Type-based Dependency Analysis for JavaScript
PLAS’13

Matthias Keil, Peter Thiemann
Institute for Computer Science
University of Freiburg
Freiburg, Germany

June 20, 2013, Seattle, WA, USA.
- JavaScript is the most important language for web sites
  - 92% of all websites use JavaScript
- Web-developers rely on third-party libraries
  - e.g. for calendars, maps, social networks

```html
<script type="text/javascript"
src="http://example.org/api/js/?ARGS">
</script>
```
Dynamic programming language
- e.g. eval, mashups

JavaScript has no security awareness
- No namespace or encapsulation management
- Global scope for variables/functions
- All scripts have the same authority

Security aspects of JavaScript received much attention
- Static or dynamic analysis techniques
- Guarantees by reducing the functionality
1. Libraries may get access to sensitive data
2. User code may be prone to injection attacks

- **Naive approach**: detect information flow
  - Pure information flow is too unflexible for investigating injection attacks
  - Ignores sanitized values

- **Our approach**: dependency analysis
  - Addresses both scenarios
  - Sanitized values are acceptable
  - Static Analysis
  - Implemented as extension of a type analyzer
Information flow

```javascript
var t = Cookie.get('access_token');
// processing
// ...
Ajax.request('example.org', t);
```
Information flow

```javascript
var t = Cookie.get('access_token');
// processing
// ...
Ajax.request('example.org', t);
```

Sanitization

```javascript
var input = document.getElementById('id');
function sanitizer(value) {
    // clean up value
}
// processing
// ...
Ajax.request('example.org', sanitizer(input));
```
Dependency Analysis
- Information flow pioneered by Denning
- Determines potential data flow between program points
- Related to simple security types

Flow-sensitive analysis
- Abstracts data tainting
- Stated as type-system
Built-in \( \text{trace}^\ell(e[, id]) \) and \( \text{untrace}(e, id) \) function

- \( \ell \) – unique taint
- \( id \) – tag name
- Behaves like an identity function

Implicit classes: UNSAFE, SAFE

Policy-file

- For predefined values (e.g. DOM, JavaScript API)

---

```
trace.policy

# Object Trace
trace: HTMLDocument.cookie; Ajax.request;
Array.prototype; Math.abs;
```
```javascript
var userHandler = function (uid) {
    var name = '';
    var onSuccess = function (response) { name = response; };
    if (alreadyLoaded) {
        Cookie.request(uid, onSuccess);
    } else {
        Ajax.request('example.org', uid, onSuccess);
    }
    return name;
};
var name = userHandler(trace("uid "));
```

- Security properties on a fine level of granularity
- Distinguish different sources
var userHandler = function(uid) {
    var name = ''; 
    var onSuccess = function(response) {name = response;}; 
    if (alreadyLoaded) { // alreadyLoaded=true
        Cookie.request(uid, onSuccess);
    } else {
        Ajax.request('example.org', uid, onSuccess);
    }
    return name;
};

var name = userHandler(trace("uid"));

- Security properties on a fine level of granularity
- Distinguish different sources
var userHandler = function (uid) {
    var name = ''; 
    var onSuccess = function (response) { name = response; };
    if (alreadyLoaded) { // alreadyLoaded=(true|false)
        Cookie.request(uid, onSuccess);
    } else {
        Ajax.request('example.org', uid, onSuccess);
    }
    return name;
};

var name = userHandler(trace("uid"));

- Security properties on a fine level of granularity
- Distinguish different sources
Load Foreign Code = trace(function() {
  Array.prototype.foreach = function(callback) {
    // do something
  }
};
loadForeignCode();
// do something
array.foreach(function(k, v) {
  result = k + v;
});

- Protect code from being compromised
- Encapsulation of foreign code
$ = \text{function}(id) \{ \\
\quad \text{return } \text{trace}(\text{document.getElementById(id).value, "#DOM");} \\
\} \\
\text{function sanitizer(value) \{ \\
\quad \text{// escape value} \\
\quad \text{return } \text{untrace}(\text{value, "#DOM");} \\
\} \\
\text{// do something} \\
\text{var input } = \$\("\text{text}\); \\
\text{var sanitizedInput } = \text{sanitizer(input);} \\
\text{consumer(sanitizedInput);} \\
\\\n\text{Avoid injection attacks} \\
\quad \text{e.g. only escaped values used} \\
\text{Change taint classes}
varsanitizedInput = i_know_what_i_do ? sanitizer(input) : input;

- Mixture of sanitized and unsanitized taints
- Flagged as an error
\begin{verbatim}
var input

v = trace(value, '\#DOM');

v' = trace(another, '\#ANOTHER');

sanitized = untrace(input, '\#DOM');
\end{verbatim}
$v = \text{trace}(\text{value}, '\#\text{DOM}');$

$\ell \#\text{DOM}, \text{UNSAFE}$

$\text{var} \ \text{input}$

$\text{sanitized} = \text{untrace}(\text{input}, '\#\text{DOM}');$

$\ell \#\text{DOM}, \text{UNSAFE}$

$v' = \text{trace}(\text{another}, '\#\text{ANOTHER}');$

$\ell' \#\text{ANOTHER}, \text{UNSAFE}$
\[ v = \text{trace}(\text{value}, \ '#\text{DOM}'); \]

\[ \nu : \ell#\text{DOM},\text{UNSAFE} \]

\[ \text{var input} \]

\[ \text{input} : \ell#\text{DOM},\text{UNSAFE}, \ell#\text{ANOTHER},\text{UNSAFE} \]

\[ v' = \text{trace}(\text{another}, \ '#\text{ANOTHER}'); \]

\[ \nu' : \ell#\text{ANOTHER},\text{UNSAFE} \]

\[ \text{sanitized} = \text{untrace}(\text{input}, \ '#\text{DOM}'); \]
\[ v = \text{trace}(\text{value}, '\#\text{DOM}'); \]

\[ v : \ell\#\text{DOM}, \text{UNSAFE} \]

\[ \text{var} \quad \text{input} \]

\[ \text{input} : \ell\#\text{DOM}, \text{UNSAFE}, \ell'\#\text{ANOTHER}, \text{UNSAFE} \]

\[ \text{sanitized} = \text{untrace}(\text{input}, '\#\text{DOM}'); \]

\[ \text{sanitized} : \ell\#\text{DOM}, \text{SAFE}, \ell'\#\text{ANOTHER}, \text{UNSAFE} \]

\[ v' = \text{trace}(\text{another}, '\#\text{ANOTHER}'); \]

\[ v' : \ell'\#\text{ANOTHER}, \text{UNSAFE} \]
input = trace(value, '#DOM');

if

input = untrace(input, '#DOM');
input = `trace(value, '#DOM');`

input : ℓ#DOM, UNSAFE

if

input : ℓ#DOM, UNSAFE

input = `untrace(input, '#DOM');`

input : ℓ#DOM, SAFE

input : ℓ#DOM, SAFE
input = \text{trace}(\text{value}, '\#\text{DOM}');

\text{if}

input = \text{untrace}(\text{input}, '\#\text{DOM}');

input : \ell^{\#\text{DOM}, \text{UNSAFE}}

input : \ell^{\#\text{DOM}, \text{SAFE}}\), \ell^{\#\text{DOM}, \text{UNSAFE}}
Formalization based on a typed JavaScript Core calculus

Dependency Tracking Semantics
- Marker propagation for upcoming values
- Not meant to perform a dynamic analysis

Static analysis based on the type system for dependency tracking
- Termination-insensitive noninterference based on the types
- Correct abstraction of the tracking semantics
- Termination of the abstract analysis
Implementation extends TAJS, a Type Analyzer for JavaScript developed by Anders Møller and others

- Abstract values and states are extended with abstract taints
- The control flow graph is extended by special nodes for implicit flows
- $\text{trace}^l$ and $\text{untrace}$ implemented as built-in functions
- Policy file for pre-labeling of built-in objects
Designed and implemented a type-based dependency analysis for JavaScript

- Analysis of information flow
- Encapsulation of foreign code
- Declassification of values (by changing taint-classes)

Dependency analysis is not a security analysis

- Investigate noninterference
- Ensure confidentiality
- Verify correct sanitizer placement
Questions?

Thank you for your attention.