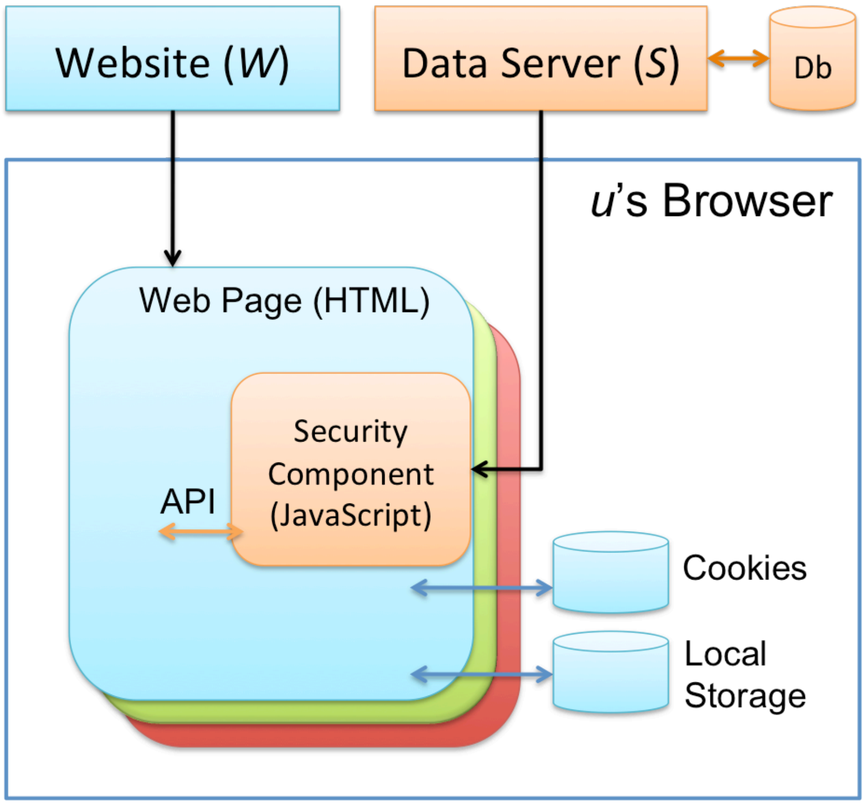
Language-based Defenses against Untrusted Browser Origins

K Bhargavan A Delignat-Lavaud S Maffeis

INRIA Paris INRIA Paris Imperial College London

Towards Defensive Web Components

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



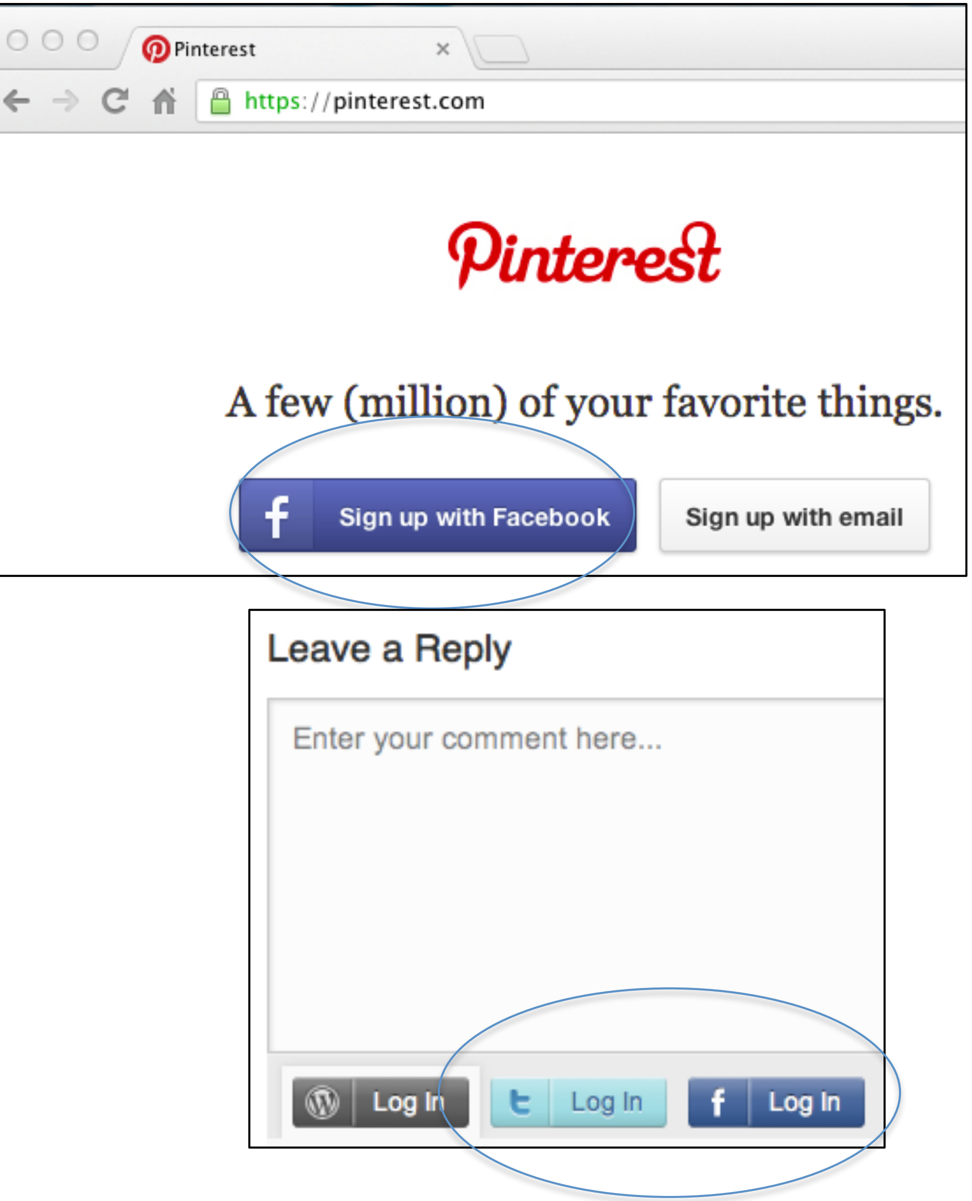
Threats:

- Malicious host server
- Buggy or malicious scripts
- XSS attacks

Component Goals:

- Its functionality cannot be tampered with
- Its secrets cannot be stolen

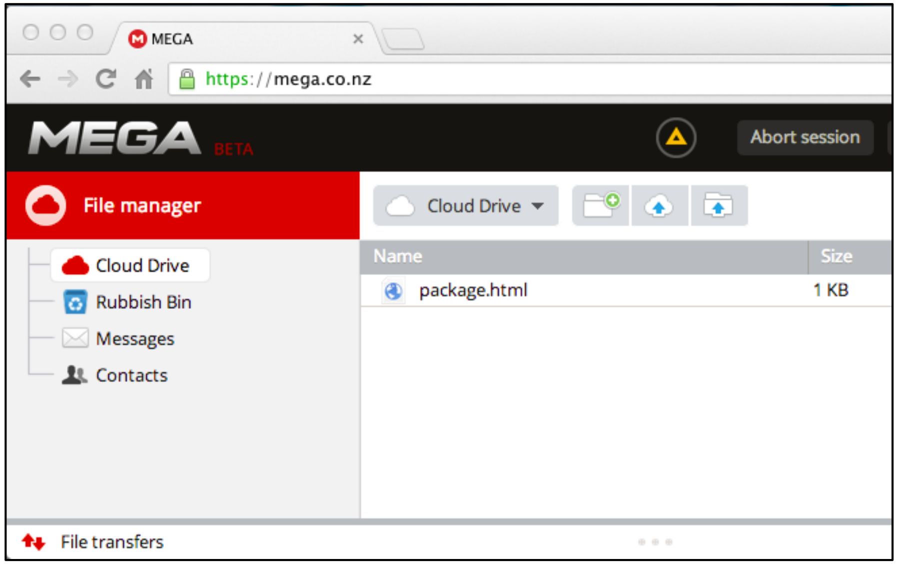
Example: Single Sign-On

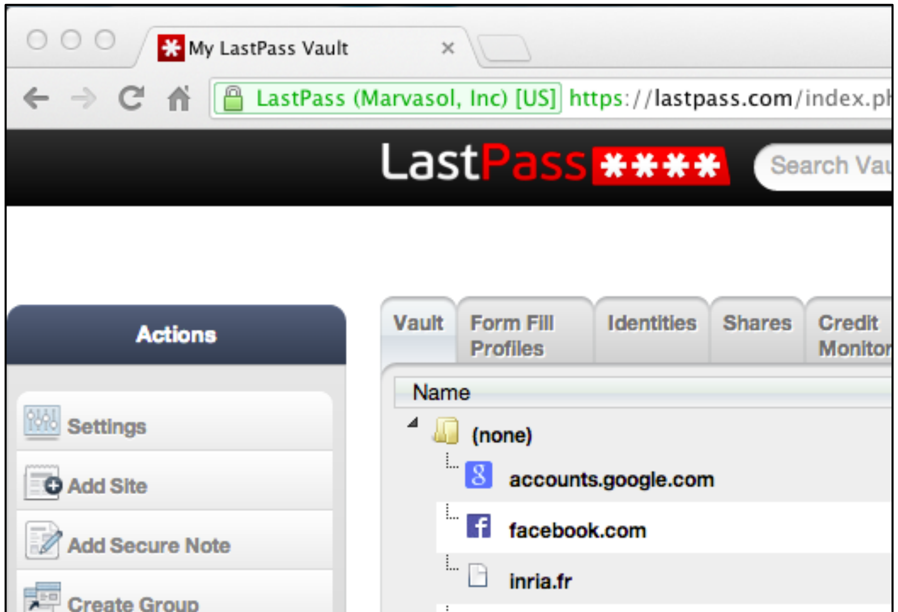


- Provides access to user's identity and social data
- Runs 3-party authentication and authorization protocol
- Holds secret access token

- 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, …

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

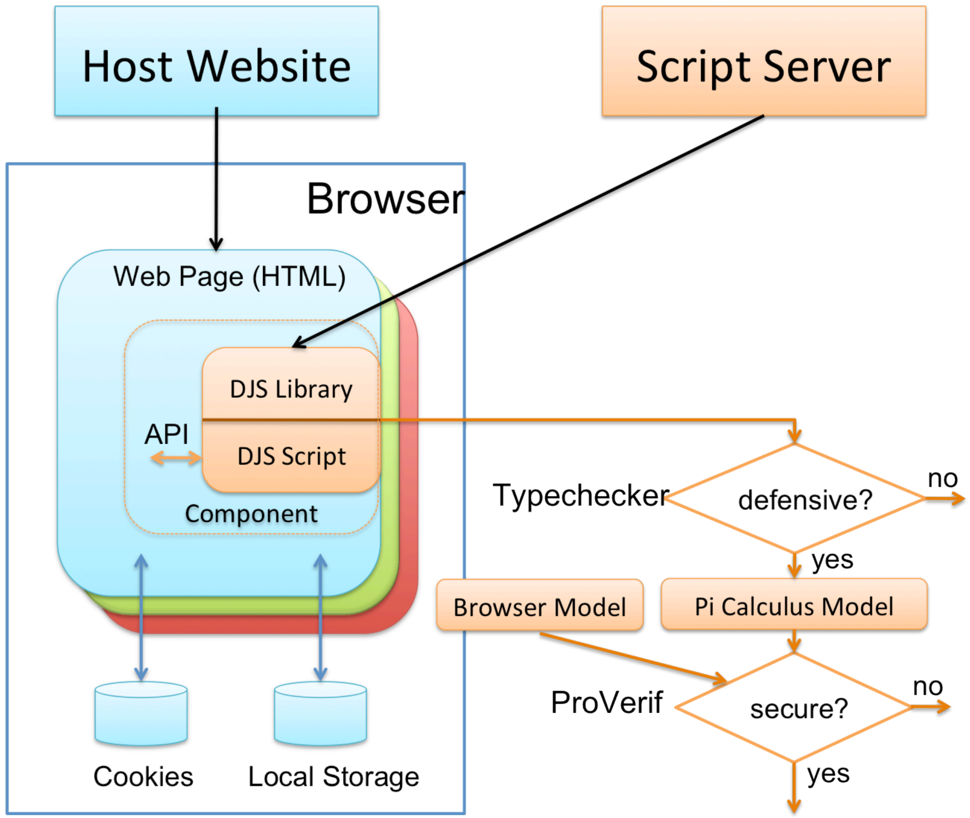
Towards Robust Component Security

- 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)

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

> Page can read & write global variables, DOM, localStorage

  - even when running with malicious scripts
  - attacker’s goal: bypass acl and/or steal the token

Example: Using JavaScript Closures

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

```
<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

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

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

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

```
var api = (function(){
    var token = "XXXYYYZZZ";
    var acl = ["https://rest.W.com",...]
    var hmac = function(k,x){...f()...
    var api = function(url){
      if (acl.indexOf(url)  !== -
            return xhr(url+"
    return api;
  })()
```

Array.prototype. indexOf can be redefined to

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

```
var api = (function(){
   var token = "XXXYYYZZZ";
   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

Program	LOC	Typing	PV LOC	ProVerif
DJCL	1728	300ms	114	No Goal
JOSE	160	36ms	9	No Goal
Sec. AJAX	61	7ms	243	12s
LastPass	43	42ms	164	21s
Facebook	135	42ms	356	43s
ConfiChair	80	31ms	203	25s

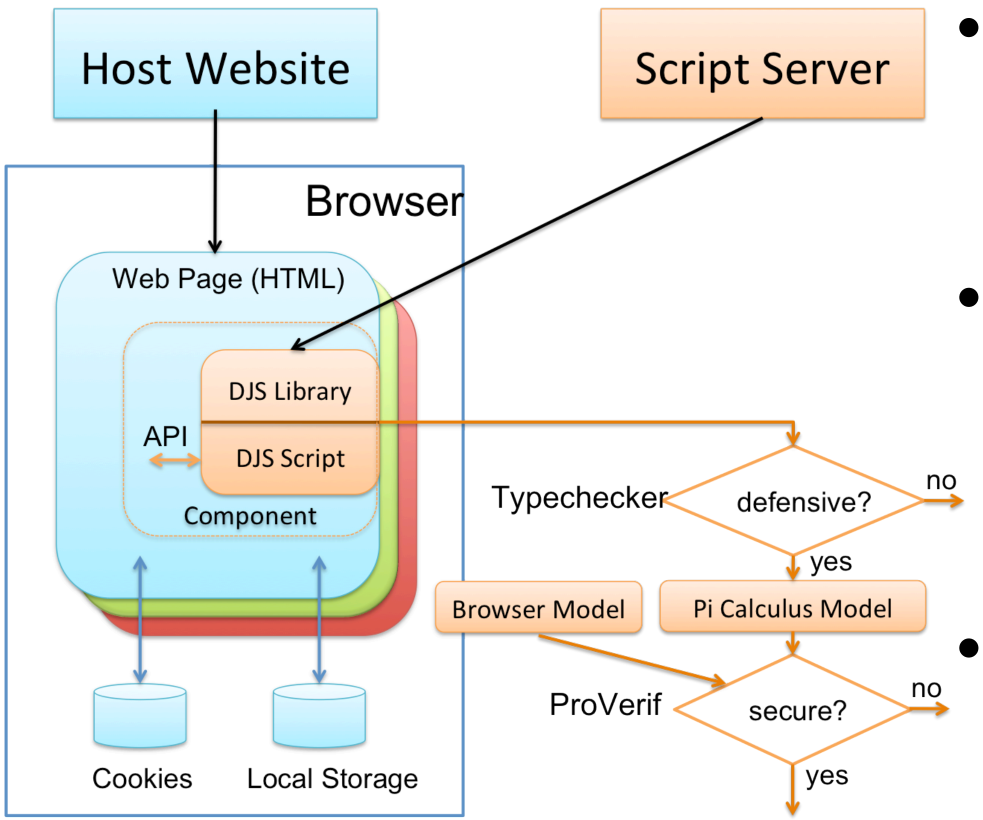
Program	LOC	Typing	PV LOC	ProVerif
DJCL	1728	300ms	114	No Goal
JOSE	160	36ms	9	No Goal
Sec. AJAX	61	7ms	243	12s
LastPass	43	42ms	164	21s
Facebook	135	42ms	356	43s
ConfiChair	80	31ms	203	25s

DJS APPLICATIONS

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)

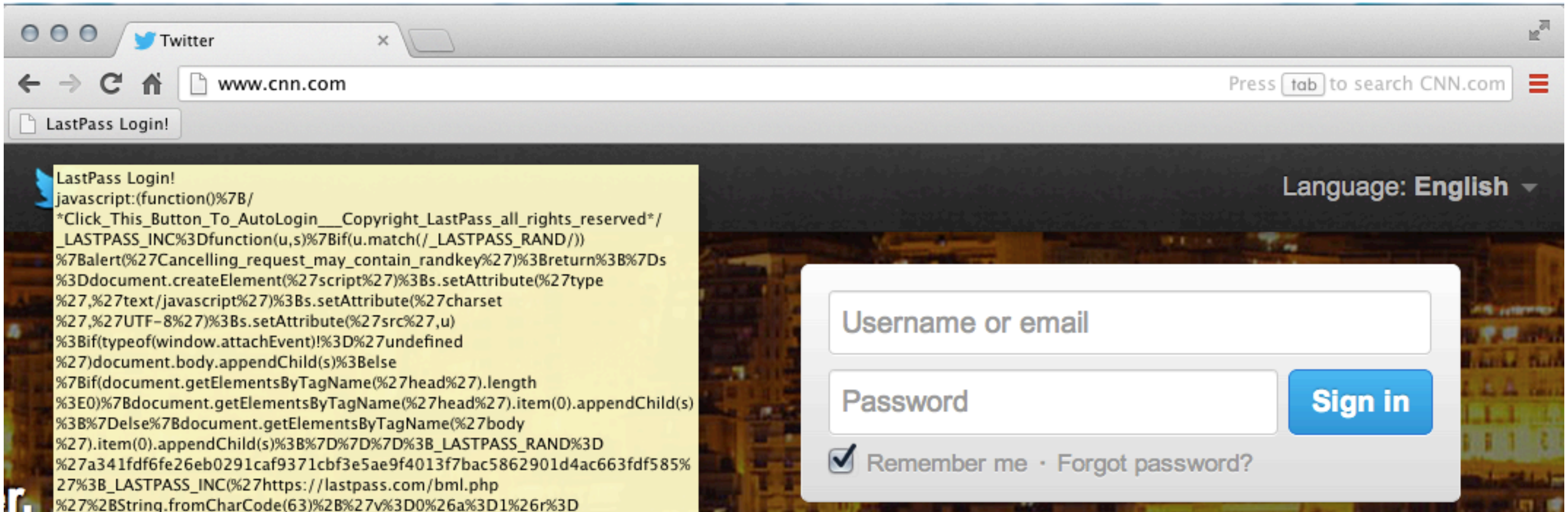
DJS2PV: Verifying DJS Applications



- 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

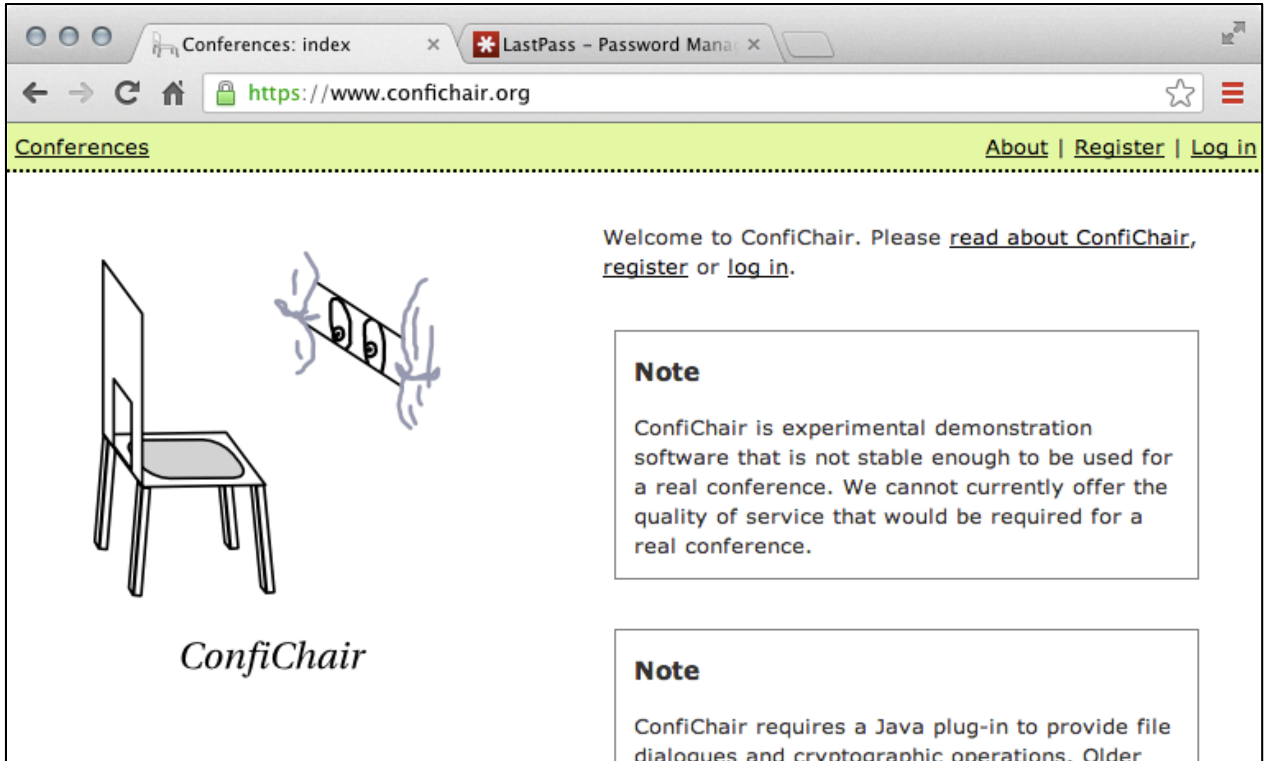
Script-level Access Control for FB

- 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