25 Million Flows Later – Large-scale Detection of DOM-based XSS

Published at ACM CCS 2013
Sebastian Lekies, Ben Stock, **Martin Johns**

Me, myself and I

Dr. Martin Johns

- Background in software engineering
- Academic work on Software and Web security at the Universities of Hamburg and Passau
- PhD on Web Security (with special focus on XSS)
- Since 2009: SAP Research in Karlsruhe
  - Scientific lead and coordinator of the EU FP7 project WebSand
  - Head of the WebSec research group at SAP

Agenda

- XSS Basics
- Implementation of a taint-aware browsing engine
- Large-scale Measurement of suspicious flows
- Verifying vulnerabilities
- Conclusion

Cross-Site Scripting

- Execution of attacker-controlled code on the client
- Three kinds:
  - Persistent XSS: stored in a database (guestbook, news comments, ..)
  - Reflected XSS: user-provided data echoed back into the page (search forms, ..)

  Server side

  - DOM-based XSS: using data coming from the Document Object Model „Tree“ (DOM)
    - may also be URL, also cookies, ...

  Client side

Cross-Site Scripting: problem statement

- **Main problem:** attacker‘s content ends in document and is not filtered/encoded
  - common for server- and client-side flaws
- Flow of data: from attacker-controllable *source* to security-sensitive *sink*
- Our Focus: client side JavaScript code
  - **Sources**: e.g. the URL
  - **Sinks**: e.g. document.write

What does a DOM-based vuln. look like?

```
document.write("<img src='//adve.rt/ise?hash=" + location.hash.slice(1)+ "' />");
```

- Intended functionality:
  - http://example.org/#mypage
  - `<img src='//adve.rt/ise?hash=mypage'/>`
- Exploiting the vuln:
  - `http://example.org/#'/><script>alert(1)</script>`
  - `<img src='//adve.rt/ise?hash='/>`

    `<script>alert(1)</script>`

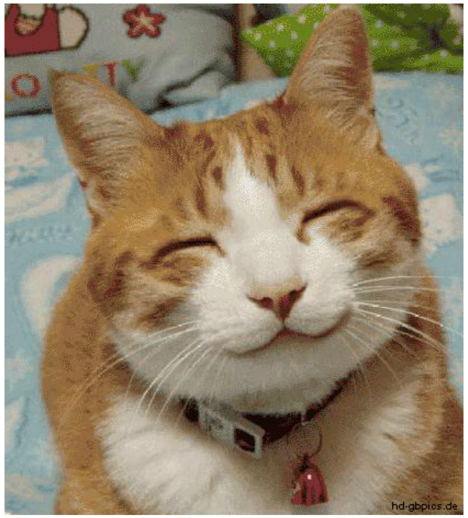
    `'/>`

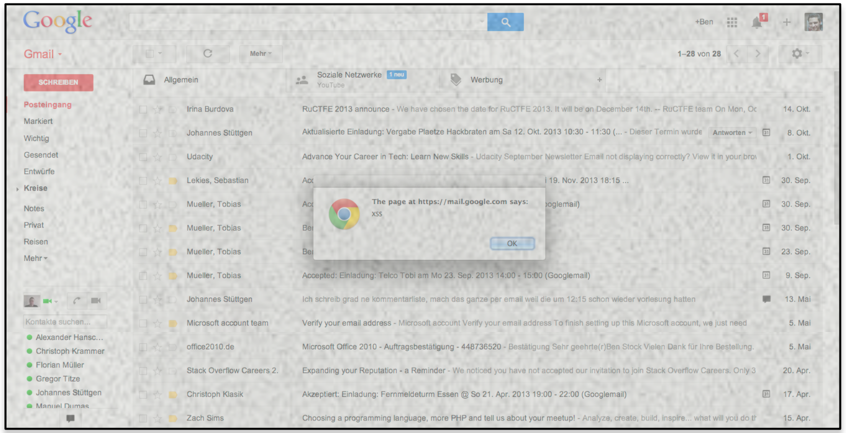
How does the attacker exploit this?

a. Send a crafted link to the victim

b. Embed vulnerable page with payload into his own page

http://kittenpics.org





Source: http://www.hd-gbpics.de/gbbilder/katzen/katzen2.jpg

How to analyze vulnerabilities?

- Static analysis?
  - tricky, due to very dynamic nature of JS
- Manual code audit?
  - minification ➔ look at Google Maps JavaScript code...
  - not large-scale..

- **Our approach: dynamic analysis**

Our contribution

- Large-scale analysis of DOMXSS vulnerabilities on the Web
  - Automated detecting of suspicious flows
  - Automated validation of vulnerabilities
- Key components
  - Taint-aware browsing engine
  - Crawling infrastructure
  - Context-specific exploit generator
  - Exploit verification using the crawler

Building a taint-aware browsing engine to find suspicious flows

Our approach: use dynamic taint-tracking

- **Taint-Tracking**: Track the flow of data from source to sink
  - Implemented into a real browser (Chromium with V8 JS engine)
  - Implements state-of-the-art APIs
  - Covers edge cases (at least for that browser)
- **Requirements**
  - Taint all relevent values / propagate taints
  - Report all sinks accesses
  - be as precise as possible
    - byte-level tainting

Representing sources

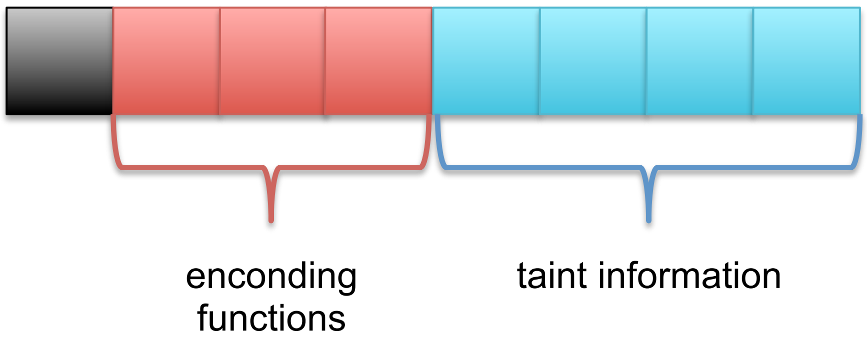
- In terms of DOMXSS, we have **14** sources
- additionally, **three** relevant, built-in encoding functions
  - escape, encodeURI and encodeURIComponent
  - .. **may** prevent XSS vulnerabilites if **used properly**
- Goal: store *source* + *bitmask of encoding functions* for each character

Representing sources (cntd)

- 14 sources ➔ **4** bits sufficient
- 3 relevant built-in functions ➔ **3** bits sufficient

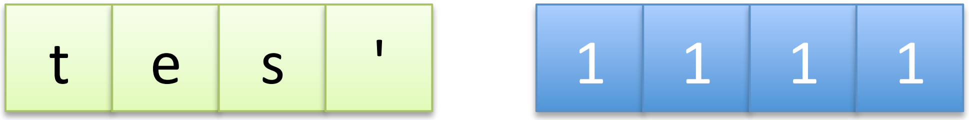
7 bits < 1 byte

- ➔ 1 Byte sufficient to store source + encoding functions
  - encoding functions and counterparts set/unset bits

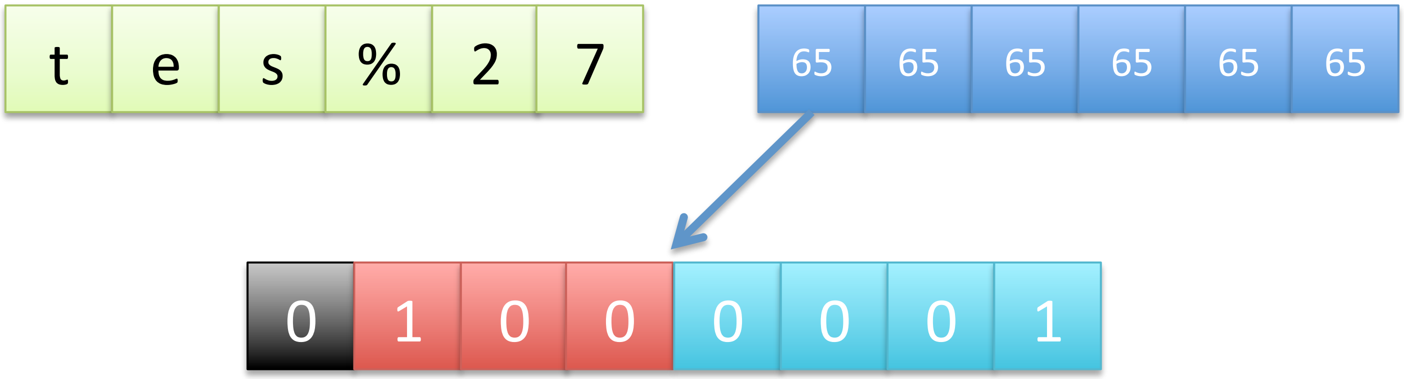


Marking strings and propagating taint

- Each source API (e.g. URL or cookie) attaches taint bytes
  - identifing the source of a char
  - `var x = location.hash.slice(1);`



  - `x = escape(x);`



Necessary code changes

- **V8 Strings must be taint-aware**
  - **String-modifying functions**
    - substring, appending, splitting, ..
  - **Regular Expressions**
    - extracting, replacing
  - ...
- Also: **WebKit** strings must be taint-aware

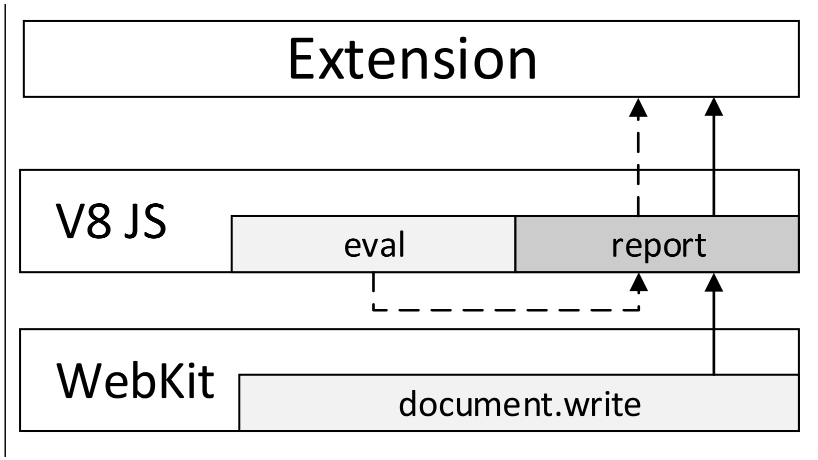
```
document.title = location.hash;
document.write(document.title);
```

  - ➔ Conversion from V8 to WebKit string must propagate taint

- For details on implementation please refer to the paper

Detecting sink access

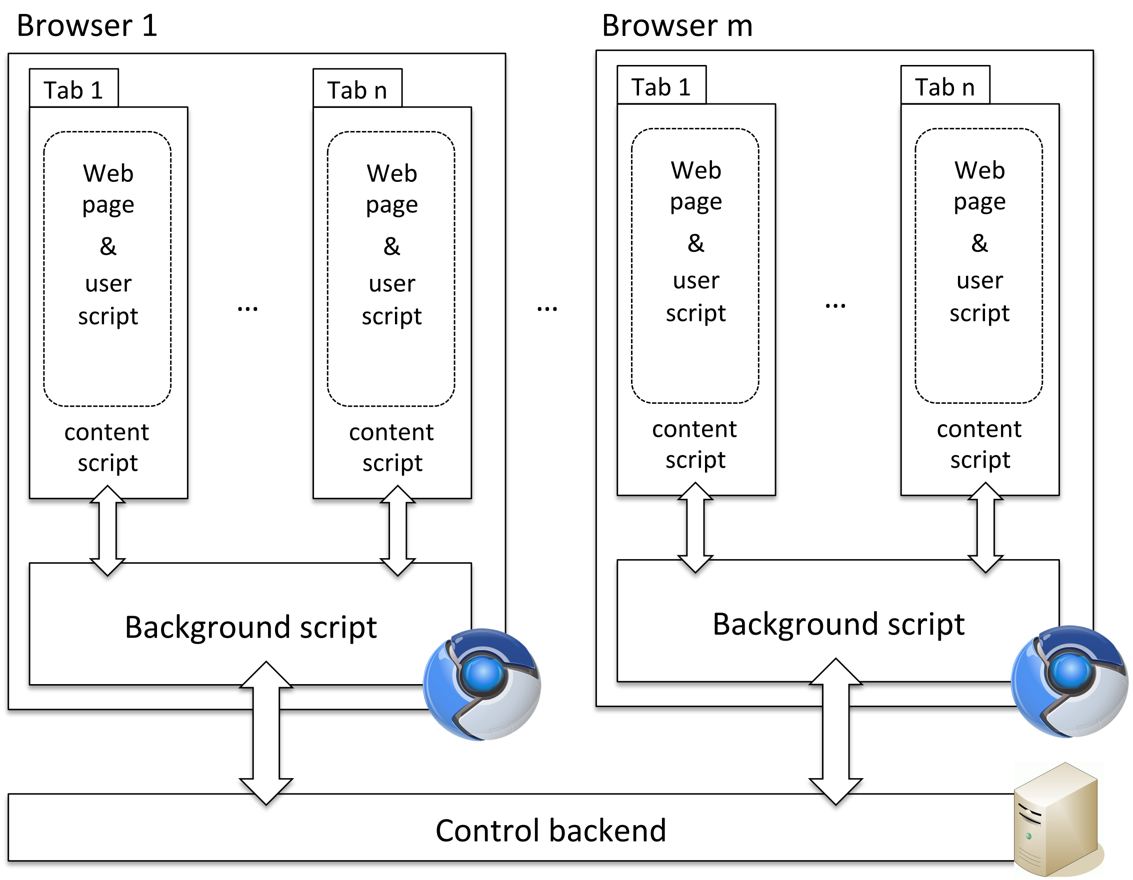
- **All relevant sinks patched to report flow and details**
  - such as text, taint information, source code location
- **We built a Chrome extension to handle reporting**
  - keep core changes as small as possible
  - repack information in JavaScript
  - stub function directly inside V8



Empirical study on suspicious flows

Crawling the Web (at University scale)

- Crawler infrastructure constisting of
  - modified, taint-aware browsing engine
  - browser extension to direct the engine
  - Dispatching and reporting backend
- In total, we ran 6 machines



Empirical study

- **Shallow crawl of Alexa Top 5000 Web Sites**
  - Main page + first level of links
  - **504,275** URLs scanned in roughly 5 days
    - on average containing ~8,64 frames
  - total of **4,358,031** analyzed documents

- **Step 1: Flow detection**
  - **24,474,306** data flows from user input to security-sensitive sinks
    - „data flow“ defined as a piece of data from a source flowing to the sink
      - NOT actual sink access
    - roughly 3 different flows per sink access

Recorded flows

Sources

Sinks

	URL	Cookie	Referrer	window.name	postMessage	WebStorage	Total
HTML	1,356,796	1,535,299	240,341	35,446	35,103	16,387	3,219,392
JavaScript	22,962	359,962	511	617,743	448,311	279,383	1,728,872
URL	3,798,228	2,556,709	313,617	83,218	18,919	28,052	6,798,743
Cookie	220,300	10,227,050	25,062	1,328,634	2,554	5,618	11,809,218
post Message	451,170	77,202	696	45,220	11,053	117,575	702,916
Web Storage	41,739	65,772	1,586	434	194	105,440	215,165
Total	5,891,195	14,821,994	581,813	2,110,715	516,134	552,455	24,474,306
Filters	64,78%	52,81%	83,99%	57,69%	**1,57%**	30,31%	

Context-Sensitive Generation of Cross-Site Scripting Payloads

Motivation

- Current Situation:
  - Taint-tracking engine delivers suspicious flows
  - Suspicious flow != Vulnerability
- Why may suspicious flows not be exploitable?
  - e.g. custom filter, validation or encoding function

```
<script>
  if (/^[a-z][0-9]+$/.test(location.hash.slice(1)) {
    document.write(location.hash.slice(1));
  }
</script>
```

- Validation needed: **working exploit**

Anatomy of an XSS Exploit

- Cross-Site Scripting exploits are context-specific:
  - HTML Context
    - Vulnerability:

```
document.write("<input value='"
        + location.hash.slice(1) + "'>");
```

    - Exploit:

```
'><script>alert(1)</script><textarea>
```

  - JavaScript Context
    - Vulnerability:

```
eval("var x = '" + location.hash + "'");
```

    - Exploit:

```
'; alert(1); //
```

  - URL Context
    - Vulnerability:

```
var frame=document.createElement("iframe");
frame.src=location.hash.slice(1) + "/test.html";
```

    - Exploit:

```
javascript:alert(1); //
```

Anatomy of an XSS Exploit

Break-out Sequence *Payload* *Break-in / Comment Sequence*

- Context-Sensitivity
  - Breakout-Sequence: Highly context sensitive (generation is difficult)
  - Payload: Not context sensitive (arbitrary JavaScript code)
  - Comment Sequence: Very easy to generate (choose from a handful of options)

Breaking out of JavaScript contexts

- JavaScript Context

```
<script>
  var code = 'function test(){'
             + 'var x = "' + location.href + '";'
             //inside function test
             + 'doSomething(x);'
           + '}';
  //top level
  eval(code);
</script>
```

- Visiting http://example.org/test.html

```
function test() {
  var x = "http://example.org/test.html";
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^
                  location.href
  doSomething(x);
}
```

Syntax tree to working exploit

```
function test() {
  var x = "http://example.org/";

  doSomething(x);
}
```

```
FunctionDeclaration
  Identifier : test
  FunctionConstructor
    Identifier : test
    Block
      Declaration
        Identifier : x
        StringLiteral : "http://example.org"
    ExpressionStmt
      SpecialOperation : FUNCTION_CALL
        Reference
          Identifier : doSomething
```

Tainted value aka injection point

- Two options here:
  - break out of string
  - break out of function definition
- Latter is more reliable
  - function test not necessarily called automatically on „normal“ execution

Generating a valid exploit

```
FunctionDeclaration
  Identifier : test
  FunctionConstructor
    Identifier : test
    Block                                          → }
      Declaration                                  → ;
        Identifier : x
        StringLiteral : "http://example.org"       → “
    ExpressionStmt
      SpecialOperation : FUNCTION_CALL
        Reference
          Identifier : doSomething
```

- Traverse the AST upwards and “end” the branches
  - Breakout Sequence: “;}
- Put it together:
  - Payload: __reportingFunction__(1234);
  - Comment: //
  - **Exploit: ";}__reportingFunction__(1234);//**
  - Visit: http://example.org/#";}__reportFunction__(1234);//

Validating vulnerabilities

- First focus: easy to exploit vulnerabilities
  - *Sources*: location and referrer
  - *Sinks*: direct execution sinks
    - HTML sinks (document.write, innerHTML ,...)
    - JavaScript sinks (eval, ...)
  - Only unencoded strings

- Not in the focus (yet): second-order vulnerabilities
  - to cookie and from cookie to eval
  - ...

Empirical study

- Step 2: Flow reduction
  - Only JavaScript and HTML sinks: 24,474,306 ➔ 4,948,264
  - Only directly controllable sources: 4,948,264 ➔ 1,825,598
  - Only unfiltered flows: 1,825,598 ➔ **313,794**

- Step 3: Precise exploit generation and validation
  - Generated a total of **181,238** unique test cases
  - rest were duplicates (same URL and payload)
    - basically same vuln twice in same page

Verifying vulnerabilities

- Step 3: Exploit validation
  - **181,238** unique test cases were executed
  - **69,987** Exploits were executed successfully

- Step 4: Further analysis
  - **8,163** unique vulnerabilities
  - …affecting **701** domains
  - …of all loaded frames (i.e. also from outside Top 5000)

  - **6,167** unique vulnerabilities
  - …affecting **480** Alexa top 5000 domains
  - At least, **9.6 %** of the top 5000 Web pages contain at least one DOM-based XSS problem
  - This number only represents the lower bound (!)

Summary

- We built a tool capable of detecting flows
  - patching the browser
  - building the extension
  - crawling the Web
- We built an automated exploit generator
  - taking into account the exact taint information
  - ... and specific contexts
- We found that at least **480** of the top **5000** domains carry a DOM-XSS vuln

Outlook on future work

Sources

Sinks

	URL	Cookie	Referrer	window.name	postMessage	WebStorage	Total
HTML	1,356,796	1,535,299	240,341	35,446	35,103	16,387	3,219,392
JavaScript	22,962	359,962	511	617,743	448,311	279,383	1,728,872
URL	3,798,228	2,556,709	313,617	83,218	18,919	28,052	6,798,743
Cookie	220,300	10,227,050	25,062	1,328,634	2,554	5,618	11,809,218
post Message	451,170	77,202	696	45,220	11,053	117,575	702,916
Web Storage	41,739	65,772	1,586	434	194	105,440	215,165
Total	5,891,195	14,821,994	581,813	2,110,715	516,134	552,455	24,474,306
Filters	64,78%	52,81%	83,99%	57,69%	**1,57%**	30,31%	

Outlook on future work

Sources

Sinks

	URL	Cookie	Referrer	window.name	postMessage	WebStorage	Total
HTML	1,356,796	1,535,299	240,341	35,446	35,103	16,387	3,219,392
JavaScript	22,962	359,962	511	617,743	448,311	279,383	1,728,872
URL	3,798,228	2,556,709	313,617	83,218	18,919	28,052	6,798,743
Cookie	220,300	10,227,050	25,062	1,328,634	2,554	5,618	11,809,218
post Message	451,170	77,202	696	45,220	11,053	117,575	702,916
Web Storage	41,739	65,772	1,586	434	194	105,440	215,165
Total	5,891,195	14,821,994	581,813	2,110,715	516,134	552,455	24,474,306
Filters	64,78%	52,81%	83,99%	57,69%	**1,57%**	30,31%	

Outlook on future work

Sources

Sinks

	URL	Cookie	Referrer	window.name	postMessage	WebStorage	Total
HTML	1,356,796	1,535,299	240,341	35,446	35,103	16,387	3,219,392
JavaScript	22,962	359,962	511	617,743	448,311	279,383	1,728,872
URL	3,798,228	2,556,709	313,617	83,218	18,919	28,052	6,798,743
Cookie	220,300	10,227,050	25,062	1,328,634	2,554	5,618	11,809,218
post Message	451,170	77,202	696	45,220	11,053	117,575	702,916
Web Storage	41,739	65,772	1,586	434	194	105,440	215,165
Total	5,891,195	14,821,994	581,813	2,110,715	516,134	552,455	24,474,306
Filters	64,78%	52,81%	83,99%	57,69%	**1,57%**	30,31%	

window.name flows

- Huge number of flows from window.name
    - closer analysis shows programming errors
    - variable „name“ defined in global scope
        - or not declared with keyword „var“
    - global object = window...
- Might actually have privacy impact
    - window.name can be read cross-domain

```
<script>
    var name = doSomething();
    document.write(name);
</script>
```

```
function test(){
    name = doSomething();
    document.write(name);
};
```

Outlook on future work

Sources

Sinks

	URL	Cookie	Referrer	window.name	postMessage	WebStorage	Total
HTML	1,356,796	1,535,299	240,341	35,446	35,103	16,387	3,219,392
JavaScript	22,962	359,962	511	617,743	448,311	279,383	1,728,872
URL	3,798,228	2,556,709	313,617	83,218	18,919	28,052	6,798,743
Cookie	220,300	10,227,050	25,062	1,328,634	2,554	5,618	11,809,218
post Message	451,170	77,202	696	45,220	11,053	117,575	702,916
Web Storage	41,739	65,772	1,586	434	194	105,440	215,165
Total	5,891,195	14,821,994	581,813	2,110,715	516,134	552,455	24,474,306
Filters	64,78%	52,81%	83,99%	57,69%	**1,57%**	30,31%	

Outlook on future work

Sources

Sinks

	URL	Cookie	Referrer	window.name	postMessage	WebStorage	Total
HTML	1,356,796	1,535,299	240,341	35,446	35,103	16,387	3,219,392
JavaScript	22,962	359,962	511	617,743	448,311	279,383	1,728,872
URL	3,798,228	2,556,709	313,617	83,218	18,919	28,052	6,798,743
Cookie	220,300	10,227,050	25,062	1,328,634	2,554	5,618	11,809,218
post Message	451,170	77,202	696	45,220	11,053	117,575	702,916
Web Storage	41,739	65,772	1,586	434	194	105,440	215,165
Total	5,891,195	14,821,994	581,813	2,110,715	516,134	552,455	24,474,306
Filters	64,78%	52,81%	83,99%	57,69%	**1,57%**	30,31%	

Thank you for your attention!

Martin Johns

@datenkeller

martin.johns@sap.com