ZigZag - Automatically Hardening Web Applications Against Client-side Validation Vulnerabilities

Michael Weissbacher, William Robertson, Engin Kirda, Christopher Kruegel, Giovanni Vigna
XMLHTTPRequest (XHR) and postMessage
XMLHttpRequest (XHR) and postMessage
XMLHttpRequest (XHR) and postMessage
XMLHTTPRequest (XHR) and postMessage
Client-Side Validation Vulnerabilities

- Bugs in client-side code of web application (JavaScript)
- Unsafe usage of untrusted data
- Validation
  - missing
  - broken
Client-Side Validation Vulnerabilities

- **Attacks**
  - Origin mis-attribution
  - Command injection
  - Cookie leakage

- **Defense / Detection**
  - Server-side detection not possible
  - Client-side
Example of CSV Vulnerability

```javascript
function listener(event){
    if (event.origin.indexOf("domain-a.ru") != -1){
        eval(event.data);
    }
}

if (window.addEventListener){
    window.addEventListener("message", listener, false);
} else {
    window.attachEvent("onmessage", listener);
}
```
Example of CSV Vulnerability

```javascript
function listener(event){
  if (event.origin.indexOf("domain-a.ru") != -1){
    eval(event.data);
  }
}
```

- Developer must verify origin correctly
- Counterexample: "domain-a.ru.attacker.com"
Example Attack

Attacker → Email → Victim User
Example Attack
Example Attack
Goal: Secure JavaScript Apps

- Hardening benign-but-buggy applications
- Fully automatic: no developer interaction
- Detection / defense in browser alone
  - No browser modifications or extensions
- Handle unknown vulnerabilities
Related Work

- CSV has been shown to be prevalent [1,2]
- 84 out of top 10,000 websites vulnerable [1]
- Bug finding systems
  - FLAX [2], Kudzu [3]
- Defense system
  - Changes to sites [1], standards [1]

ZigZag Overview

- Anomaly detection system preventing CSV attacks
- Instrumentation of client-side JavaScript
- Generates model of benign behavior
- Two phases
  - Learning of benign behavior
  - Prevent attacks
Learning Phase
Learning Phase
Learning Phase
Learning Phase
Learning Phase
Learning Phase
Learning Phase
Background: Program Invariants

● Assertions over variables at program points
  ○ \( x = y \)
  ○ \( x + 5 = y \)
  ○ \( x < y \)

● Dynamic detection
  ○ Statistical likelihood
  ○ DAIKON [1]

# Supported Invariants

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Invariants</th>
</tr>
</thead>
<tbody>
<tr>
<td>All</td>
<td>Types</td>
</tr>
<tr>
<td>Numbers</td>
<td>Equality, inequality, oneOf</td>
</tr>
<tr>
<td>String</td>
<td>Length, equality, oneOf, isPrintable, <strong>isJSON, isEmail, isURL, isNumber</strong></td>
</tr>
<tr>
<td>Boolean</td>
<td>Equality</td>
</tr>
<tr>
<td>Objects</td>
<td>All of the above for object properties</td>
</tr>
<tr>
<td>Functions</td>
<td><strong>Calling function</strong>, return value</td>
</tr>
</tbody>
</table>
Generated Invariants

- Function is only invoked from the global scope

- `typeof(v0) === 'object'`  
  `&& v0.origin === 'http://domain-a.ru'`

- `v0.data === $('#right_buttons').hide();`  
  `|| v0.data === 'calculator()'`
Enforcement Phase
Enforcement Phase
Enforcement Phase
Enforcement Phase
Enforcement Phase
Enforcement Phase
Enforcement Phase
Enforcement Details

Original

```javascript
function x(a, b) {
    c = a + b;
    return c;
}
```

Instrumented

```javascript
function x(a, b) {
    var callcounter = __calltrace(functionid, codeid, sessionid);
    c = a + b;
    return __exittrace(functionid, callcounter, subexitid, codeid, sessionid, c);
}
```
Enforcement Details

```javascript
__calltrace = function(functionid, codeid, sessionid) {
    var v0 = arguments.callee.caller.caller.arguments[0];
    var v1 = ...
    if ( functionid === 0 ) {
        __assert(typeof(v0) === 'object' && v0.origin === 'http://domain-a.ru' );
        __assert(v0.data === '%$('#right_buttons').hide();' ||
                v0.data === 'calculator()' );
        ...
    } else if ( functionid === 1 ) {
        ...
    }
    ...
    return __incCallCounter();
}
```
Deployment Options

- **Transparent proxy**
  - Deploy at gateway
  - Some latency
- **On-site rewriting**
  - One-time instrumentation
  - Protects all users
- **Invariant sharing**
  - Proxy setup
Challenges

● In-Browser web app runtime performance
  ○ Instrumentation of functions that are not security relevant

● Server-side code templates
  ○ New code can be generated for individual page visits
Targeted Instrumentation

● What functions are relevant to be instrumented?

● Lightweight static analysis to refine instrumentation
  ○ document, window, ...
  ○ XHR, eval, postMessage, ...

  Output: list of functions to be instrumented

● Benefits: increased performance at runtime
Server-side JavaScript Templates

- Parameterize code with:
  - Configuration
  - Username
  - Timestamps, etc.

- Programs differ, but AST is similar

- Singleton training sets: Anomaly detection not possible
Server-side JavaScript Templates

- Parameterize code with:
  - Configuration
  - User Name
  - Timestamps, etc.

- Programs differ, but AST is similar

- Singleton training sets: Anomaly detection not possible

Q: How to generate invariants over classes of programs?
Server-side JavaScript Templates

// Server-side JavaScript template

var state = {
    user: {{username}},
    session: {{sessionid}}
};

// Client-side JavaScript code after template instantiation

var state = {
    user: "UserX",
    session: 0
};

var state = {
    user: "UserY",
    session: 1
};

var state = {
    user: "UserZ",
    session: 2
};
Server-side JavaScript Templates

// Server-side JavaScript template

var state = {
    user: {{username}},
    session: {{sessionid}}
};

// Client-side JavaScript code after template instantiation

var state = {
    user: "UserX",
    session: 0
};

var state = {
    user: "UserY",
    session: 1
};

var state = {
    user: "UserZ",
    session: 2
};
Program Generalization

● When is code similar?
  ○ Values of primitives differ
  ○ Object properties differ

● AST can differ due to:
  ○ Object properties omitted
  ○ Order of properties

● Goal:
  ○ Structural comparison
  ○ Fast instrumentation
Structural Comparison

Uninstrumented program - extract abstract syntax tree
Structural Comparison

Detect program points for generalization

[Diagram of a tree structure with nodes and branches]

Michael Weissbacher, Northeastern University, Boston
Structural Comparison

Generalize: add placeholders
Structural Comparison

Generalize: add placeholders

Extract canonical string representation of AST for matching
Program Generalization

● Generalize program by:
  ○ Removing primitive values
  ○ Remove primitive object properties
  ○ Order object properties

● Instrument generalized version
Program Generalization (ctd.)

- When new program is detected as specialized version:
  - Differences from the template are detected
  - Generated patch-set is applied to instrumented version

- Result:
  - Anomaly detection extended from same to similar programs
  - Performance close to cached instrumentation
Evaluation: Security Benefits

- Four real-world vulnerable sites
- Synthetic webmail application

- Attacks caught
- Functionality retained
Evaluation: Performance

- Alexa Top 20
  - Median overhead: 2.01s (112%)
  - No false positives

- Microbenchmark: 0.66ms overhead
Contributions

- In-browser anomaly detection system for hardening against previously unknown CSV vulnerabilities
- Invariant patching: extending invariant detection to server-side templates
Conclusions

- CSV attack prevention through invariants
- Real-world obstacles
  - Server-side template
  - Function whitelisting
- ZigZag
  - Effective defense
  - Enforcement in browser only
  - Moderate performance overhead
Thank you for your attention

@mweissbacher