Spectator: Detection and Containment of JavaScript Worms

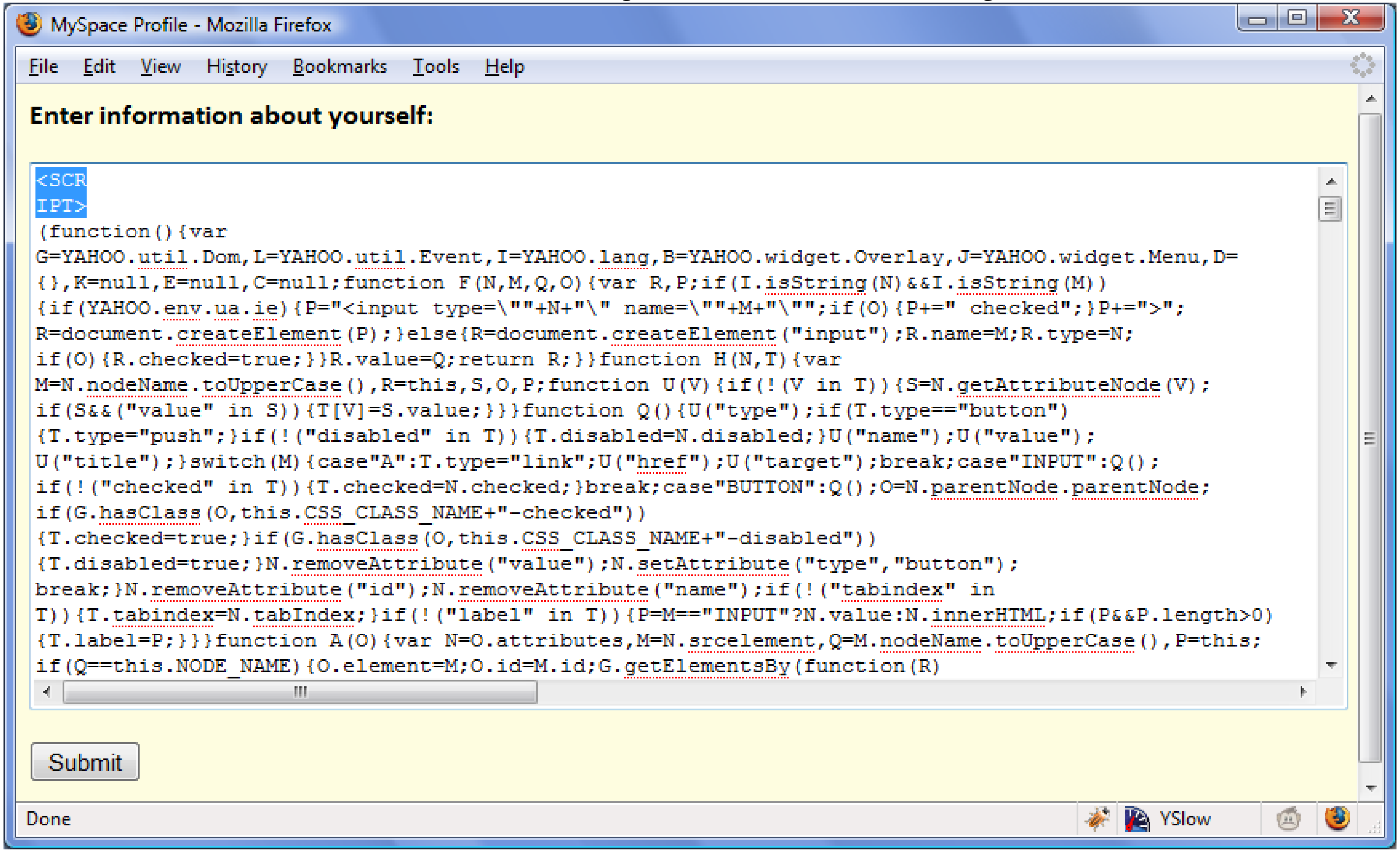
By Livshits & Cui

Presented by Colin

The Problem

- AJAX gives JS an environment nearly as flexible as a C/asm on a desktop OS
  - Buffer overruns allow asm code injection
  - Tainted string propagation allows JS code injection
- Now worms can propagate through JS as well

Example: Samy



One guy figures out how to embed Javascript in CSS, which MySpace doesn't filter

Samy (cont.)

- Visitors to his profile run the JS on page load
- The script “friends” the author, then adds the same source to their profile.
- Now anyone who visits that profile would also get infected, and so on…

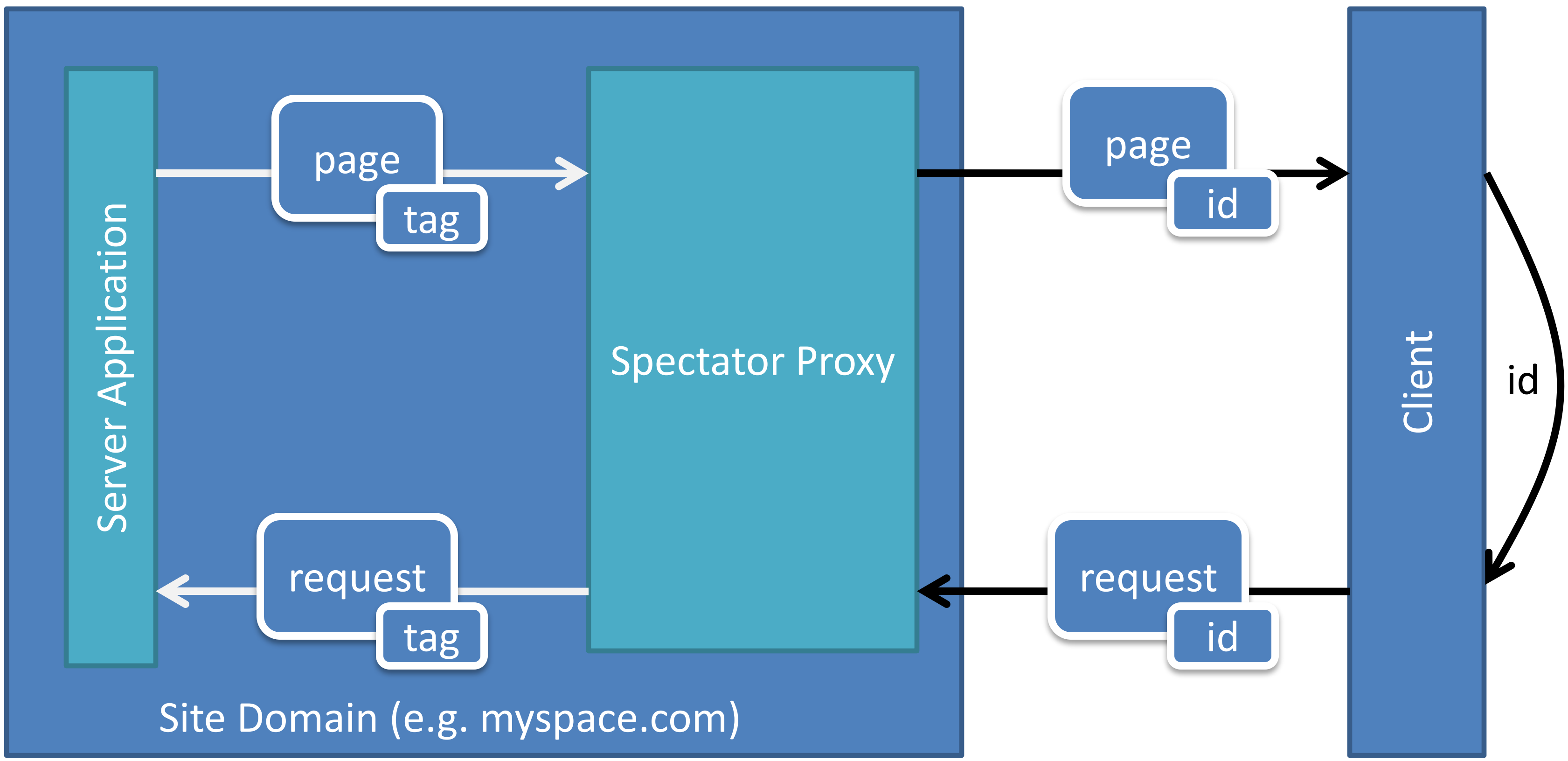
It Gets Worse…

- This could potentially work on a site like GMail...
- Windows Scripting Engine understands JS…
- Sophos lists over 380 JS worms
- All known static analyses for finding these bugs are either unsound, or sound for a narrow class of bugs, so we really can’t just find them all statically

Idea for a Solution

- Monitor the interactions of *many* users, and watch the propagation of information
  - If the same information propagates across, say 100 users, this is probably a worm.

Overall Design



Server-Side Tag Flow

- Server Interactions
  - Proxy tags requests containing HTML/JS
  - Proxy checks for tags in pages pulled from the server

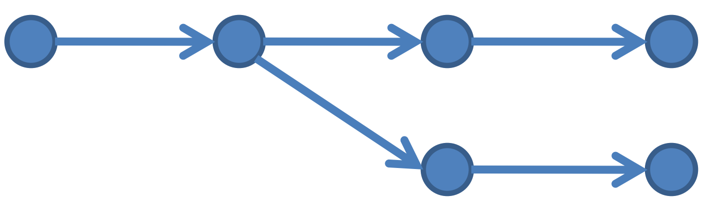
```
<div spectator_tag=134>
  <a onclick="javascript:...">...</a>
</div>
```

Client-Side Tag Flow

- Client Interactions
  - Proxy issues HTTP-only cookie w/ ID for the set of tags in the current page
  - Browser sends ID back to proxy w/ each request

Tracking Causality

- A tag present on a page is assumed to *cause* the subsequent request
- Consider a propagation graph:



Propagation Graphs

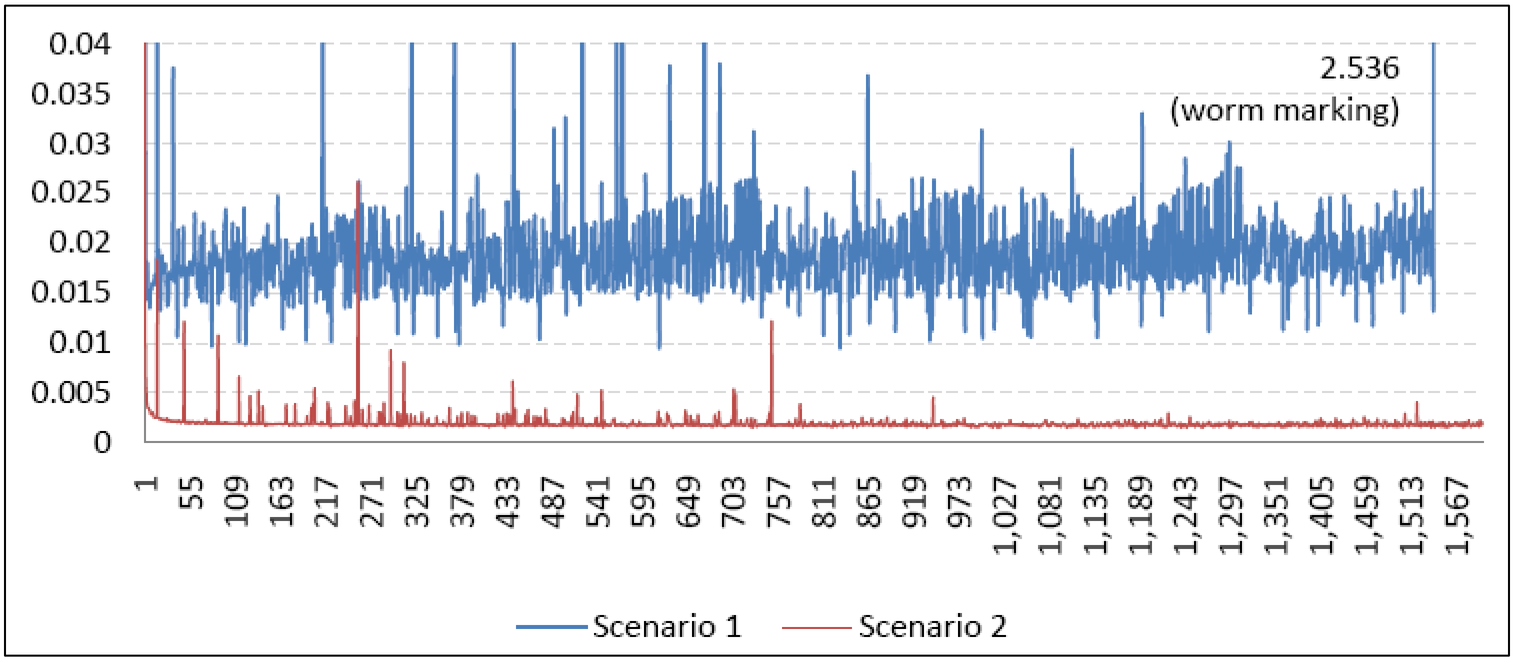
- Record propagation of tags on upload
- Track IPs along with tags
- Heuristic: If the # of unique IPs along a path exceeds a threshold *d*, flag a worm
- Accurately modeling the graph is exponential

	Accurate Graph	Approximate Graph
Time to insert	$O(2^n)$	O(1) on average
Space to track path length	O(n)	O(n)
Blocking futher propagation	O(n)	O(n)

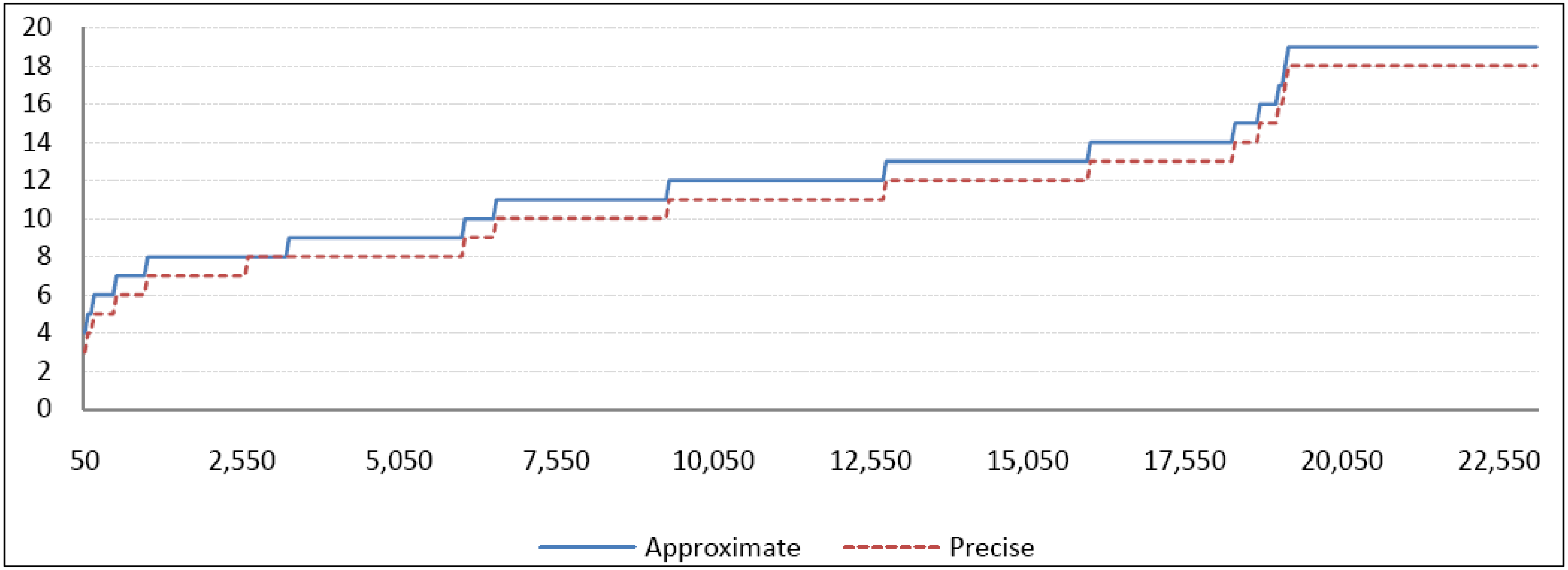
Simulations

- Used a MySpace clone to test scaling
- Three propagation models
  - Random
  - Linear
  - Biased
- Tested scalability of graph tracking

Graph Insertion Time



Graph Diameter



Proof-of-Concept Exploit

- Used AJAX blog
- Implemented a manual-propagation worm
- Spectator detected and stopped the worm

Discussion

- Where do false negatives come from? Can a worm trick Spectator by hiding propagation behind legitimate user activity?
- What assumptions does Spectator make about interactions of individual users (think about multiple windows, tabs…)
- Is this a good match for Gmail’s HTTPS-only connections?

Static Detection of Security Vulnerabilities in Scripting Languages

By Xie & Aiken

Presented by Colin

The Problem

- SQL Injection
- PHP makes it difficult to do a traditional static analysis
  - include
  - extract
  - dynamic typing
  - implicit casts everywhere
  - scoping & uninitialized variables

A Solution

- A 3-tier static analysis
  - Symbolic execution to summarize basic blocks
    - Well-chosen symbolic domain
  - Block summaries make function summaries
  - Function summaries build a program summary

Symbolic Execution for Basic Blocks

- Novel choice of symbolic values
  - Strings modeled as concatenations of literals and non-deterministic containment

    $<\beta_1,\ldots,\beta_n>$ where $\beta = \ldots \mid \text{contains}(\sigma) \mid \ldots$
  - Booleans include an ultra-lightweight use of dependent types:

    $\text{untaint}(\sigma_0,\sigma_1)$

Block Summaries

- E: must be sanitized on entry
- D: locations defined by the block
- F: value flow
- T: true if the block exits the program
- R: return value if not a termination block
- U: locations untainted by this block

Example Block & Summary

```
validate($q);
$r = db_query($q.$a);
return $r;
```

- E: {$a}
- D: {$r}
- F: {}
- T: false
- R: { _|_ }
- U: {$q}

Using Block Summaries

- Paper hand-waves with “well-known techniques”
  - Backward propagation of sanitization req.s
  - Forward propagation of sanitized values, returns, with intersection or union at join points
- Dealing with untaint:

if (<untaint(σ_0,σ_1)>) {
    <check with σ_1 sanitized>
} else {
    <check with σ_0 sanitized>
}

Function Summaries

- E: must be sanitized on entry
- R: values that may propagate to the return val
- S: values always sanitized by the function
- X: whether the function always exits the program

Example Function & Summary

```
function
runq($q, $a) {
  validate($q);
  $r =
  db_query($q.$a);
  return $r;
}
```

- E: {$a}
- R: contains($q, $a)
- S: {$q}
- X: false

Using Function Summaries

- Replace formal arguments with actual arguments in the summary
- Cut successors if the function always exits

Checking Main

```
function
runq($q, $a) {
  validate($q);
  $r =
  db_query($q.$a);
  return $r;
}

runq($q,$a);
```

- E: {$a}
- R: contains($q, $a)
- S: {$q}
- X: false

E is the set of unsanitized program inputs!

Evaluation

App (KLOC)	Errors	Bugs (FP)	Warnings
News Pro (6.5)	8	8 (0)	8
myBloggie (9.2)	16	16 (0)	23
PHP Webthings (38.3)	20	20 (0)	6
DCP Portal (121)	39	39 (0)	55
e107 (126)	16	16 (0)	23
Total	99	99 (0)	115

- Only errors were investigated, warnings may contain more bugs.
- Hand-waving on the vulnerability and bug verification details.

PHP Fusion

- Uses extract($_POST, EXTR_OVERWRITE)
- Allows exploits by adding extra POST parameters for variables uninitialized in the source
- Example: $new_pass is uninitialized

```
for ($i=0;$i<7;$i++)
    $new_pass .= chr(rand(97,122));
…
$result = dbquery("UPDATE ".$db_prefix."users
    SET user_password=md5('$new_pass')
    WHERE user_id=' ".$data['user_id']." ' ");
```

PHP Fusion

- Uses extract($_POST, EXTR_OVERWRITE)
- Allows exploits by adding extra POST parameters for variables uninitialized in the source
- Example: $new_pass is uninitialized

```
for ($i=0;$i<7;$i++)
    $new_pass .= chr(rand(97,122));
…
$result = dbquery(“UPDATE ”.$db_prefix.“users
    SET user_password=md5(‘$new_pass’)
    WHERE user_id=‘ ”.$data[‘user_id’].” ‘ “);
```

Exploit parameter:
&new_pass=abc%27%29%2cuser_level=%27103%27%2cuser_aim=%28%27

Produces $result:

```
UPDATE users SET user_password=md5(‘abc’), user_level=‘103’, user_aim=‘?????’)
        WHERE user_id=‘userid’
```

Comparing to PQL

Xie & Aiken (PHP)

- Tailored to PHP's built-in string concatenation
- Infers sanitization functions from a base set
- Handles relation between return values and sanitized values
- Unsound (specialized to strings and booleans)
- Effective, few FP
- Roughly, taint inference

Livshits & Lam (Java)

- Requires specifying the propagation relation
- Sanitizers must be omitted from derivation function
- Cannot handle sanitization checkers, only producers of new sanitized values
- Sound
- Effective, few FP
- Roughly, taint flow analysis

Discussion

- How much would need to change to track other sorts of properties?
- What makes this system unsound?
- Where exactly does this system lose precision?