind your firewall
  - Jikto
MySpace does bad input validation
  - Users to post a subset of HTML on their pages
  - No `<script>` tags, no use of the word “javascript”, etc
User “Samy” discovers some holes
- Some browsers allow JavaScript in style attributes
- Some browsers interpret “java\nscript” as “javascript”
- Circumvents MySpace's efforts to prevent JavaScript
Samy adds JavaScript (Ajax) to his page

- Visitors to his page automatically add Samy as a friend and inserts “Samy is my hero” into their profile
- Visitors to a page where Samy is a friend take the same actions

MySpace goes down
Port Scanning Behind Firewalls - 1/2

1) “show me dancing pigs!”
2) “check this out”
3) port scan results

Server
Firewall
Malicious Web page
Browser

scan
scan
scan
Port Scanning Behind Firewalls - 2/2

- Request images from internal IPs
  - `<img src="192.168.0.4:8080"/>`
  - Use timeout/onerror to determine if hosts respond
- `<iframe/>` with timer/onload to map web servers
- Fingerprint webapps using known image names
JavaScript vulnerability scanner
(Billy Hoffman with credit to pdp for crawler)
Spreads like worm over XSS vulnerabilities
Uses Google as proxy to bypass same origin policy
  - Same Origin Policy: basis for browser security
  - JavaScript can't see content from other domains
  - Protects sites from each other
Jikto - 2/2

Victim

"Infected" page

Target Site

Google Translate

Malicious Site

attack

vulnerability scanner
Moral to the Story

- No new vulnerabilities here, just better exploits
- Good offense makes good defense more important
- Good offense is making fast progress
OLD RISKS RECONSIDERED
Defending Ajax: Old Risks Reconsidered

New name, same game

- Old vulnerabilities, new programming language
- Input validation
- Exposing the server
Old Vulnerabilities, New Language

Cross-site scripting in pure JavaScript:

```javascript
q = location.search.split("q=")[1];
q = unescape(q);
div.innerHTML = "searching for " + q;
```
Old Risks: Input validation - 1/3

Easy to lose track of where validation is performed
Old Risks: Input validation - 2/3

- More entry points on the server
  - More, smaller, requests
  - Decentralized design
  - Easy to over-expose
More subtle entry points on the server

- Looks like Web services
- Hard to tell if method call initiated locally (safe) or remotely (dangerous)
- Harder to tell what can be trusted
Example: DWR

```xml
<dwr>
  <allow>
    ...
    <create creator="new"
      javascript="ApartmentDAO"
      class="dwr.sample.ApartmentDAO">
      <exclude method="countApartments"/>
    </create>
  </allow>
</dwr>
```
NEW PROBLEMS
Dirty-data shooters rely on Web 1.0 conventions
- HTTP
- HTML forms
  - 1 parameter = 1 application variable
Ajax = more complex data structures
Ajax requires sophisticated browser emulation
How do you spider an Ajax application?
Looks much more like testing conventional software
Cross-Site Request Forgery (CSRF)

- Cross-Site Request Forgery
  - JavaScript submits HTTP requests on victim's behalf
  - Allows attacker to submit commands, but not inspect the response (Same Origin Policy)

- Application is vulnerable if it:
  - Relies on user’s identity
    (e.g. persistent or session cookies)
  - Does not have secondary authentication mechanism

- Attack against data integrity
New: JavaScript Hijacking - 1/2

- Builds on CSRF
- Breaks confidentiality through loophole in SOP
- Vulnerable if:
  - Site responds to HTTP GET
  - Transmits sensitive data in JavaScript syntax
New: JavaScript Hijacking - 2/2

Malicious Server

1) “show me dancing pigs!”

2) “check this out”

3) confidential data

Mal page
{ witness code }
<script src="...">

Browser

GET

Ajax Application

JavaScript

FORTIFY®
Defenses Against JavaScript Hijacking

- Prevent CSRF
  - Decline malicious requests by requiring unique token
  - ... and remember
  - Default to POST not enough
    (Developers add GET so that result can be cached)
  - Check for a known HTTP header not enough
    (Flash CSRF vulnerability)

- Prevent execution of JavaScript
  - while(1);, /* ... */; etc
  - ... and remember
  - calling parseJSON() rather than eval() does not help
STATE OF THE FRAMEWORKS
## How 12 Popular Frameworks Stack Up

<table>
<thead>
<tr>
<th>Framework</th>
<th>Summary</th>
<th>Prevents JavaScript Hijacking?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prototype</td>
<td>Supports JSON. Defaults to POST when no method is specified, but is easily customizable for using either POST or GET.</td>
<td>No</td>
</tr>
<tr>
<td>Script.aculo.us</td>
<td>Supports JSON. Provides additional UI controls and uses the <strong>Prototype</strong> library for generating requests.</td>
<td>No</td>
</tr>
<tr>
<td>Dojo</td>
<td>Supports JSON. Defaults to POST, but does not explicitly prevent JavaScript Hijacking.</td>
<td>No</td>
</tr>
<tr>
<td>DWR 1.1.4</td>
<td>Uses an expanded version of JSON. Does not implement any JavaScript Hijacking prevention mechanisms.</td>
<td>No</td>
</tr>
<tr>
<td>Moo.fx</td>
<td>Supports JSON. Defaults to POST, but can easily be configured to use GET.</td>
<td>No</td>
</tr>
<tr>
<td>jQuery</td>
<td>Supports JSON. Defaults to GET.</td>
<td>No</td>
</tr>
<tr>
<td>Yahoo! UI</td>
<td>Supports JSON. Responds to GET requests.</td>
<td>No</td>
</tr>
<tr>
<td>Rico</td>
<td>Does not currently support JSON, but will in the future. Supports XML as a data transfer format. Defaults to GET.</td>
<td>N/A</td>
</tr>
<tr>
<td>Microsoft Atlas</td>
<td>Supports JSON. Uses POST by default, but allows programmers to easily change POST to GET and encourages doing so for performance and caching.</td>
<td>No</td>
</tr>
<tr>
<td>MochiKit</td>
<td>Supports JSON. Defaults to GET.</td>
<td>No</td>
</tr>
<tr>
<td>xajax</td>
<td>Does not currently support JSON. Supports XML as a data transfer format.</td>
<td>N/A</td>
</tr>
<tr>
<td>GWT</td>
<td>Supports JSON. Uses POST by default; however, documentation describes how to make GET requests instead and does not mention any security ramifications.</td>
<td>No</td>
</tr>
</tbody>
</table>
We will fix it

“...Thanks for the heads up on this...”

“...Thanks for the paper. We are looking at the issue, and we’re starting to formulate some solutions that mesh well with what you’re suggesting...”
This is not a client-side framework problem
  
  “... Entirely dependent on the server to do the right thing”
  
  “Why the hell should there be security documentation in client frameworks?”
  
  “I added comment stripping support so that people would shut up, not because it’s useful in theory or practice...”
You are recommending bad practices

“But by recommending bad practices, and by failing to strongly recommend good practices, you are making things worse...”
Proof is in the Pudding

3 releases that contain a fix:
- Dojo 0.4.3
  - Implemented /* ... */ , but...
  - Only for JSON, not JavaScript
- DWR 2.0
  - Implemented double-cookie submission
  - Accepted our /* ... */ patch
- Prototype 1.5.1
  - Implemented /* ... */

3 frameworks planned a fix:
- GWT
- MochiKit
- Script.aculo.us

<table>
<thead>
<tr>
<th>Framework</th>
<th>Fixed in Next Release</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dojo</td>
<td>Yes</td>
</tr>
<tr>
<td>DWR</td>
<td>Yes</td>
</tr>
<tr>
<td>GWT</td>
<td>Yes</td>
</tr>
<tr>
<td>jQuery</td>
<td>No Response</td>
</tr>
<tr>
<td>Microsoft Atlas</td>
<td>No</td>
</tr>
<tr>
<td>MochiKit</td>
<td>Yes</td>
</tr>
<tr>
<td>Moo.fx</td>
<td>No Response</td>
</tr>
<tr>
<td>Prototype</td>
<td>Yes</td>
</tr>
<tr>
<td>Rico</td>
<td>N/A</td>
</tr>
<tr>
<td>Script.aculo.us</td>
<td>Yes</td>
</tr>
<tr>
<td>xajax</td>
<td>N/A</td>
</tr>
<tr>
<td>Yahoo! UI</td>
<td>No</td>
</tr>
</tbody>
</table>
Also Good

- 4 mentioned in documentation:
  - dojotoolkit.org/2007/04/02/note-javascript-hijacking
  - getahead.org/dwr/security/script-tag-protection
  - www.prototypejs.org/learn/json
  - developer.yahoo.com/security/
AUTOMATED PROTECTIONS
Static Analysis (Fortify SCA)

- **Challenge:**
  - Lacking standard way to build dynamic JavaScript

- Identify the use of vulnerable frameworks
  - API calls
  - Configuration files
  - Source files

- Identify construction of JavaScript responses
  - `response.setContentType("text/javascript")`
Real-Time Analysis (Fortify RTA)

- Guard to prevent CSRF
- Insert CSRF token on the page using JavaScript
- Verify the unique token is present in requests
A system administration problem

Not unique – just a way to parse results of CSRF

Not interesting – just “View Source”

JavaScript Hijacking can be prevented by:
  - Defaulting to HTTP POST
  - Using parseJSON() instead of eval()
  - Using object notation {} rather than array notation []
  - Checking for “application/json” in content-type
Frameworks offer a chance to build security in
  - Prevent CSRF
  - Provide secure defaults

Opportunity for better validation
  - Better separation between display and controller
  - Better definition for browser/server interaction
Better Web standards
- Cookies: broken, need *working* HTTP only cookies
- Browsers: broken, need to distinguish between scripts from different domains

Flash?
- Documented security best-practices
- Hard to mistake data and code
- Well-defined system for cross-domain communication
Mashups Aren’t Secure

- Scripts from multiple domains == security problem
- Fine for maps, not okay for confidential data
**Summary**

- Ajax: a matter of degree
- Attack techniques improving quickly
- Technology pushed well past original design goals
- Old best practices still apply
  - Don’t trust the client
  - Protect the server
- New challenges
  - Harder to test
  - More complexity == more room for error
- A new ally
  - Build security into Ajax frameworks