Language-Based Isolation of Untrusted JavaScript

Sergio Maffeis
EPSRC Career Acceleration Fellow,
Imperial College London.

In collaboration with John Mitchell and Ankur Taly,
Stanford University.

CREST Open Workshop, April 6, 2011.

Motivation

- We want to improve the security of web applications using programming-language techniques.
- We focus on the client-side (more standardized).
  - JavaScript is *the* language of the browser.
- Examples: web advertising and social networking.
  - Benefit from embedding third-party code.
- Problem: *let trusted and untrusted JavaScript code interact safely in the same execution environment.*

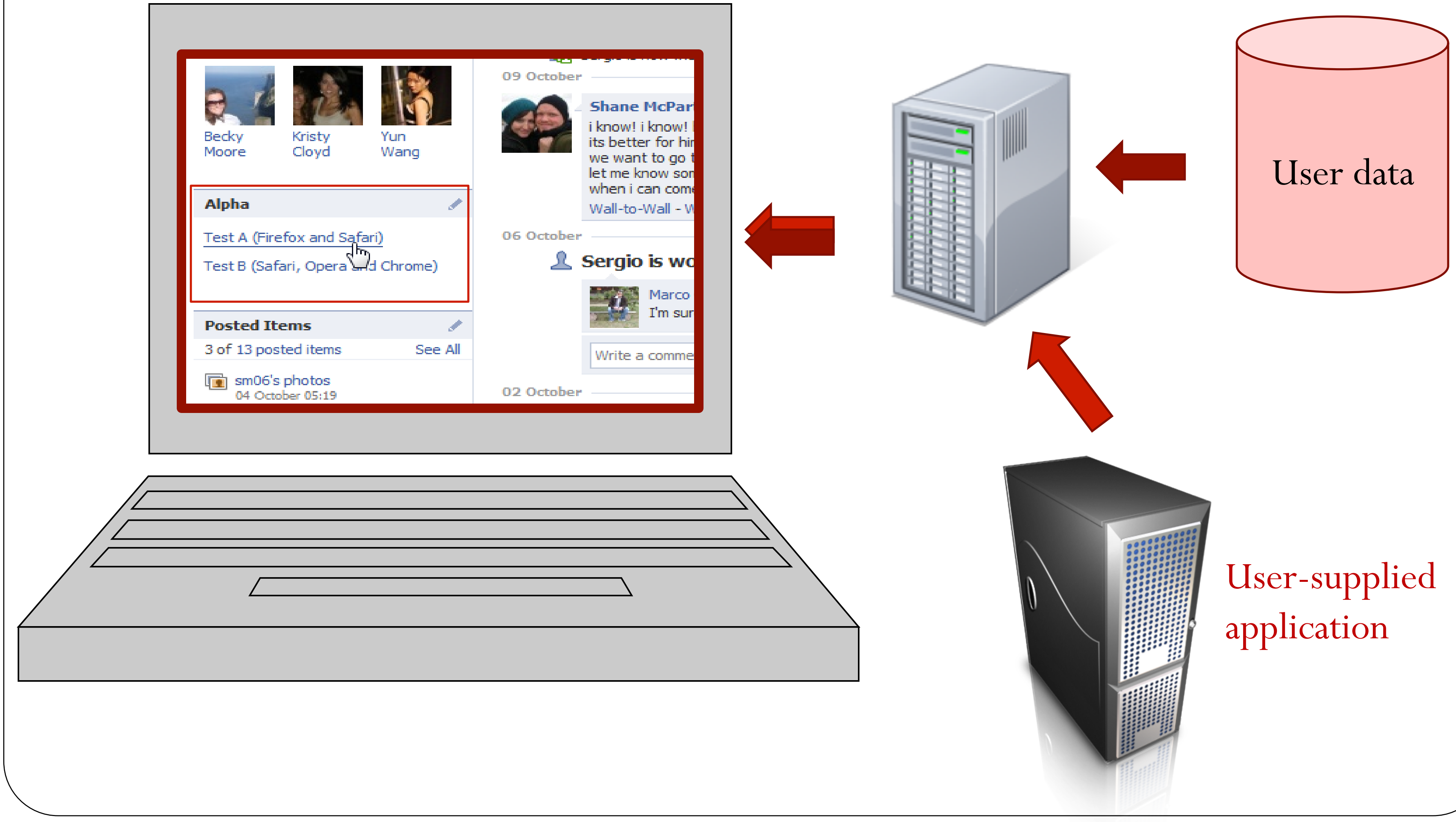
JavaScript: challenges

- Prototype-based object inheritance:
  `Object.prototype.a=“foo”;`
- Objects as mutable records of functions with implicit `this`:
  `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, that can be redefined.
  `Object.prototype.toString = o.b;`

JavaScript: operational semantics

- We built a small-step semantics amenable to formal proofs.
  - Focus on the standardized ECMA 3 (hence, no DOM).
  - Model validated experimentally with browsers and shells.
  - Theorems about sanity-check properties.
- Operational semantics for *real* programming languages is hard.
  - Sheer size.
  - JavaScript challenges.
  - Established techniques do not work.
    - `while(e){s}` ≠ `if(e){s;while(e){s}};`
    - `var x={}; x+1;` depends on `Object.prototype.valueOf;`

Third-party content: Apps



User data

User-supplied application

Browser-based JavaScript sandbox

- Same Origin Policy and inline frames can sandbox untrusted code in an isolated execution environment.
- There is a considerable price to pay.
  - Full JavaScript can be too powerful.
  - Interactions with other applications are severely limited.
  - Framed applications are restricted to a confined region of the screen.

JavaScript sandboxing JavaScript

- A different approach.
  - Trusted and untrusted JavaScript run in the same execution environment.
  - Trusted code enforces a software sandbox on the untrusted code.
  - Fine grained control on the interaction between applications.
- We singled out three instances:
  - Facebook FBJS (viral social network).
  - Yahoo’s ADsafe (high value advertising).
  - Google Caja (web gadget platform).

Security goal

- Concrete security goals.
  - No direct access to the DOM.
  - No tampering with the execution environment.
- Idea: blacklist global variables (`document`, `Object`,… & host libraries).
- Not easy to enforce in JavaScript.
  - Reflection
  - Semantics oddities
  - Implicit accesses, …
- A solution must be compatible with running multiple untrusted apps.

Our blacklisting subset

- B is a list of identifiers not to be accessed by untrusted code.
- P_{nat} is the set of identifiers that can be accessed implicitly.
  - For example reading `Object` or writing `length`.
- Solution: we can enforce B (compatibly with P_{nat}) by filtering and rewriting untrusted code.
  - Disallowing all terms containing an identifier from B.
  - Including `eval`, `Function` and `constructor` in B by default.
  - Rewriting `e1[e2]` to `e1[IDX(e2)]`.

The run time monitor IDX

- We need auxiliary variables, prefixed with `$` and included in B.

```
var $String=String;
var $B={p1:true;...,pn:true,eval:true,…,$:true,…};
```

- Rewrite `e1[e2]` to `e1[IDX(e2)]`, where

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

- Our rewriting faithfully emulates the semantics.

```
e1[e2] ➔ va1[e2] ➔ va1[va2] ➔ l[va2] ➔ l[m]
```

Evaluation

- Theorem: our JavaScript subset prevents access to the identifiers in B (compatibly with P_{nat}).
- Our enforcement does not alter the semantics of good code.
- Two main limitations.
  - Variables are blacklisted together with property names.
    - If `x` is blacklisted, we must blacklist also `obj.x`.
    - Heavy to separate namespaces of multiple applications.
  - Default blacklisting of `eval`, `Function`.

Preventing scope manipulation

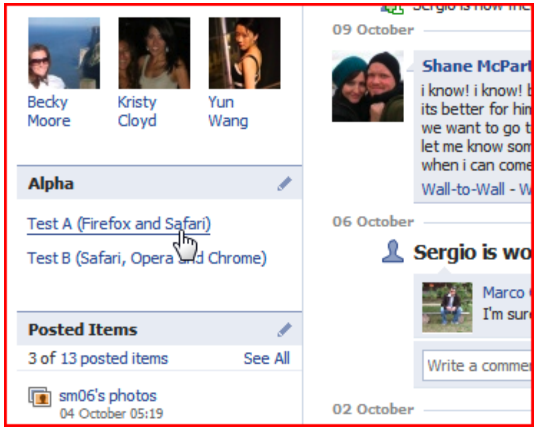
- We want to prevent explicit access to scope objects.

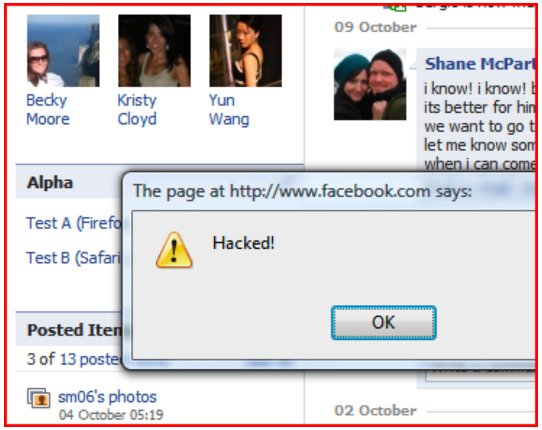
  `this.x=1; var o={y:41}; with (o){x+y};`
- The global scope (in this talk).
  - Evaluate `window` or `this` in the global environment.
  - Evaluate `(function(){return this})()`.
  - Call native functions with same semantics as above {`sort`, `concat`, `reverse`, `valueOf`}.
- Local scope objects (see papers).

Isolating the global scope

- Enforcement mechanism.
  - Save reference to global object in a private (blacklisted) variable.
    `var $Global=window;`
  - Rewrite `this` to `(this==$Global?null,this)`.
- No need to blacklist `sort`, `concat`, `reverse`, `valueOf`.
  - We can wrap them and sanitize returned values in a similar fashion.
- Benefits of isolating the global scope.
  - Statically filter out the global variables to be protected, no need to include them in the runtime blacklist used by `IDX`.
  - Multiple apps can coexist easily (only global variables need to be disjoint).

Comparison with Facebook



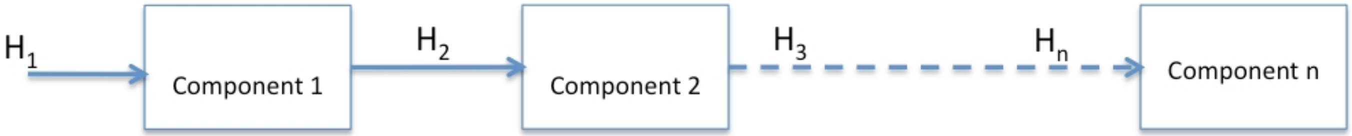


The page at http://www.facebook.com says:

Hacked!

- Our subsets are similar to FBJS but:
  - Preserve original semantics more closely.
  - Proofs increase confidence in the correctness.
- Differences pointed to vulnerabilities in FBJS (and Yahoo! ADsafe).
  - Exploits: we built FBJS applications able to reach the DOM.
  - We proposed fixes to Facebook.
  - Considerable potential for damage (popular apps have 20M+ users).

Inter-component isolation



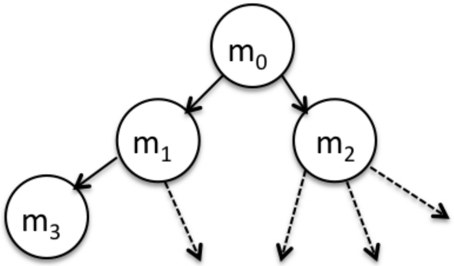
- Components: JavaScript programs $t_1, \ldots t_n$.
- Mashup: sequential composition $t_1; \ldots ; t_n$.
- Shared resources: JavaScript heap locations.
- *Inter-component isolation*:

  Verify/enforce that any two components access disjoint sets of resources.

Capability safe languages

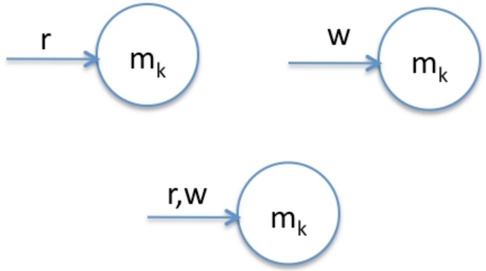
- Each program is endowed with capabilities, which are its only means for designating and accessing resources.
- Our approach.
  - Given a programming language, define formally:
    - *Capability systems*.
    - *Capability safety*.
  - Use *capability safety* to check *inter-component isolation*.

Capability systems: definitions



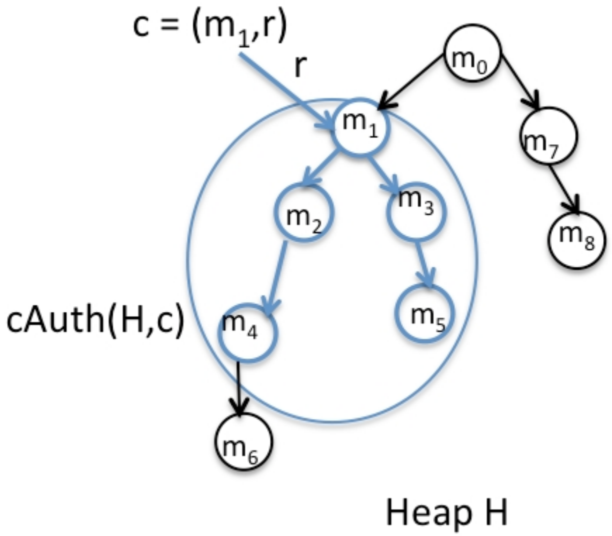
- Resources:
  - Smallest granularity of read/write heap locations m_0, m_1, …
  - Typically organized as a graph.
- Subjects:
  - Entities that access resources.
  - Program expressions t_0, t_1, …

Capability systems: definitions



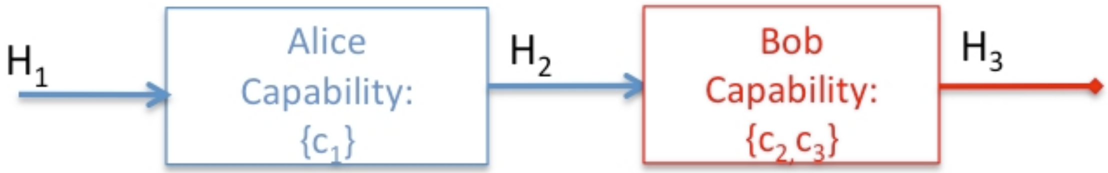
- Capability *C*:
  - Unforgeable entity that designates and provides access to a resource.
  - Pair *(m,p)* of resource *m* and permission *p* in {*r,w*}.
- Subject-capability map *tCap*:
  - Each subject is endowed with certain capabilities.
  - *tCap(t)* is the set of capabilities associated with subject *t*.

Authority



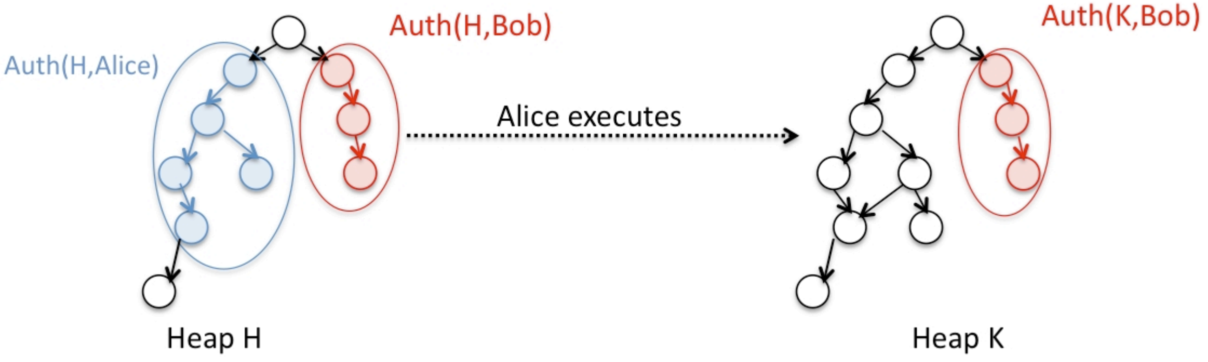
- Authority of a capability *cAuth*:
  - Upper-bound on resources that can be accessed using the capability.
  - *cAuth(H,c)* is the authority of capability *c* in heap *H*.
- Authority of a subject *Auth*:
  - Subjects hold capabilities which provide authority.
  - $Auth(H, t) = \bigcup_{c \text{ in } tCap(t)} cAuth(H,c)$ is the authority of subject *t* in heap *H*.

Capabilities and mashup isolation



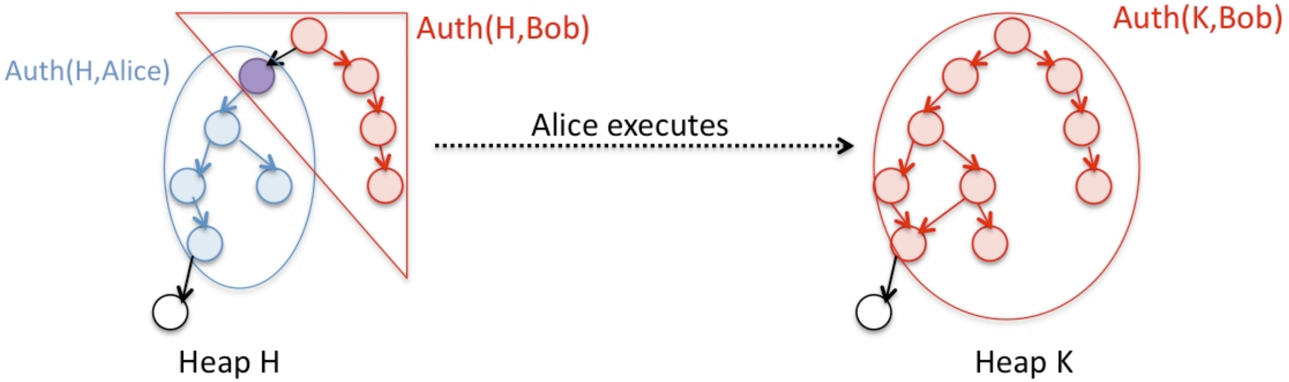
- Idea: allocate capabilities with disjoint authority to Alice and Bob.
  - The authority of a capability depends on the heap.
  - We would like $Auth(H_1, Alice) \cap Auth(H_2, Bob) = \varnothing$.
  - But we know only H_1…
- Strategy:
  - Define a stronger property (*capability safety*) so that it is enough to check $Auth(H_1, Alice) \cap Auth(H_1, Bob) = \varnothing$.

Only connectivity begets connectivity



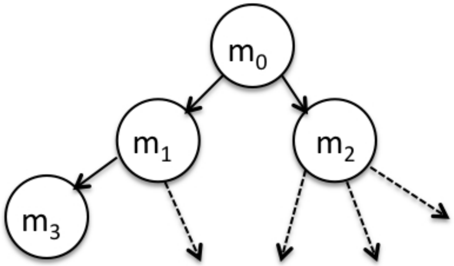
- IF the authority of Alice and Bob in *H* does not overlap THEN Bob's authority does not change.

No authority amplification



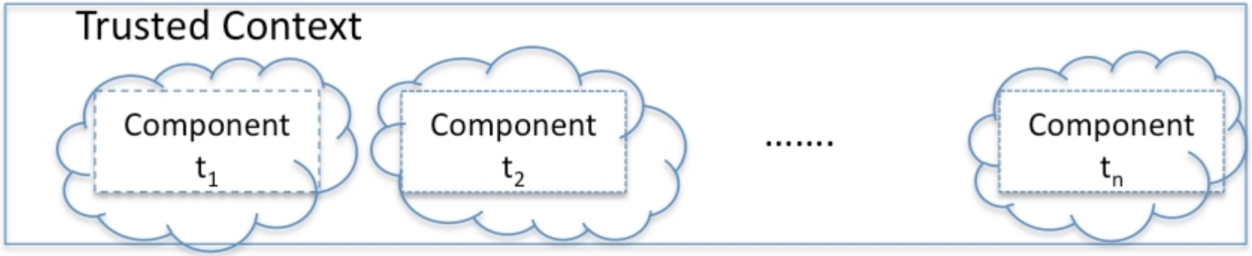
- IF the authority of Alice and Bob in *H* *does* overlap THEN Bob's authority in *K* is at-most:
  - the union of Alice's and Bob's authority in *H*;
  - plus any new authority created by Alice.

Capability safety



- A *capability system* [*C*,*tCap(t)*,*cAuth(H,c)*] is *safe* iff
  1. All access derives from capabilities.
  2. The authority of a capability satisfies topology-only bounds.
  3. *Only connectivity begets connectivity* holds for *cAuth*.
  4. *No authority amplification* holds for *cAuth*.

Isolation Theorem



- *Authority isolation*:
  - Given a heap H and components $t_1, \ldots, t_n$, *authority isolation* holds iff for all $i \neq j$, $Auth(H,t_i)$ and $Auth(H,t_j)$ do not overlap.
- Theorem: *authority isolation* implies *inter-component isolation*.
- The result holds for any sequential imperative language.

Applications of the Isolation Theorem

- JavaScript mashups:
  - We proved that a variant of our JavaScript subset for host isolation is *capability safe*.
  - We derived an enforcement function that guarantees *authority isolation*.
    - Make native function objects read-only.
    - Wrap native functions so they never receive the global object as `this`.
- Google Caja:
  - We formalized the core of the Cajita subset of JavaScript.
  - We proved that our model of Cajita is *capability safe*.

Concluding remarks

- We used programming language techniques to study safe JavaScript subsets.
  - Provably correct solutions.
  - Validated by experiment.
  - Impact on real applications.
- Limitations.
  - Proofs by hand are long and error-prone.
  - We separate components. What about controlled interaction?
- Future work.
  - Mechanization of semantics in a proof assistant.
  - Tool to enforce subsets and scan libraries.

Language-Based Isolation of Untrusted JavaScript

Sergio Maffeis
EPSRC Career Acceleration Fellow,
Imperial College London.

In collaboration with John Mitchell and Ankur Taly,
Stanford University.

CREST Open Workshop, April 6, 2011.