# Language Based isolation of Untrusted JavaScript

Ankur Taly

Dept. of Computer Science, Stanford University

Joint work with Sergio Maffeis (Imperial College London) and John C. Mitchell (Stanford University)

# Outline

# Motivation

Many contemporary websites incorporate untrusted *JavaScript* content:

- Third party advertisements, Widgets.
- Social Networking sites : User written applications

## Isolation Problem

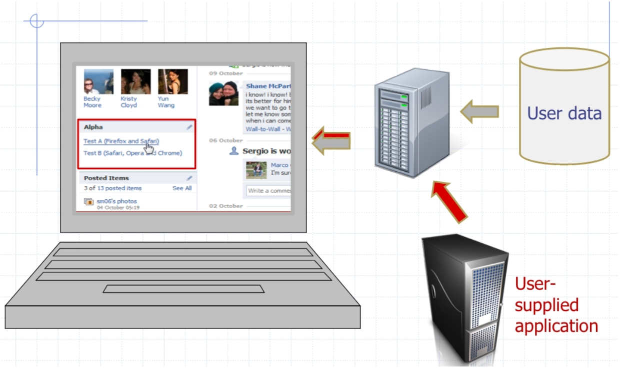
Design security mechanisms which allow untrusted code to perform valuable interactions and at the same time prevent intrusion and malicious damage.

Browser isolation mechanism via Iframes is too restrictive

- Applications need more permissive interaction with the hosting page.
- Hosting page has less control, arbitrary JS can run inside the Iframe !

**This Work**: Focus on untrusted code NOT placed in an Iframe.

# FaceBook setup




- Assume that it is possible for the publisher to preprocess untrusted content before it adds it to the page.

# Program Analysis Problem

## Problem

Given an untrusted JavaScript program $P$ and a description of the hosting page, design a procedure to statically or dynamically via run time checks guarantee that $P$ does not access any security critical portions of the hosting page.

- eval, Function, e1[e2]: Dynamic generation of code, static analysis seems very difficult.

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

**Existing Approach**: Solve the problem for subsets of JavaScript that are more amenable to static analysis.

- FBJS: This talk, ADsafe: See paper.

# A bit more about JavaScript

- First class functions, Prototype based language, re-definable object properties.
- Scope Objects/Stack frames can be first class JavaScript objects: Variable names ⇔ Property names.
- Implicit type conversions which can trigger user code.

```
var y = "a"; var x = {valueOf: function(){ return y;}}
x = x + 10;
js> "a10"
```

# Outline

1. Motivation
2. Case Study : FBJS
   - Design
   - Attacks and Challenges
3. Solving the Isolation problem
   - Formulating the Problem
   - Formal Semantics of JavaScript
   - Isolating library code
   - Isolating other untrusted code
4. Ongoing and Future Work

# Case Study: *FBJS*

*FBJS*: Subset of JavaScript for writing Facebook applications.

**Static Checks**

- Forbid identifiers eval, Function.
- Disallow explicit access to security critical properties (via the dot notation -o.p) __parent__, constructor, . . . .

**Rewriting**

- $o[p] \rightarrow o[idx(p)]$ where $idx(p) = bad$ if $p \in Blacklist$ else $idx(p) = p$.
- $this \rightarrow ref(this)$ where $ref(x) = x$ if $x \neq window$ else $ref(x) = null$
- Add application specific prefix to all top-level identifiers.
  - Example: $o.p \longrightarrow a1234\_o.p$, separates effective namespace of an application from others.

# An attack on FBJS (Nov'08)

## Goal of the Attack

Get a handle to the global object in the application code.

## Almost works

```
var getthis = function(){ return this;};
```

- this gets re-written to ref(this) and the code returns null.
- ref is defined in the global object (base scope object) and application code is disallowed from having handle to global object.
- Can we shadow ref in the current scope object ?

# Attack: Getting the current scope object

## Catch this ! - GET_SCOPE

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

- ECMA-262 semantics for try−catch says that whenever an exception is thrown:
  - New object (say 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).
- In the above code this for f resolves to o.
- Escapes the ref check as $o \neq$ global object.

# Summary of our analysis of FBJS

- We found other attacks on the "ref" and "idx" mechanism (see papers).
- Number of subtleties related to the expressiveness and complexity of JavaScript.
- Finding temporary fixes to the currently known attacks is NOT sufficient. Several million users: Impact value of a single attack is VERY high.

**Formal Analysis**

It is important to do a formal analysis based on traditional programming language foundations to design provable secure isolation techniques

# Outline

# Formulating the Problem

**Problem**

*Ensure that a piece of untrusted code written in a safe subset does not access certain global variables.*

**Enforcement Mechanisms**:

- Filtering (Static Checking): Robust and Efficient, Semantics is unaltered.
- Rewriting (Dynamic Checking) :Run time overhead, Preserving semantics is non-trivial, Proofs are more involved.

Divide this problem into two:

- Isolation from global variables corresponding to library code.
- Isolation from global variables corresponding to other applications.

# Formulating the Problem

**Problem**

*Ensure that a piece of untrusted code written in a safe subset does not access certain global variables.*

**Enforcement Mechanisms**:

- Filtering (Static Checking): Robust and Efficient, Semantics is unaltered.
- Rewriting (Dynamic Checking) :Run time overhead, Preserving semantics is non-trivial, Proofs are more involved.

Divide this problem into two:

- Isolation from global variables corresponding to library code.
- Isolation from global variables corresponding to other applications.

# Isolation from Library Code

**Problem 1**

Design a meaningful sublanguage $Jt$ so that given a program $P \in Jt$, we can statically determine a finite set $Access(P)$ which is the list of all property names that can be potentially accessed.

Isolation from library code

- Create a Blacklist $\mathcal{B}$ of all security critical properties of the global object .
- Filter $P$ if $Access(P) \cap \mathcal{B} \neq \emptyset$.

# Isolation from Library Code

**Problem 1**

Design a meaningful sublanguage $Jt$ so that given a program $P \in Jt$, we can statically determine a finite set $Access(P)$ which is the list of all property names that can be potentially accessed.

Isolation from library code

- Create a Blacklist $\mathcal{B}$ of all security critical properties of the global object .
- Filter $P$ if $Access(P) \cap \mathcal{B} \neq \emptyset$.

# Isolation from other untrusted code

Rename identifiers in order to separate the namespace of untrusted code. Example: x = x + 5 ⟶ a123_x = a123_x + 5.

- Don't rename property names
  - Properties may be inherited from native objects whose are not renamed. o.toString is renamed to a123_o.toString and NOT a123_o.a123_toString
- Restrict access to scope objects
  - Renamed Variables can be differentiated from unrenamed ones by accessing them as property names.
  - var x = 42; this.x returns 42 while var a123_x = 42; this.x returns "Reference error".

**Problem 2**

Define a meaningful sublanguage *Js* so that no program $P \in Js$ can return a pointer to a scope object.

# Isolation from other untrusted code

Rename identifiers in order to separate the namespace of untrusted code. Example: $x = x + 5 \longrightarrow$ a123_x $=$ a123_x $+ 5$.

- Don't rename property names
  - Properties may be inherited from native objects whose are not renamed. o.toString is renamed to a123_o.toString and NOT a123_o.a123_toString
- Restrict access to scope objects
  - Renamed Variables can be differentiated from unrenamed ones by accessing them as property names.
  - var x = 42; this.x returns 42 while var a123_x = 42; this.x returns "Reference error".

**Problem 2**

Define a meaningful sublanguage *Js* so that no program $P \in Js$ can return a pointer to a scope object.

# Formal Semantics of JavaScript

Formalized all of ECMA-262-3$^{rd}$ edition.

- Small step style operational semantics.
- Very long (70 pages of ascii).
- This is already quite non-trivial (none of our FBJS attacks involved the DOM).
- DOM is just treated as a library object in our world.

# A glimpse of the rules

## State

Program state is represented as a triple $\langle H, l, t \rangle$.

- $H$: Denotes the Heap, mapping from the set of locations($\mathbb{L}$) to objects.
- $l$: Location of the current scope object (or current activation record).
- $t$: Term being evaluated.

- Three semantic functions $\xrightarrow{e}$, $\xrightarrow{s}$, $\xrightarrow{P}$ for expressions, statements and programs.
- Atomic transitions: $\mathrm{H,l,t} \longrightarrow \mathrm{H',l',t'}$
- Contextual rules: $\dfrac{\mathrm{H,l,t} \xrightarrow{e} \mathrm{H',l',t'}}{\mathrm{H,l,C[t]} \xrightarrow{e} \mathrm{H',l',C[t']}}$

# Solving Problem 1 - Isolation of property names

## Textual Property

Given a program $P$, all property names that get accessed must appear textually in the code.

**Issues**:

- Undecidable in general: Not all code appears textually.
- Implicitly accessed property names: $\mathcal{P}_{nat} = \{0,1,2,\ldots\} \bigcup$

$$\left\{ \begin{array}{l} \text{toString, valueOf, length, prototype,} \\ \text{constructor, message, arguments, Object, Array} \end{array} \right\}$$

## Textual Property (revised)

Define a meaningful sublanguage $Jt$ such that for any program $P \in Jt$, if execution of $P$ accesses property $p$ of some object, then either $p \in Prop_{nat}$ or $p$ appears textually in $P$.

# Solving Problem 1 - Isolation of property names

## Textual Property

Given a program $P$, all property names that get accessed must appear textually in the code.

**Issues**:

- Undecidable in general: Not all code appears textually.
- Implicitly accessed property names: $\mathcal{P}_{nat} = \{0,1,2,...\} \bigcup$

$$\left\{ \begin{array}{l} \text{toString, valueOf, length, prototype,} \\ \text{constructor, message, arguments, Object, Array} \end{array} \right\}$$

## Textual Property (revised)

Define a meaningful sublanguage $Jt$ such that for any program $P \in Jt$, if execution of $P$ accesses property $p$ of some object, then either $p \in Prop_{nat}$ or $p$ appears textually in $P$.

# Sublanguage *Jt*

Designing the subset:

- Add separate sorts for strings (m), property names (mp) and identifiers(x) in the semantics.
- Identify "bad" reduction rules in the semantics which involve conversions:

| | | | |
|---|---|---|---|
| Strings | $\longrightarrow$ | Property names | (like e[e]) |
| Strings | $\longrightarrow$ | Code | (*like* eval) |

- Terms whose reduction can potentially invoke one of these rules are bad.

**Defining *Jt***

*Jt* is defined as ECMA-262 MINUS: eval, Function, hasOwnProperty, propertyIsEnumerable, constructor; e[e], e in e; for (e in e) s;

# Theorem 1

Notation:

- $\tau(S)$: Reduction trace of state $S = S_1 \rightarrow, \ldots, \rightarrow S_n, \rightarrow \ldots$
- $\mathcal{A}(S)$: Set of property names that get accessed during a single step state transition. Extended naturally to traces.
- $\mathcal{N}(S)$: Set of all identifiers appearing in state $S$.
- $Initial(J)$: Initial states corresponding to terms in subset $J$.

**Theorem 1 (Isolating property names)**

For all well-formed states $S_0$ in $Initial(Jt)$ ,

$$\mathcal{A}(\tau(S_0)) \subseteq \text{Id2Prop}(\mathcal{N}(S_0)) \cup \mathcal{P}_{nat}.$$

# Glimpse of the proof

Let $\mathcal{R}^{bad}$ be the set of reduction rules which involve conversions from Strings to Property Names/Identifiers/Programs.

**Main Lemma**

For all initial states terms in $S_0$ in $Initial(Jt)$, the reduction trace of $S_0$ never involves a reduction rule from $\mathcal{R}^{bad}$.

*Proof Idea*

- Standard inductive invariant based proof but on a VERY huge and non-straightforward semantics.
- (Hard part) Find a goodness predicates on states so that

Init $\forall S_0 \in Initial(Jt) : Good(S_0)$.

Induction $\forall S_1, S_2 : Good(S_1) \;\wedge\; S_1 \rightarrow S_2 \Rightarrow Good(S_2)$.

Safety $\forall S : Good(S) \rightarrow$ No rule from $\mathcal{R}^{bad}$ applies to S.

# The goodness predicate *Good*

## Term goodness

We say that a term $t$ is *good*, denoted by $Good_{Jt}(t)$ iff it has the following properties

- Structure of $t$ does not contain any of eval, Function, hasOwnProperty, propertyIsEnumerable and constructor as property names or identifiers
- Structure of $t$ does not contain any sub terms with any contexts of the form eforin(), pforin(), cEval(), FunParse() or [ ] contexts and any constructs of the form e in e, for (e in e) s and e[e].
- Structure of $t$ does not contain any of the heap addresses $l_{Function}, l_{eval}, l_{hOP}, l_{pIE}$

# The goodness predicate *Good*

## Heap goodness

We say that a heap is *good*, denoted by $Good_{Jt}(H)$, iff $H$ has the following properties

$$
\begin{array}{lcl}
\forall l, p : H(l).p = l_{Function} & \Rightarrow & p = \text{constructor} \\
 & & \quad \vee \;\; p = \text{Function} \\
\forall l, p : H(l).p = l_{eval} & \Rightarrow & p = \text{eval} \\
\forall l, p : H(l).p = l_{hOP} & \Rightarrow & p = \text{hasOwnProperty} \\
\forall l, p : H(l).p = l_{pIE} & \Rightarrow & p = \text{propertyIsEnumerable}
\end{array}
$$

# Isolating scope object

- For initial empty heap state, scope object is only accessible during scope resolution and using this.
- Heap addresses accessed during scope resolution are never returned at the top level.
- Object.prototype.valueOf, Array.prototype.sort/concat/reverse can potentially return the global object.

**Defining *Js***

The subset *Js* is defined as *Jt*, MINUS: this; valueOf, sort, concat and reverse;

# Isolating scope object

- For initial empty heap state, scope object is only accessible during scope resolution and using this.
- Heap addresses accessed during scope resolution are never returned at the top level.
- Object.prototype.valueOf, Array.prototype.sort/concat/reverse can potentially return the global object.

**Defining *Js***

The subset *Js* is defined as *Jt*, MINUS: this; valueOf, sort, concat and reverse;

# Theorem 2

Notation:

- $l_G$ : Heap address of global object.
- $\mathit{Final}(S)$: Final term in $\tau(S)$ (if it exists)
- $\mathcal{V}(S')$: Value part of the state (if it exists).

**Theorem 2 (Isolation of scope object)**

For any well-formed state $S_0 \in \mathit{Initial}(Js)$, let $S' = \mathit{Final}(S_0)$.

$$\mathcal{V}(S') \neq l_G$$

# Additional Property: Identifier Renaming

**Property**

No program in subset *Js* can get a handle to a scope object
$\Rightarrow$ variable names can never be accessed as properties
$\Rightarrow$ variable names can be renamed without modifying the semantics.

Catch: Don't rename variable with same name as native properties.

- toString() evaluates to *"[object␣Window]"*
- a123_toString() evaluates to Reference error.

**Theorem 3 (Closure under renaming)**

For all well-formed states $S_0$ in *Initial*(*Js*), if $\alpha$ is a safe state renaming function with respect to $S_0$, then $\alpha(\tau(S_0))$ equals $\tau(\alpha(S_0))$.

# Solution for Facebook

Mitigating the attack

- Define `$FBJS.ref` in a different name-space.
- Disallow application code from accessing property names beginning with "$".

A mostly syntactic alternative to FBJS: *Js*.

- Blacklisting security critical properties will isolate library code.
- Supports variable renaming so global variables from untrusted code form different applications can be isolated.
- Downside : Too conservative

Building on this Work

- We extended language with run-time checks (rewriting/wrapping) in spirit of FBJS and showed that the safety properties hold (W2SP'09, ESORICS'09).
- Resulting language is very close to FBJS in terms of expressivity.

# Solution for Facebook

Mitigating the attack

- Define `$FBJS.ref` in a different name-space.
- Disallow application code from accessing property names beginning with "$".

A mostly syntactic alternative to FBJS: *Js*.

- Blacklisting security critical properties will isolate library code.
- Supports variable renaming so global variables from untrusted code form different applications can be isolated.
- Downside : Too conservative

Building on this Work

- We extended language with run-time checks (rewriting/wrapping) in spirit of FBJS and showed that the safety properties hold (W2SP'09, ESORICS'09).
- Resulting language is very close to FBJS in terms of expressivity.

# Ongoing and Future Work

On the Usefulness side

- Extend the above results to apply to JavaScript supported by various browsers which include features beyond the ECMA-262 spec, such as getter, setters etc.
- Design mechanisms to enforce partial (and controlled) isolation between untrusted code (Think Mashups !).

On the Technical side

- Write the semantics in machine readable format so that the proofs can be automated.
- Express proofs of safety for subsets so that they are extensible.

Thank You !