Object Capabilities and Isolation of Untrusted Web Applications

Sergio Maffeis

EPSRC Research Fellow, Imperial College London

Joint work with John C. Mitchell and Ankur Taly (Stanford University).

OWASP AppSec Research 2010, Stockholm.
Motivation
Moivation
Approaches to Isolation

Different ways to isolate mashup components:

- Client-side browser abstractions/extensions.
  - SOP+IFRAME, Beep, iGoogle, etc.
- Server side filtering and rewriting.
  - FBJS, ADSafe, Caja.

Our approach: use formal programming language techniques to

- Formalize server-side solutions.
- Study their security properties.
- Design new enforcement mechanisms.

Formal proofs increase confidence and often help to discover bugs!!
Basic Mashups

Mashup with non-interacting components.

- Client-side language: JavaScript.
  - In the paper: any sequential imperative language with a small-step operational semantics.
- Mashup components: programs $t_1, \ldots, t_n$ in JavaScript.
- Mashup: sequential composition $t_1; \ldots; t_n$.
- Shared Resource: program heap.
Mashup Isolation Problem

Verify/Enforce the following:

1. **Host Isolation**: No component can access any security-critical resource of the hosting page (e.g. `window.location`).

2. **Inter-component Isolation**: For all $i, j$, component $i$ and $j$ must access disjoint set of heap resources.

State of the art:

1. We know how to enforce host isolation (CSF'09, ESORICS'09).

2. Inter-component isolation is tricky:
   - Library functions are implicitly shared by components.
   - Need complete privilege separation.
Mashup Isolation Problem

Verify/Enforce the following:

1. **Host Isolation**: No component can access any security-critical resource of the hosting page (e.g. `window.location`).

2. **Inter-component Isolation**: For all $i, j$, component $i$ and $j$ must access disjoint set of heap resources.

State of the art:

1. We know how to enforce host isolation (CSF’09, ESORICS’09).

2. Inter-component isolation is tricky:
   - Library functions are implicitly shared by components.
   - Need complete privilege separation.
Main Idea: Each program is endowed with some capabilities, which are its only means for designating and accessing resources.

Object Capability languages (Rees, Stiegler, Wagner, Miller):
- Capability ideas applied to object-oriented languages.
- Properties: Connectivity begets Connectivity, No Authority Amplification, Defensive Consistency, ...

Intuitively seems very relevant for mashup isolation.

We need formal definitions for carrying out rigorous proofs.
Plan

- Given a programming language, define formally
  - Capability Systems.
  - Capability Safety.

- Use Capability Safety to check inter-component isolation.

- Validate the approach using realistic examples.
Capability Systems: Basic Features

**Resources** \((m_0, m_1, \ldots)\)

- *Smallest granularity of read/write locations on the heap.*
- Typically organized as a graph.

**Subjects:**

- *Entities that access resources.*
- Program expressions \(t_0, t_1, \ldots\)
Capability

Capability ($C$)

- Unforgeable entity that designates and provides access to a resource.
- Pair $(m, p)$ of resource $m$ and permission $p \subseteq \{r, w\}$.

Subject-Capability Map $tCap$

- Each subject possesses certain capabilities.
- $tCap(t)$ is the set of capabilities associated with subject $t$. 
Authority

Authority of a Capability ($c_{Auth}$)

- **Upper-bound on resources that can be accessed using the capability.**

- $cAuth(H, c)$ is the authority of capability $c$ w.r.t heap $H$.

Authority of a Subject ($Auth$)

- Subjects hold capabilities which provide authority.

- $Auth(H, t) = \bigcup_{c \in tCap(t)} cAuth(H, t)$ is the authority of subject $t$ w.r.t heap $H$. 

Heap $H$
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 $c$ w.r.t heap $H$.

Authority of a Subject (Auth)
- Subjects hold capabilities which provide authority.
- $Auth(H, t) = \bigcup_{c \in tCap(t)} cAuth(H, t)$ is the authority of subject $t$ w.r.t heap $H$. 
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 $c$ w.r.t heap $H$.

Authority of a Subject (Auth)
- Subjects hold capabilities which provide authority.
- $Auth(H, t) = \bigcup_{c \in tCap(t)} cAuth(H, t)$ is the authority of subject $t$ w.r.t 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.
- \( \text{Auth}(H_1, Alice) \cap \text{Auth}(H_2, Bob) = \emptyset \) must hold.
- But we have only \( H_1 \)...

Next few slides

We define capability safety and show that for safe systems, checking \( \text{Auth}(H_1, Alice) \cap \text{Auth}(H_1, Bob) = \emptyset \) is sufficient.
Idea: allocate capabilities with **disjoint authority** to Alice and Bob.

- The authority of a capability depends on the heap.
- $$\text{Auth}(H_1, Alice) \cap \text{Auth}(H_2, Bob) = \emptyset$$ must hold.
- But we have only $$H_1$$...

**Next few slides**

We define **capability safety** and show that for safe systems, checking $$\text{Auth}(H_1, Alice) \cap \text{Auth}(H_1, Bob) = \emptyset$$ is sufficient.
A capability system $[C, tCap(t), cAuth(H, c)]$ is safe iff

1. All Access derives from Capabilities
2. Authority of a capability satisfies topology-only bounds
3. Only Connectivity begets Connectivity
4. No Authority Amplification

Capabilities systems have other interesting properties, but these are sufficient for isolation.
Capability Safety

A capability system \([C, tCap(t), cAuth(H, c)]\) is safe iff

1. All Access derives from Capabilities
2. Authority of a capability satisfies topology-only bounds
3. Only Connectivity begets Connectivity
4. No Authority Amplification

Capabilities systems have other interesting properties, but these are sufficient for isolation.
Consider principals Alice and Bob.

- Alice executes and changes the heap from \( H \) to \( K \).
- **Only Connectivity begets Connectivity** and **No Authority Amplification** give us a relation between \( \text{Auth}(H, Bob) \) and \( \text{Auth}(K, Bob) \).
Only Connectivity begets connectivity

IF Bob’s and Alice’s authority with respect to $H$ do not overlap

THEN Bob’s authority stays the same

Formally, $\text{Auth}(K, Bob) = \text{Auth}(H, Bob)$
Only Connectivity begets connectivity

IF Bob’s and Alice’s authority with respect to $H$ do not overlap

THEN Bob’s authority stays the same

Formally, $\text{Auth}(K, Bob) = \text{Auth}(H, Bob)$
No Authority Amplification

IF Bob’s and Alice’s authority with respect to $H$ do overlap
THEN Bob’s authority w.r.t $K$ is at-most

- Both Alice’s and Bob’s authority w.r.t $H$.
- Any new authority created by Alice.

Formally, $\text{Auth}(K, Bob) \subseteq \text{Auth}(H, Bob) \cup \text{Auth}(H, Alice) \cup \text{Act}(K)$
No Authority Amplification

IF Bob’s and Alice’s authority with respect to $H$ do overlap
THEN Bob’s authority w.r.t $K$ is at-most

- Both Alice’s and Bob’s authority w.r.t $H$.
- Any new authority created by Alice.

Formally, $Auth(K, Bob) \subseteq Auth(H, Bob) \cup Auth(H, Alice) \cup Act(K)$
Isolation Theorem

Definition: Authority-Isolation

For an initial heap $H$ and components $t_1, \ldots, t_n$, authority isolation holds iff for all $i, j, i \neq j$:

$Auth(H, t_i)$ and $Auth(H, t_j)$ do not overlap

Theorem

Authority-Isolation $\Rightarrow$ Inter-component Isolation

Proven for any sequential imperative language (operational semantics).
Isolation Theorem

Definition: Authority-Isolation
For an initial heap $H$ and components $t_1, \ldots, t_n$, authority isolation holds iff for all $i, j, i \neq j$:

$\text{Auth}(H, t_i)$ and $\text{Auth}(H, t_j)$ do not overlap

Theorem
Authority-Isolation $\Rightarrow$ Inter-component Isolation

Proven for any sequential imperative language (operational semantics).
Generalization: Authority Safety

Isolation Theorem only depends on an authority $Auth(H, t)$, such that:

1. All resources accessed during the reduction of $H, t$ are in $Auth(H, t)$.
2. $Auth$ satisfies “Only Connectivity begets Connectivity”.
3. $Auth$ satisfies “No Authority Amplification”.

We call the above 3 properties as Authority Safety.

- Capability systems provide a natural definition of authority:
  \[ Auth(H, t) = \bigcup_{c \in Cap(t)} cAuth(H, t). \]
- There are many other possible definitions.
Generalization: Authority Safety

Isolation Theorem only depends on an authority $\text{Auth}(H, t)$, such that:

1. All resources accessed during the reduction of $H, t$ are in $\text{Auth}(H, t)$.
2. $\text{Auth}$ satisfies “Only Connectivity begets Connectivity”.
3. $\text{Auth}$ satisfies “No Authority Amplification”.

We call the above 3 properties as Authority Safety.

- Capability systems provide a natural definition of authority:

  $$\text{Auth}(H, t) = \bigcup_{c \in t\text{Cap}(t)} c\text{Auth}(H, t).$$

- There are many other possible definitions.
Applications of the Isolation Theorem

Procedure for building safe Mashups

1. Prove that a language is capability safe or authority safe.
2. Derive an enforcement function that provides authority isolation for different components.

JavaScript Mashups

- We defined $J_{safe} \subseteq$ JavaScript, and proved that it is authority safe.
- We derived an enforcement function that guarantees authority isolation.

Google Caja

- We formalized the core of Cajita $\subseteq$ JavaScript.
- We proved that our model of Cajita is capability safe.
Applications of the Isolation Theorem

Procedure for building safe Mashups

1. Prove that a language is capability safe or authority safe.
2. Derive an enforcement function that provides authority isolation for different components.

JavaScript Mashups

- We defined $J_{safe} \subseteq$ JavaScript, and proved that it is authority safe.
- We derived an enforcement function that guarantees authority isolation.

Google Caja

- We formalized the core of Cajita $\subseteq$ JavaScript.
- We proved that our model of Cajita is capability safe.
Applications of the Isolation Theorem

Procedure for building safe Mashups

1. Prove that a language is capability safe or authority safe.
2. Derive an enforcement function that provides authority isolation for different components.

JavaScript Mashups

- We defined $J_{safe} \subseteq$ JavaScript, and proved that it is authority safe.
- We derived an enforcement function that guarantees authority isolation.

Google Caja

- We formalized the core of Cajita $\subseteq$ JavaScript.
- We proved that our model of Cajita is capability safe.
**$J_{safe}$: Enforcing Host Isolation**

We define a subset of JavaScript which

1. Has a meaningful **safe** authority map.
2. Supports an **enforcement mechanism** for authority isolation.

We start with subset $J_{sub}$ defined in ESORICS’09.

- Subset defined using **Filtering, Rewriting, Wrapping** for preventing access of security-critical resources.
  - Filter `eval`, Rewrite $e_1[e_2]$ to $e_1[IDX(e_2)]$.
  - Wrap native functions . . .

- Ensures that **authority** of any term does not contain security-critical resources.
**$J_{safe}$**: Enforcing Host Isolation

We define a subset of JavaScript which

1. Has a meaningful safe authority map.
2. Supports an enforcement mechanism for authority isolation.

We start with subset $J_{sub}$ defined in ESORICS’09.

- Subset defined using **Filtering**, **Rewriting**, **Wrapping** for preventing access of security-critical resources.
  - Filter `eval`, Rewrite $e_1[e_2]$ to $e_1[IDX(e_2)]$.
  - Wrap native functions . . .

- Ensures that authority of any term does not contain security-critical resources.
$J_{safe}$: Enforcing Authority Isolation

**Name space separation:** Rename variables in different components into disjoint namespaces.

- Almost Works, but some authority overlap still exists.
  - Communication via native objects.
    
    Alice: `Alice.o.toString.channel = <msg>`
    Bob: `Bob.o.toString.channel`
  
  - Communication using side-effect cause native functions.
    
    Alice: `Alice_push = [].push; Alice_push(<msg>)`
    Bob: `Bob_pop = [].pop; Bob_pop()`

- **Fix:**
  
  - Make native function objects readonly
  
  - Wrap native functions so that they never get the global object as the this object.

The resulting subset is called $J_{safe}$. 
**J\textsubscript{safe}: Enforcing Authority Isolation**

**Name space separation:** Rename variables in different components into disjoint namespaces.

- Almost Works, but some authority overlap still exists.
  - Communication via native objects.
    - Alice: `Alice.o.toString.channel = <msg>`
    - Bob: `Bob.o.toString.channel`
  - Communication using side-effect cause native functions.
    - Alice: `Alice.push = [].push; Alice.push(<msg>)`
    - Bob: `Bob.pop = [].pop; Bob.pop()`

**Fix:**

- Make native function objects readonly
- Wrap native functions so that they never get the global object as the this object.

The resulting subset is called \( J\text{safe} \).
$J_{safe}$: Enforcing Authority Isolation

Name space separation: Rename variables in different components into disjoint namespaces.
- Almost Works, but some authority overlap still exists.
  - Communication via native objects.
    Alice: Alice.o.toString.channel = <msg>
    Bob: Bob.o.toString.channel
  - Communication using side-effect cause native functions.
    Alice: Alice_push = [].push; Alice_push(<msg>)
    Bob: Bob_pop = [].pop; Bob_pop()

Fix:
- Make native function objects readonly
- Wrap native functions so that they never get the global object as the this object.

The resulting subset is called $J_{safe}$. 
**\( J_{safe} \): Enforcing Authority Isolation**

**Name space separation**: Rename variables in different components into disjoint namespaces.

- Almost Works, but some authority overlap still exists.
  - Communication via native objects.
    - Alice: `Alice.o.toString.channel = <msg>`
    - Bob: `Bob.o.toString.channel`
  - Communication using side-effect cause native functions.
    - Alice: `Alice_push = [].push; Alice_push(<msg>)`
    - Bob: `Bob_pop = [].pop; Bob_pop()`

- **Fix:**
  - Make native function objects *readonly*
  - Wrap native functions so that they never get the global object as the `this` object.

The resulting subset is called \( J_{safe} \).
$J_{safe}$ is authority safe

Main Contributions:

- We define an authority map $Auth_{J_{safe}}(H, t)$ for all heaps $H$ and programs $t$.
- **Theorem 1**: $Auth_{J_{safe}}(H, t)$ is a safe authority map.
- **Theorem 2**: Namespace separation enforces authority isolation for $J_{safe}$ programs.

Remarks:

- $J_{safe}$ is more expressive than Facebook $FBJS$ and Yahoo! $ADsafe$.
- Thinking in terms of authority helped us find new attacks on Facebook $FBJS$ and Yahoo! $ADsafe$. (See the paper!)
$J_{safe}$ is authority safe

Main Contributions:

- We define an authority map $Auth_{J_{safe}}(H, t)$ for all heaps $H$ and programs $t$.
- **Theorem 1**: $Auth_{J_{safe}}(H, t)$ is a safe authority map.
- **Theorem 2**: Namespace separation enforces authority isolation for $J_{safe}$ programs.

Remarks:

- $J_{safe}$ is more expressive than Facebook $FBJS$ and Yahoo! $ADsafe$.
- Thinking in terms of authority helped us find new attacks on Facebook $FBJS$ and Yahoo! $ADsafe$. *(See the paper!)*
Conclusions

Results:
- Capability Safety $\Rightarrow$ Authority Safety $\Rightarrow$ Isolation.
- $J_{\text{safe}}$ is Authority safe.
- Cajita is Capability safe.

Future Work:
- Formalize other aspects of capability systems:
  - absolute encapsulation,
  - defensive consistency.
- Use the above for controlling interaction between components.
Object Capabilities and Isolation of Untrusted Web Applications

Sergio Maffeis

EPSRC Research Fellow, Imperial College London

Joint work with John C. Mitchell and Ankur Taly (Stanford University).

OWASP AppSec Research 2010, Stockholm.