Language-based Defenses against Untrusted Browser Origins

K Bhargavan
INRIA Paris

A Delignat-Lavaud
INRIA Paris

S Maffeis
Imperial College London
Towards Defensive Web Components

• How do we write security-sensitive JavaScript components that may be safely executed within partially-trusted websites?

Threats:
• Malicious host server
• Buggy or malicious scripts
• XSS attacks

Component Goals:
• Its functionality cannot be tampered with
• Its secrets cannot be stolen
Example: Single Sign-On

- Provides access to user’s identity and social data
- Runs 3-party authentication and authorization protocol
- Holds secret access token
- How to prevent access token leaks?
  - to unauthorized hosts
  - by malicious, buggy scripts on honest hosts
Example: Client-side Encryption

- Storage and retrieval of encrypted data using a client-side crypto library
  - Cloud storage services
  - Password managers
- Long-term encryption keys never leave the client
- How to protect against encryption key leaks?
  - by other scripts on page
Survey of Web Security Components

• We studied and analyzed mechanisms used by popular web security components
  – Single sign-on, Password managers, Encrypted cloud storage services, Privacy-sensitive web applications

• Variety of deployment techniques with different levels of code integrity and isolation
  – <script>
  – Dynamically load script and eval
  – <iframe>
  – Java applet
  – bookmarklet
  – browser extension
Attacks on Surveyed Components

- Unauthorized websites can fool components into releasing secrets meant for honest websites
- Attackers can exploit standard website vulnerabilities on authorized websites to steal component secrets
  - XSS, Open Redirectors, CSRF, ...

<table>
<thead>
<tr>
<th>Product</th>
<th>Category</th>
<th>Protection Mechanism</th>
<th>Attack Vectors Found</th>
<th>Secrets Stolen</th>
</tr>
</thead>
<tbody>
<tr>
<td>Facebook</td>
<td>Single Sign-On Provider</td>
<td>Frames</td>
<td>Origin Spoofing, URL Parsing Confusion</td>
<td>Login Credential, API Access Token</td>
</tr>
<tr>
<td>Helios, Yahoo, Bitly</td>
<td>Single Sign-On Clients</td>
<td>OAuth Login</td>
<td>HTTP Redirector, Hosted Pages, Malicious JavaScript, CSP Reports</td>
<td>Login Credential, API Access Token</td>
</tr>
<tr>
<td>WordPress, Dropbox</td>
<td>Web Browser</td>
<td>Same-Origin Policy</td>
<td></td>
<td>Login Credential, API Access Token</td>
</tr>
<tr>
<td>Firefox</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1Password, RoboForm</td>
<td>Password Manager</td>
<td>Browser Extension</td>
<td>URL Parsing Confusion, Metadata Tampering</td>
<td>Password</td>
</tr>
<tr>
<td>Verisign, SuperGenPass</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SpiderOak</td>
<td>Encrypted Cloud Storage</td>
<td>Server-side Crypto</td>
<td>CSRF</td>
<td>Files, Encryption Key</td>
</tr>
<tr>
<td>Wuala</td>
<td>Encrypted Cloud Storage</td>
<td>Java Applet, Crypto</td>
<td>Client-side Exposure</td>
<td>Files, Encryption Key</td>
</tr>
<tr>
<td>Mega</td>
<td>Encrypted Cloud Storage</td>
<td>JavaScript Crypto</td>
<td>XSS</td>
<td>Encryption Key</td>
</tr>
<tr>
<td>ConfiChair, Helios</td>
<td>Crypto Web Applications</td>
<td>Java Applet, Crypto</td>
<td>XSS</td>
<td>Password, Encryption Key</td>
</tr>
</tbody>
</table>
Towards Robust Component Security

• Component security is *fragile* against same-origin attackers
  – every buggy script presents a potential attack
  – every XSS attack is fatal and leaks all secrets

• Getting component security right against cross-origin attackers is *hard*, even with strong isolation mechanisms
  – flaws in authorization logic
  – incorrect use of crypto
  – incorrect assumptions about the same origin policy

• *Need for a component programming framework that affords stronger isolation guarantees and supports automated formal analysis*
The DJS Architecture

- **DJS**: a small statically-typed subset of JavaScript
  - formal isolation guarantees against malicious context
- **DJCL**: a crypto library in DJS
  - secure communications with other trusted components
  - applications built with DJS, DJCL and browser mechanisms
- **DJS2PV**: a protocol verifier
  - verifies security goals with a symbolic model of browser, crypto

See: http://www.defensivejs.com
DJS DESIGN BY EXAMPLE
Example: Token-based API Access

- **Goal**: A JavaScript program that uses a secret token to restrict access to a REST API
  - (code excerpted from OWASP CSRFGuard 3)
    ```javascript
    <script>
    var token = "XXXYYYYZZZ...";
    var acl = ["https://rest.W.com",...]
    var api = function(url){
      if (acl.indexOf(url) !== -1) {
        return xhr(url+"?token=3+token);
      }
    }
    </script>
    ```
  - even when running with malicious scripts
  - attacker’s goal: bypass acl and/or steal the token

Page can read & write global variables, DOM, localStorage
Example: Using JavaScript Closures

• Local variables in function bodies are not exposed to the JavaScript context

```javascript
<script>
  var api = (function()
  {
    var token = "XXXYYYYZZZ";
    var acl = ["https://rest.W.com", ...]
    var api = function(url)
    {
      if (acl.indexOf(url) !== -1) {
        return xhr(url+"?token=“+token);
      }
      return api;
    }
  })()
</script>
```

Page scripts can read inline and same-origin scripts
Example: Using a Script Server

• Serve script from a separate origin
  – Page cannot read cross-origin scripts (SOP)
  – Server generates, embeds session-specific token

```html
<script src="http://scripts.W.com/api.js">
</script>
```

```javascript
var api = (function(){
    var token = "XXXYYYYZZZ";
    var acl = ["https://rest.W.com",...]
    var api = function(url){
        if (acl.indexOf(url) !== -1) {
            return xhr(url +"?token="+token);
        }
        return api;
    }()
})()
```

Browser’s XHR primitive can be redefined to steal token
Example: Using Crypto

- Instead of token, send a MAC using the token to authenticate the XHR request

```javascript
var api = (function(){
  var token = "XXXYYYYYZZZ";
  var acl = ["https://rest.W.com",...]
  var hmac = function(k,x){...f()...}
  var api = function(url){
    if (acl.indexOf(url) !== -1) {
      return xhr(url + "?token" + hmac(token,url));
    }
    return api;
  }
})();
<script src="http://scripts.W.com/api.js"></script>
```

Array.prototype.indexOf can be redefined to walk the stack to steal "k"

url provided by attacker may be an object triggering an implicit conversion (toString)
Example: Self-contained Code

• No external references
  – include all auxiliary functions within closure
  – don’t trigger implicit conversions, prototype lookups, ...

```
<script src="http://scripts.W.com/api.js">
</script>

var api = (function(){
    var token = "XXXYYYYZZZ";
    var acl = ["https://rest.W.com",...]
    var mem = function(x,acl){...}
    var hmac = function(k,x){...f(....}
    var api_url = function(url){
        if (mem(url,acl)) {
            return (url+"?token="+hmac(token,url)));}
        return (function(url){xhr(api_url(url))});
    })()
```
Example: Writing Defensive JavaScript

- It is possible to carefully write JavaScript code that protects its functionality and secrets from malicious scripts
  - relying on a separate script server,
    a cryptographic library,
    and by writing fully self-contained code
- but it can be painful and error-prone
  - easy to miss JavaScript corner cases
  - need for automated tools and formal guarantees
DJS Type System

- A sound static type system that identifies a formal subset of JavaScript and enforces our defensive idioms
  - fully self-contained, no external references
  - all code wrapped in a closure and exposed through a typed first-order API

Type Safety Guarantees:
- **Independence**: The input-output functionality of well-typed programs is the same in all JavaScript contexts
- **Encapsulation**: The only way a context can discover the content of a typed program is by calling its API
Example: Typing Guarantees

- **Independence**: External scripts cannot bypass the authorization check on `url` and `acl`

- **Encapsulation**: External scripts cannot read token, but can call `api` to learn the HMAC

```javascript
var api = (function(){
    var token = "XXXYYYYZZZ";
    var acl = ["https://rest.W.com",...]
    var mem = function(x,acl){...}
    var hmac = function(k,x){...f()....}
    var api_url = function(url){
        if (mem(url,acl)) {
            return (url+"?token=”+hmac(token,url));}
        return (function(url){return (xhr(api_url(url)))});
    }()
})()
```
Typing Restrictions

• All variables are lexically scoped
  – and statically typed
  – no implicit coercions

• Objects and arrays are defined as literals
  – not extensible
  – no prototype inheritance
  – limited support for dynamic accessors \((x[y])\)

• No eval

• No direct access to DOM or browser libraries
  – Possible to grant limited access via postMessage
Programming in DJS

• Not meant for general web applications but works well for security-critical components
  – Cryptography, Authorization Policies
  – Rest of the page remains in full JavaScript

• Type inference tool
  – Verifies that a JavaScript program is well-typed in DJS

<table>
<thead>
<tr>
<th>Program</th>
<th>LOC</th>
<th>Typing</th>
<th>PV LOC</th>
<th>ProVerif</th>
</tr>
</thead>
<tbody>
<tr>
<td>DJCL</td>
<td>1728</td>
<td>300ms</td>
<td>114</td>
<td>No Goal</td>
</tr>
<tr>
<td>JOSE</td>
<td>160</td>
<td>36ms</td>
<td>9</td>
<td>No Goal</td>
</tr>
<tr>
<td>Sec. AJAX</td>
<td>61</td>
<td>7ms</td>
<td>243</td>
<td>12s</td>
</tr>
<tr>
<td>LastPass</td>
<td>43</td>
<td>42ms</td>
<td>164</td>
<td>21s</td>
</tr>
<tr>
<td>Facebook</td>
<td>135</td>
<td>42ms</td>
<td>356</td>
<td>43s</td>
</tr>
<tr>
<td>ConfiChair</td>
<td>80</td>
<td>31ms</td>
<td>203</td>
<td>25s</td>
</tr>
<tr>
<td>Program</td>
<td>LOC</td>
<td>Typing</td>
<td>PV LOC</td>
<td>ProVerif</td>
</tr>
<tr>
<td>--------------</td>
<td>------</td>
<td>---------</td>
<td>--------</td>
<td>--------------</td>
</tr>
<tr>
<td>DJCL</td>
<td>1728</td>
<td>300ms</td>
<td>114</td>
<td>No Goal</td>
</tr>
<tr>
<td>JOSE</td>
<td>160</td>
<td>36ms</td>
<td>9</td>
<td>No Goal</td>
</tr>
<tr>
<td>Sec. AJAX</td>
<td>61</td>
<td>7ms</td>
<td>243</td>
<td>12s</td>
</tr>
<tr>
<td>LastPass</td>
<td>43</td>
<td>42ms</td>
<td>164</td>
<td>21s</td>
</tr>
<tr>
<td>Facebook</td>
<td>135</td>
<td>42ms</td>
<td>356</td>
<td>43s</td>
</tr>
<tr>
<td>ConfiChair</td>
<td>80</td>
<td>31ms</td>
<td>203</td>
<td>25s</td>
</tr>
</tbody>
</table>
DJCL: Defensive Crypto Library

• A JavaScript crypto library written in DJS
  – SHA-256, HMAC, AES CBC/CCM/GCM, RSA OAEP/PSS
  – BASE64, UTF8, JSON, JOSE

• Typing guarantees:
  – Crypto computations cannot be tampered with
  – Does not leak keys to the environment
    (except possibly through side-channels)

• High performance:
  – As fast as (or faster than) SJCL, JSBN
  – Statically-allocated, self-contained, functional code
    in JavaScript is well suited to optimization (like asm.js)
DJS2PV: Verifying DJS Applications

- **DJS to pi calculus translation**
  - uses static typing
  - DJCL -> symbolic crypto model

- **WebSpi Browser Model**
  - HTTP/HTTPS, XMLHttpRequest
  - Cookies, localStorage
  - JavaScript heap, SOP

- **ProVerif protocol verifier**
  - Dolev-Yao adversary, unbounded sessions
  - Verifies secrecy and authenticity
  - Or finds attacks
Password Manager Bookmarklet

• **LastPass Login Bookmarklet**
  – On click included code runs in the current page
  – Uses an embedded secret to perform authenticated RPC with LastPass server
  – *Attack:* Malicious script on hosting page can steal the bookmarklet secret (and hence LastPass data)
Password Manager Bookmarklet

• Improved version of LastPass Login
  – Uses DJS to isolate bookmarklet code from page
  – Secure AJAX call to LastPass server using DJCL
  – Fits in 2048 bytes (including AES, HMAC)
  – Protocol model extracted and verified with DJS2PV

• Improved Security Guarantees
  – Bookmarklet secret and LastPass passwords not revealed to malicious sites
  – Click Authentication: Form only filled if the user clicks on the bookmarklet, no automatic login
Script-level Access Control for FB

- Facebook API and token accessible to all scripts running on the host origin
  - Vulnerable to a number of web attacks
  - Open Redirectors, XSS, malicious hosted pages
  - Should be accessible only by site scripts
Script-level Access Control for FB

• Modified Facebook API design:
  – Token is never released to the page,
  – Only authorized scripts may call the FB API
  – API calls authenticated using session keys and DJCL

• Results:
  – Modify one method in the FB SDK (0.5% of codebase)
  – Add 20 lines of DJS code + DJCL to authorized scripts
  – Negligible performance impact
  – Protocol model extracted and verified with DJS2PV
XSS-Resistant Client-side Encryption

- ConfiChair website uses client-side encryption
  - Java Crypto applet with JavaScript API
  - Keys stored in local storage
  - XSS attack on any page leaks all keys
XSS-Resistant Client-side Encryption

• Our design:
  – Java applet replaced with DJCL
  – Encryption scripts embedded with session key
  – Keys stored encrypted with session key in local storage
  – No other script obtains the keys

• Result:
  – Modified less than 10 lines of website code
  – Encryption library is typechecked in DJS
  – Full crypto protocol verified with DJS2PV
Summary

• Many recent attacks on JavaScript security components
• DJS: An architecture for programming and analyzing JavaScript security components
• Small code changes yield strong isolation guarantees
  – XSS-resistant security components
  – applicable even to server-side JavaScript (e.g. Node)

• DJS programs are fast or faster than idiomatic JavaScript
  – triggers optimizations similar to asm.js
• Automated formal analysis for web crypto protocols in DJS
  – relying on formal models of crypto and the browser
Questions?

• Try it: http://www.defensivejs.com