XSSDS und noXSS

Server- und Browser-basierte XSS Erkennung

Martin Johns
University of Passau, ISL
martin.johns@uni-passau.de

Jeremias Reith
University of Hamburg, SVS
jr@noxss.org

OWASP
Frankfurt, 25.11.08

About us: The (no)XSS(DS) team

Martin Johns

- PhD candidat at Uni Passau

Jeremias Reith

- Master's student at Uni Hamburg

Björn Engelmann (bjoern@noxss.org)

- Former master's student at Uni Hamburg

Joachim Possega

- Professor at Uni Passau

Motivation

Cross-Site Scripting (XSS) is almost ubiquitous

Server-side:

- Noticing that your applications are vulnerable is hard
  - The server only sees character-streams
  - JavaScript is interpreted in the browser
  - Exploitation happens on the client-side

Client-side:

- As XSS is a client-side attack, the user should be able to protect himself
- Threats from JS exceed the scope of the attacked application
  - JavaScript malware

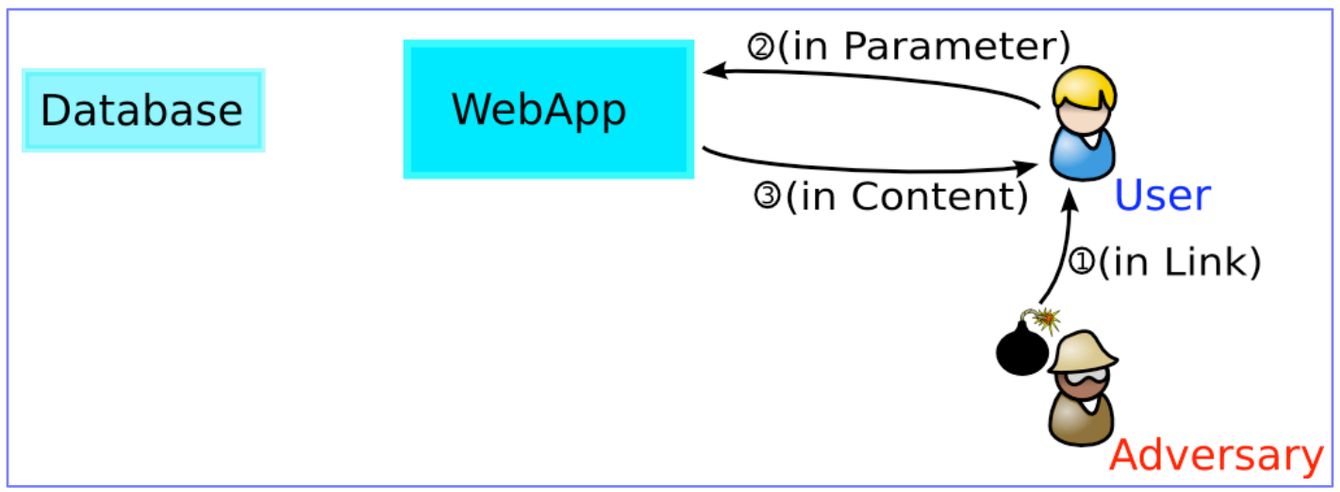
Our approaches: XSSDS (server) and noXSS (client)

Background: XSS

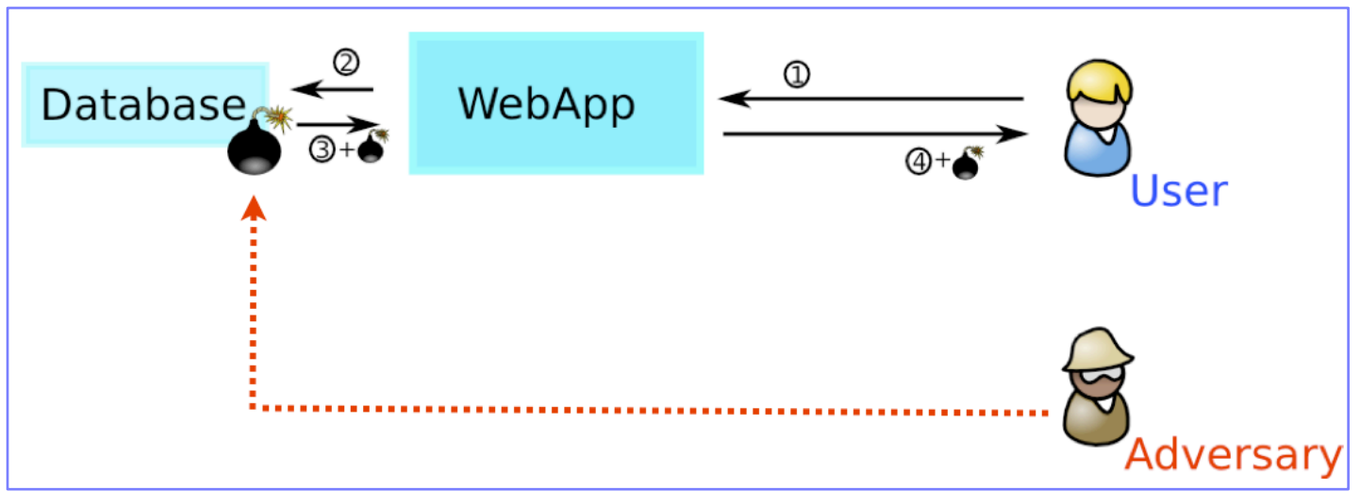
XSS == JavaScript injection

Two basic types:

- **Reflected XSS**



- **Stored XSS**



Observations

Web applications are (from the outside) rather straight forward

- Input: Parameters
- Output: HTML
- -> (semi-)functional relationship

Two basic observations

- There is a strong correlation between incoming parameters and and outgoing reflected XSS
- The set of legitimate JavaScripts of a given application is bounded

Based on these two observation we can design two detectors

Observation I

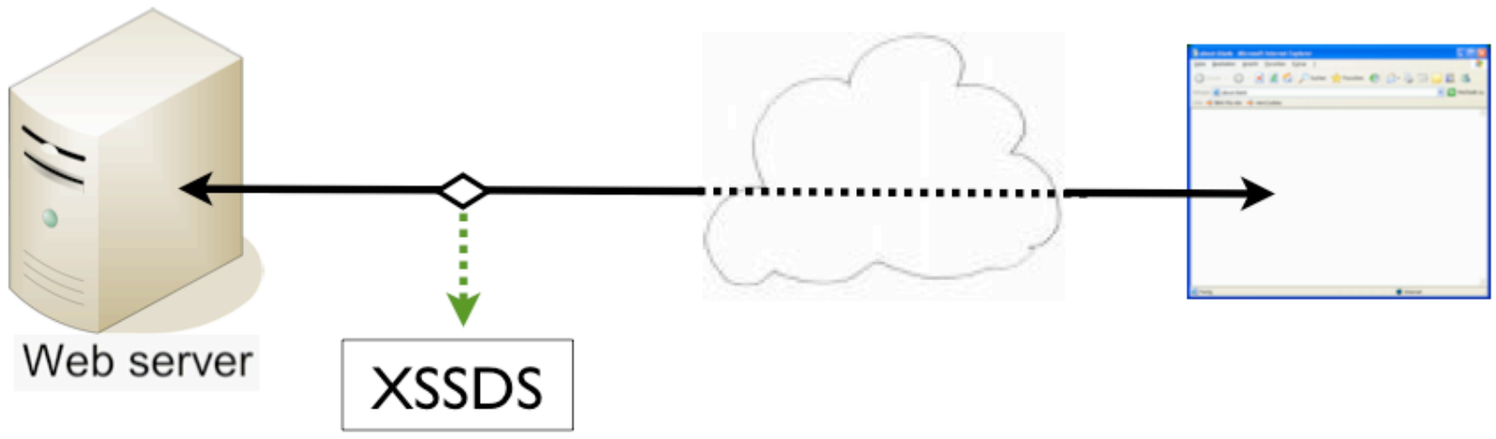
The set of legitimate JavaScripts of a given application is bounded

- The application's source code is finite
- Hence, there is a limited amount of source code responsible for creation of JavaScript code
- Such code can only produce a limited amount of script-variants
  - (modulo dynamic data-values)

Concluding detection method

- Watching the outgoing HTTP traffic to learn all legitimate scripts
- If we know all legal scripts, all unknown scripts have to be injected

Detector I



Training phase:

- Passively monitor HTTP traffic of regular application usage
  - E.g., during implementation, testing, and closed beta
- Parse resulting HTML, extract and store all JavaScripts
- Stop when no new scripts are encountered
  - Complete coverage is feasible, as we monitor complete application usage

Detection phase

- Continue to extract outgoing scripts
- Alert unknown scripts to the site's operator

Script types

Static scripts

- Always remain the same independent from parameters

Dynamic scripts

- Generated on the fly based on incoming (or server-side) data

Script types: Dynamic scripts

Data-dynamics (very common)

- Script content is static but data-values differ

```
echo "alert('hello " + $name + "!');";
```

- Solution: Replace data-values with generic placeholders

```
alert(STRING);
```

Code-repetition

- Script contains reoccurring code, very likely due to loops in the generating code

```
a[1] = "foo";
...
a[99] = "bar";
```

- Solution: Aim to learn all variants

Selective code omission

- Solution: Aim to learn all variants

Script types: External scripts

```
<script src="http://www.host.com/path/s.js">
```

In-domain

- Treat same as inline scripts

Cross-domain

- The actual script content is not seen by the detector
- Hence, instead learn a set of known external URLs
- ...and hope the external script-providers produce their scripts securely

Potential pitfall

```
eval(some_var);
```

Dynamic client-side code generation

- eval() of dynamic string constants
- Solution:
  - During script tokenizing all string constants are examined if they contain JavaScript code
- In such cases, these constants are treated as additional script-instances
- Drawback: Potential source for false positives

Implementation

Crucial:

- Reliable script extraction

Problem:

- Browser-specific lax and forgiving HTML parsing
- General purpose HTML parser libraries miss obfuscated injection methods

Solution

- Use the actual browser code
- Our prototype utilized the Firefox parser
- Production-level implementations should use more than one parsing engine

Evaluation

Data-set

- Vulnerable open-source application
- Real-life web apps

Test-vectors

- Existing issues
- Manually inserted scripts

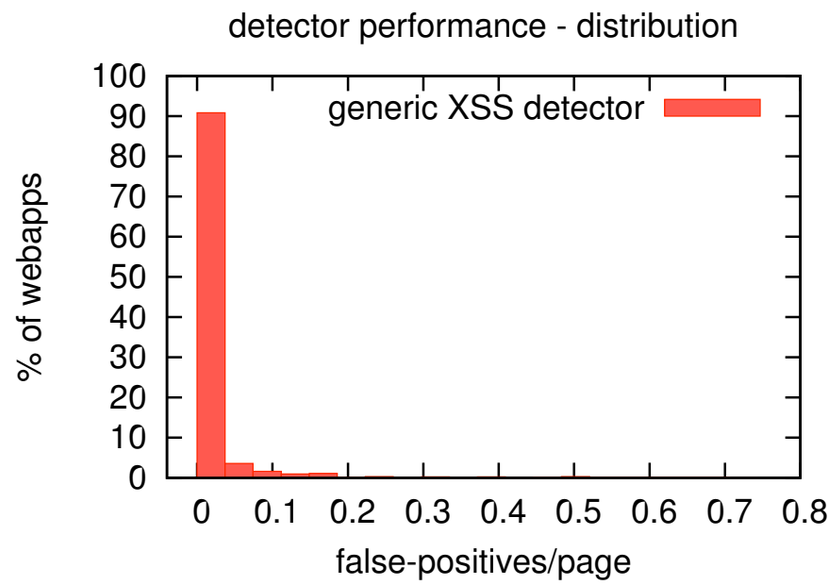
Methodology

- True vulns
  - Is the issue reported?
- False positves
  - k-fold cross-validation

Results

Detection rate

- All issues were reported
- This results in a false negative rate of 0

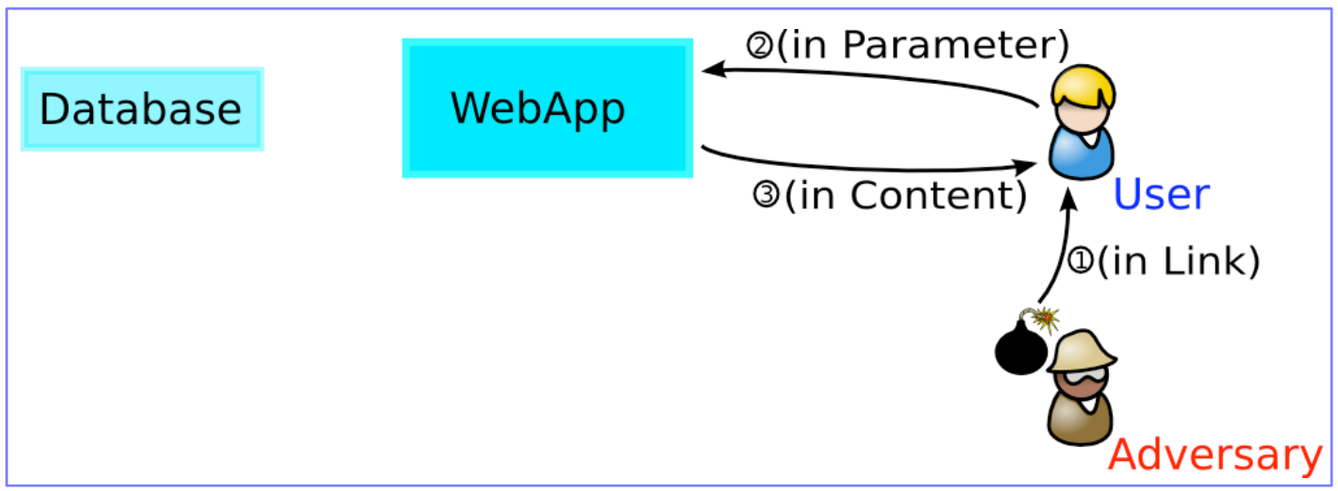


False positives

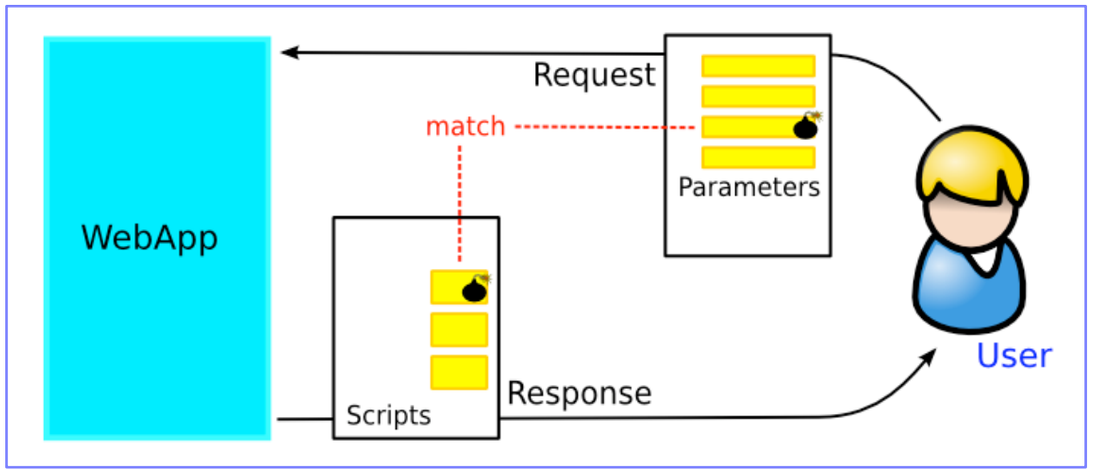
- 80% of the tested applications exposed no false positives
- The remaining 20% caused a varying amount of false positives
  - The majority of these issues was due to non-trivial dynamic code-generation which is not jet handled by our detector
  - E.g., dynamic generation of variable-names
  - In most cased easily fixed by customization

Observation II

There is a strong correlation between incoming parameters and and outgoing reflected XSS

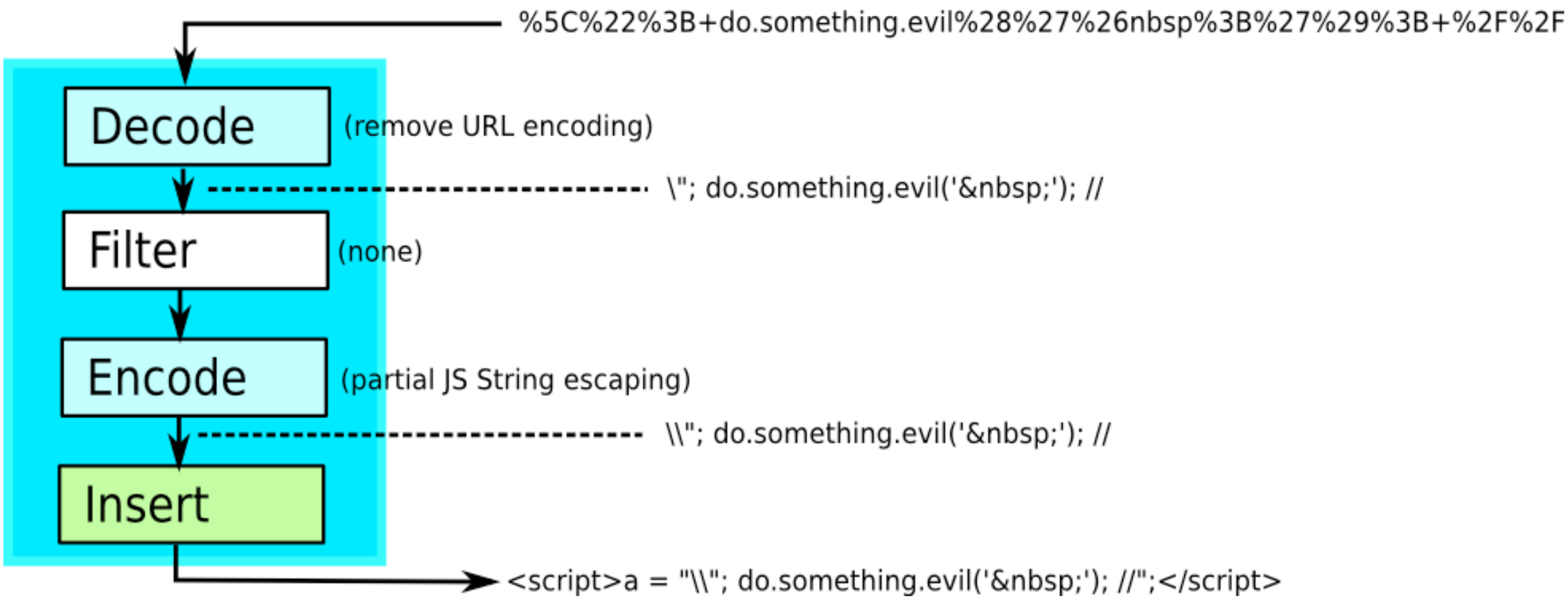


By matching incoming parameters against outgoing scripts, reflected XSS attacks should be detectable



Problem: (De|En)coding

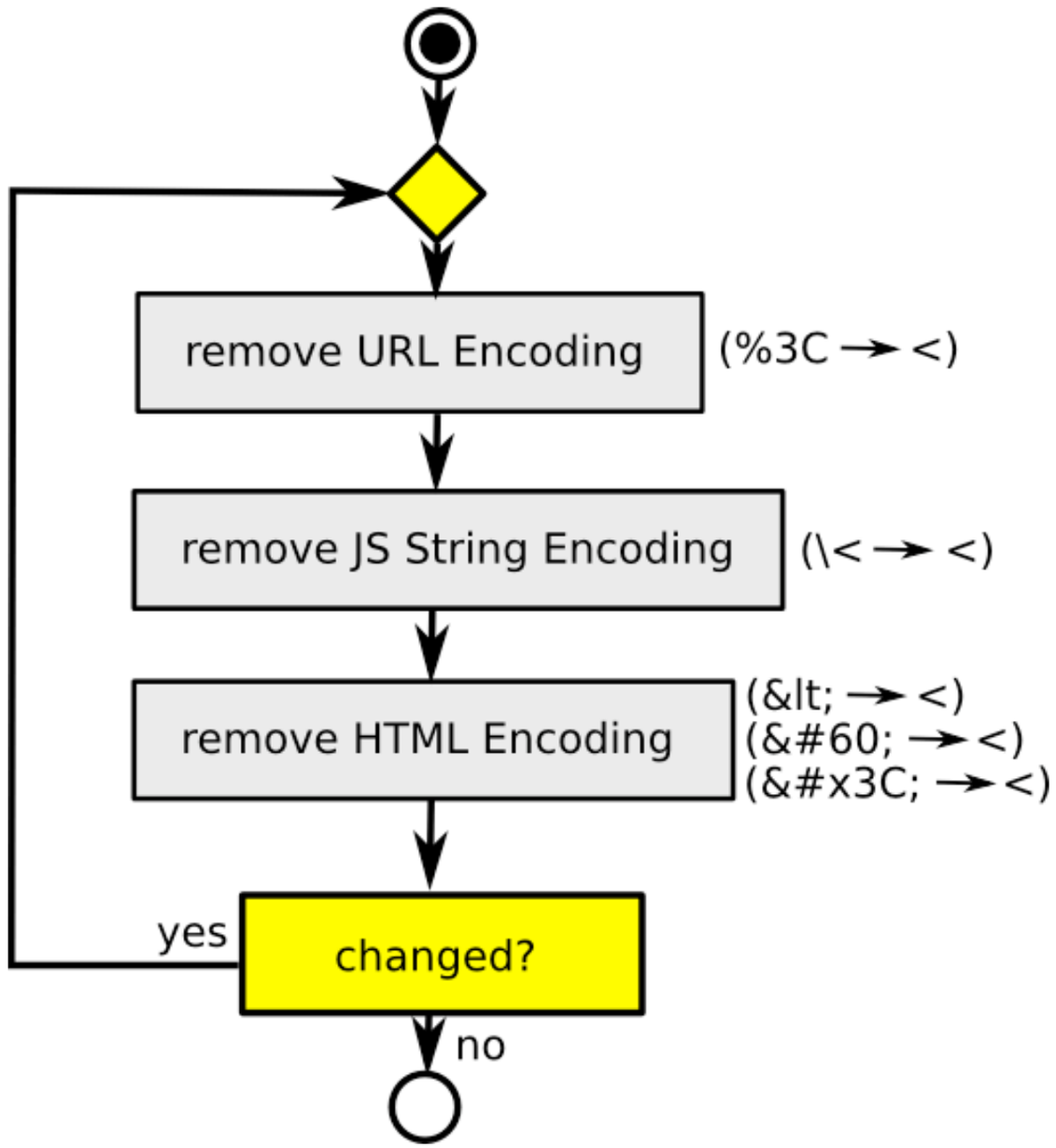
Incoming data is transformed during processing



--> Dumb matching on a character level is infeasible

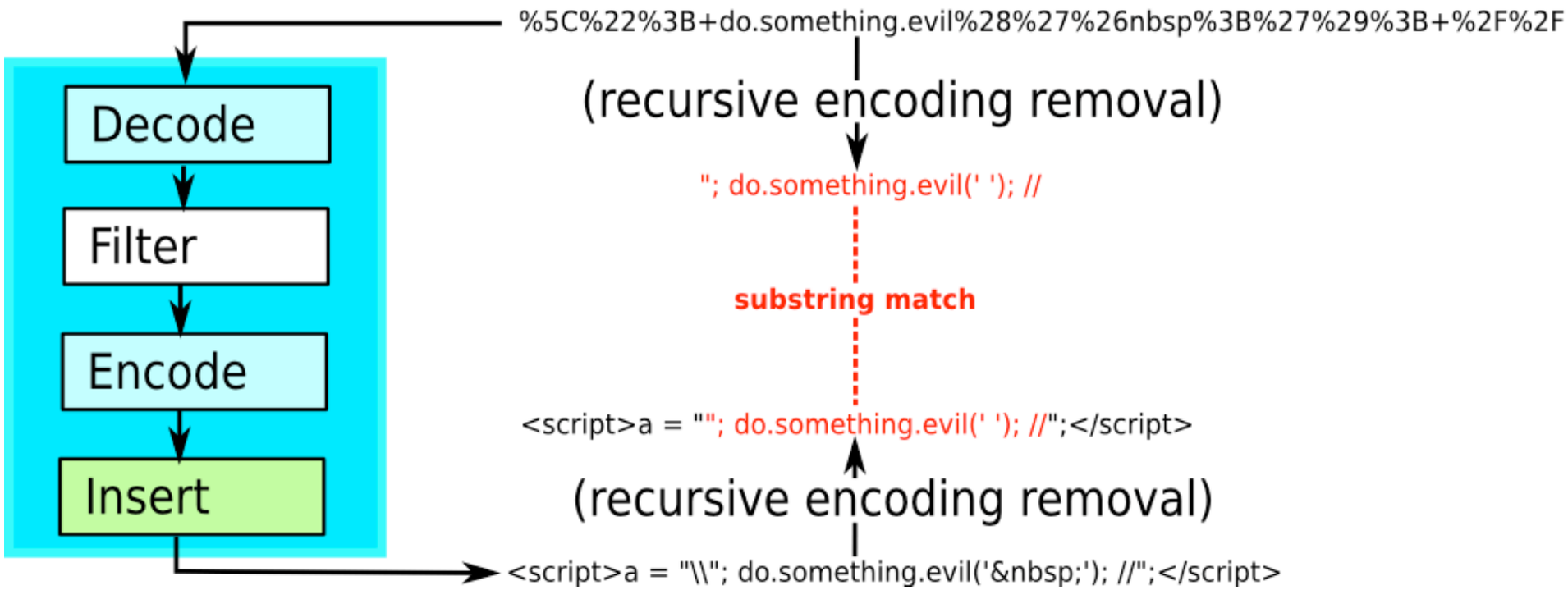
Solution

Applying recursive encoding removal on both parameters and scripts



Solution

Applying recursive encoding removal on both parameters and scripts



Remaining problem

- **If we have to deal with removal filters, further obstacles occur**

Detector II

Implementation of the outlined detection approach as server-side detector

- For details and results see the paper

Instead, we will talk about applying this technique within the browser

The Idea

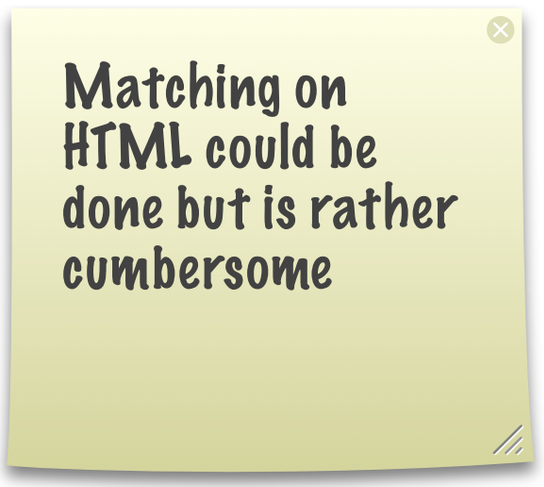
- Firefox extension for client side XSS detection
  - Usable with official Firefox (i.e. no Patching required)
  - Allows limitation to Firefox specific vectors
- Request/response matching from the XSSDS
  - Should have a lower false positive rate than classical approaches
  - More manageable than pattern based approaches

```
new RegExp(
  '(?:[\\w$\\u0080-\\uFFFF\\]][\\s\\S]*[\\(\\[\\.][\\s\\S]*(?:\\([\\s\\S]*\\)|=)|(?:' +
  fuzzify('eval|open|alert|confirm|prompt|set(?:Timeout|Interval)|[fF]unction') +
  ')[\\s\\S]*\\(|(?:' + fuzzify('setter|location') + ')[\\s\\S]*=)');

s.match(/\b(?:open|eval|set(?:Timeout|Interval)|[fF]unction|with|\[[^\]]*\w[^\]]*\]|
split|replace|toString|substr(?:ing)?|Image|fromCharCode|toLowerCase|unescape|
decodeURI(?:Component)?|atob|btoa|\${1,2})\s*(?:\/\*[\s\S]*?)?\([\s\S]*\)/);
```

Request/Response Matching

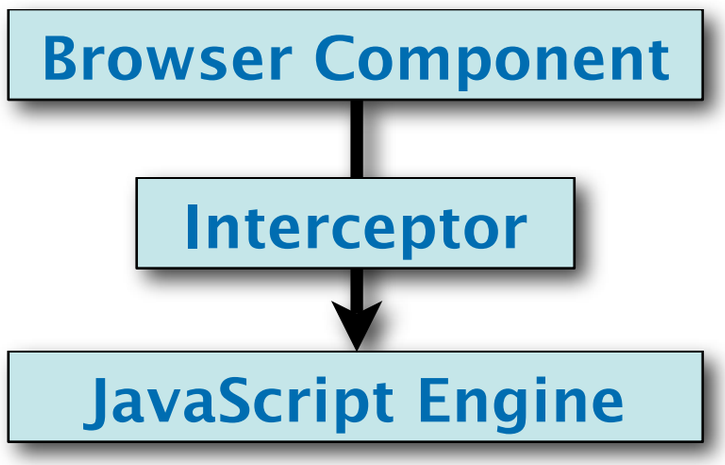
- On every request relevant request data is matched against extracted code
- A match of a given length is treated as a potential XSS attempt

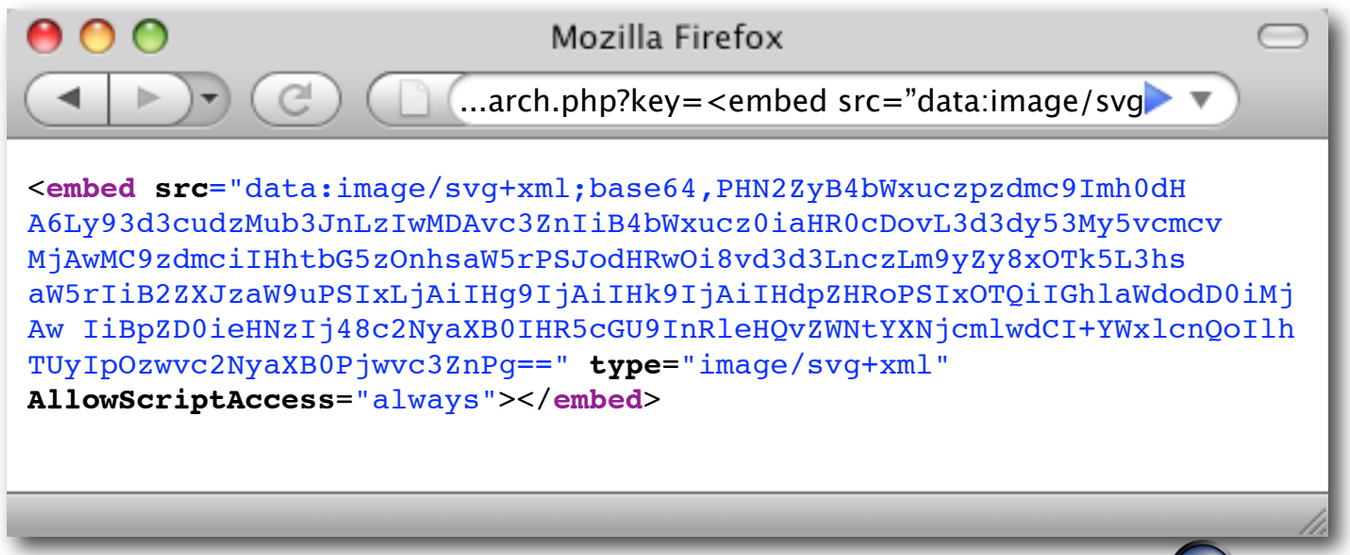
- Matching is applied to code only



JavaScript Interception

- JavaScript code extraction is not easy
- We will miss any code not directly embedded within the web page
- Hook into the interpreter and intercept any invocation of JavaScript

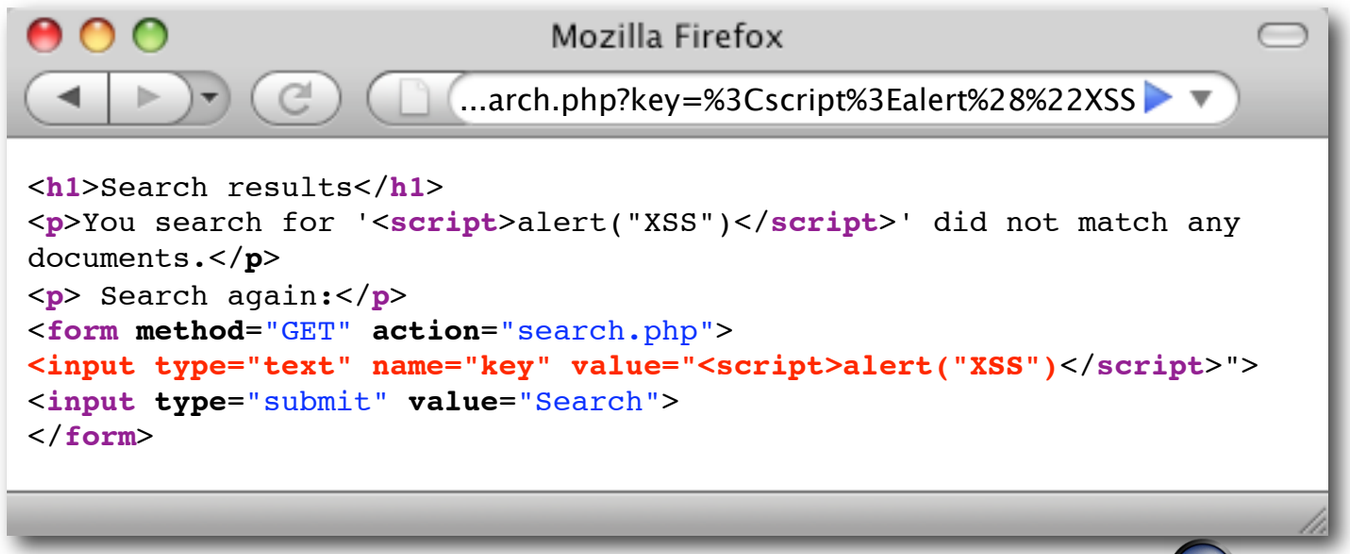




Decoding and the Mirror

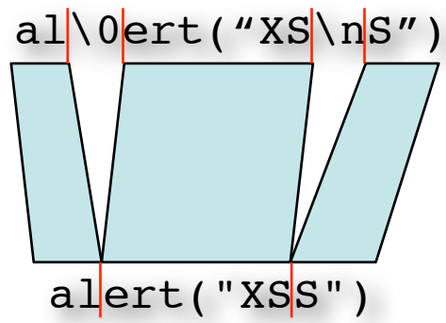
- Reflection's origin may be blurred
- Transform input in the same way the web application did?
  - Redo URL decoding and character set conversion
  - Handle other transformations

```
alert%28%22XSS%22%29

alert("XSS")

al<script>ert("XSS")   al\nert("X\0SS")
```



Subsequence Matching

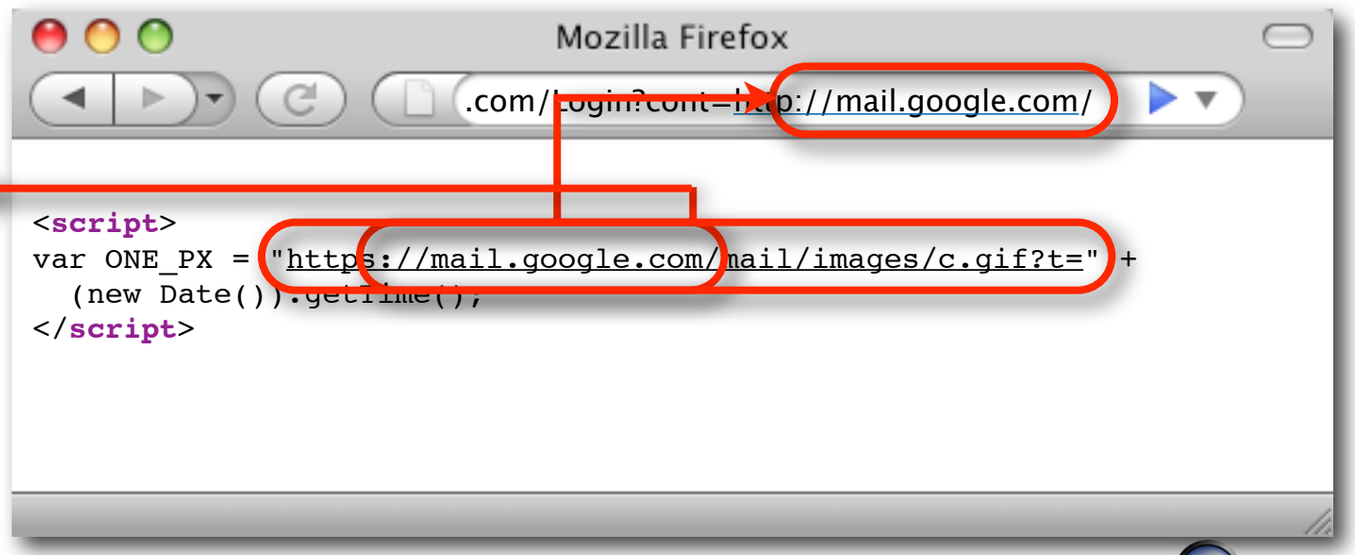
- A web application might insert or remove arbitrary characters
- Matching is done with an ALCS (All substrings longest common subsequence) variant
- Algorithm is using suffix trees



Tokenization

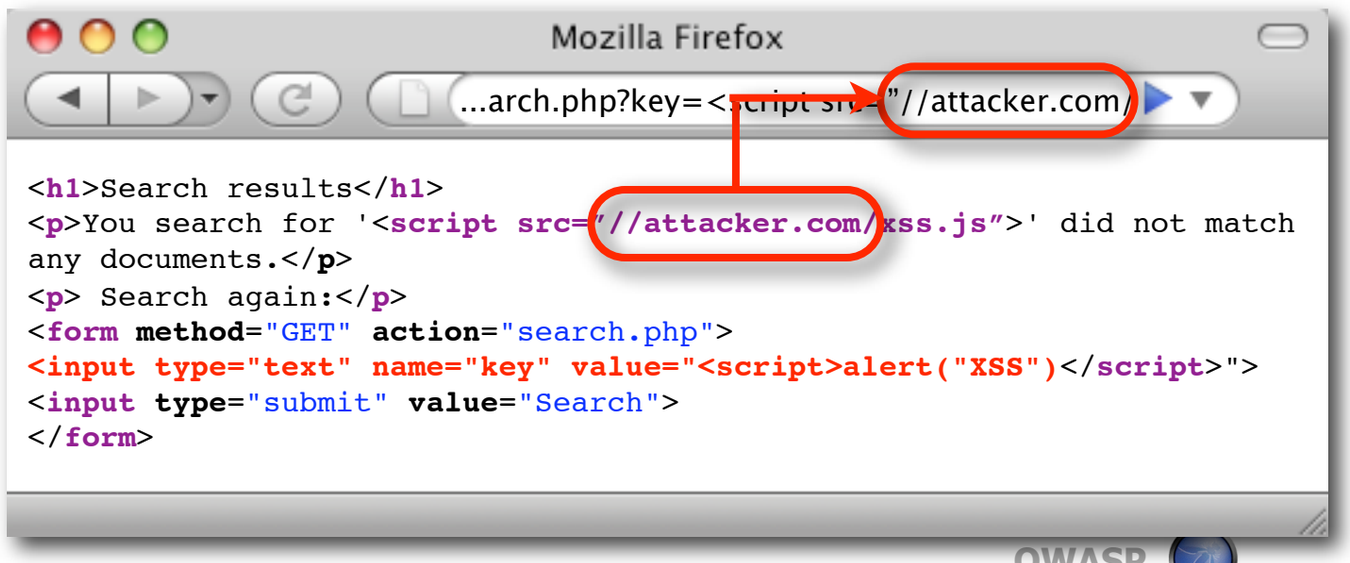
- Some matches in JavaScript code may be legitimate
- Count the number the JavaScript tokens a match consists of
- Matches spanning more than 2 tokens are considered harmful

```
TOK_VAR TOK_NAME TOK_ASSIGN TOK_STRING
TOK_PLUS TOK_LP TOK_NEW TOK_NAME TOK_LP
TOK_RP TOK_RP TOK_DOT TOK_LP TOK_RP
TOK_SEMI
```



Script file injection

- There is one case we have to cover in the markup realm
- The URL of included scripts via <script src=”...”> might be manipulated
- We will check the prefix of the URL



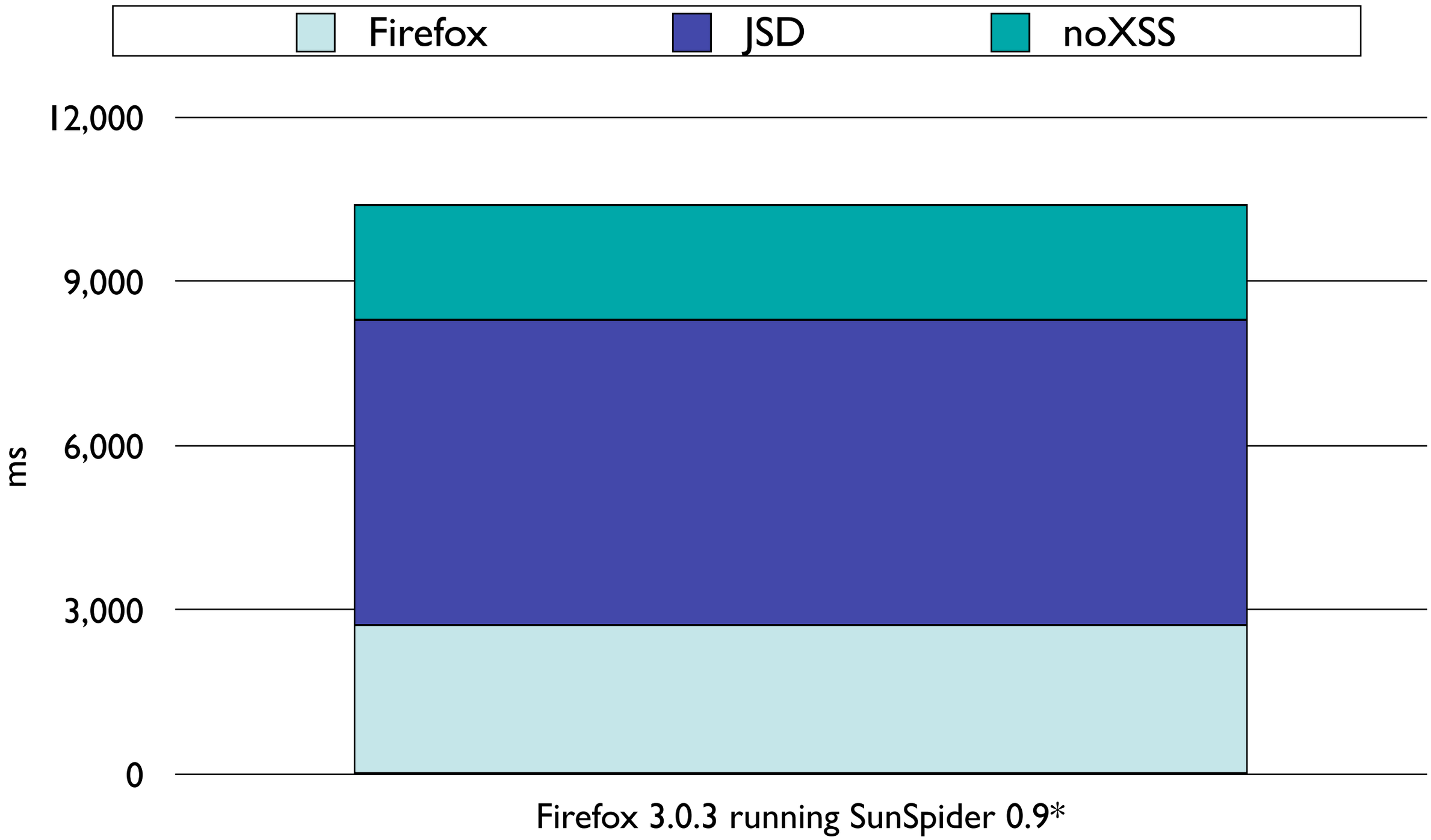
Cross Site Data Tainting

- Sometimes a payload is stored with session data on the server
- It might be inserted in a subsequent request
- We will taint any data passed across domains and check them in addition to current request data

Implementation - noXSS

- Normal Firefox extension
  - With binary components
- Uses JSD to intercept JavaScript
- Embedded SpiderMonkey is used for tokenization
- Uses exact substring matching at the moment
- Available on noXSS.org

noXSS Performance



Firefox 3.0.3 running SunSpider 0.9*

*Dual Xeon 5150 (4x 2.66 GHz)

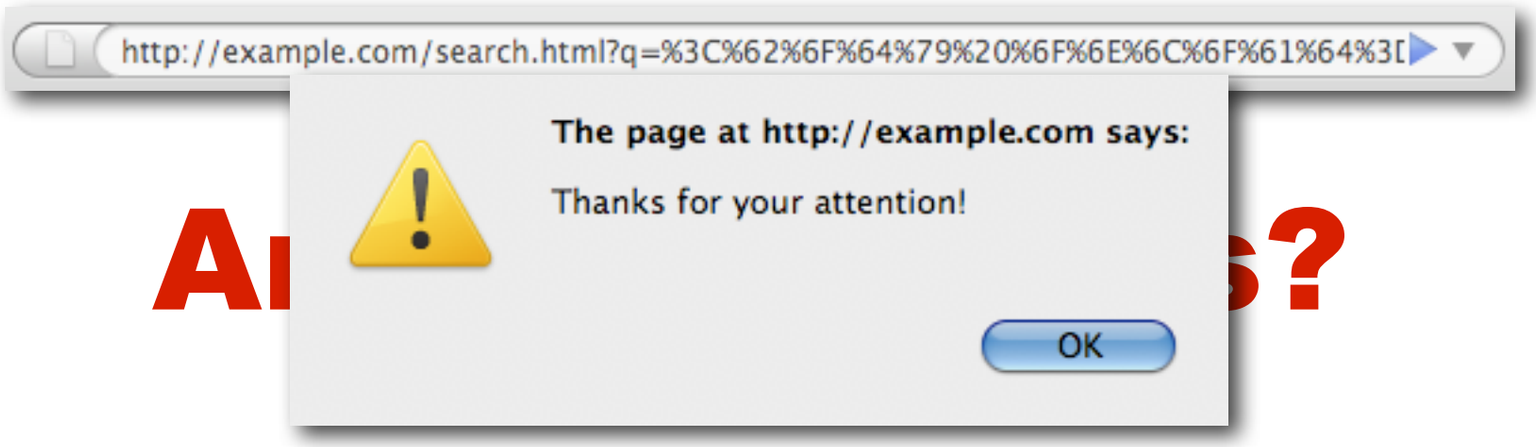
Evaluation

- Public evaluation via addons.mozilla.org
- ~65 average daily users over nearly two months
- Two classes of false positives
  - Script file injection (host name also in URL)
  - Multiple JavaScript keywords in URL
    - http://osvdb.org/search?request=document.write
    - https://developer.mozilla.org/en/DOM/document.getElementById

Future Work

- Incorporate interceptor API into Firefox
- Add public parser API to SpiderMonkey
- Implement a fast inexact matching algorithm
- Analysis of matched tokens for false positive reduction
- Better handling of script file injections
- Handling of repeated dynamic code generation (e.g. via setInterval())

The End



Any questions?