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

CCS 2013, Berlin

Sebastian Lekies, **Ben Stock**, Martin Johns

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Agenda

- XSS & Attacker Scenario
  - WebSec guys: wake up once you see a cat
- Motivation
- Our contributions
- Summary

Cross-Site Scripting

- Execution of attacker-controlled code on the client in the *context* of the vulnerable app
- Three kinds:
  - Persistent XSS: guestbook, ...
  - Reflected XSS: search forms, ...

    } Server side

  - DOM-based XSS: also called local XSS
    - content dynamically added by JS (e.g. like button), ...

    } Client side

Cross-Site Scripting: attacker model

- Attacker wants to inject own code into vuln. app
  - steal cookie
  - take abritrary action in the name of the user
  - pretend to be the server towards the user
  - ...

Source: http://blogs.sfweekly.com/thesnitch/cookie_monster.jpg

Cross-Site Scripting: problem statement

- **Main problem:** attacker‘s content ends in document and is not properly 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

Example of a DOMXSS vulnerability

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

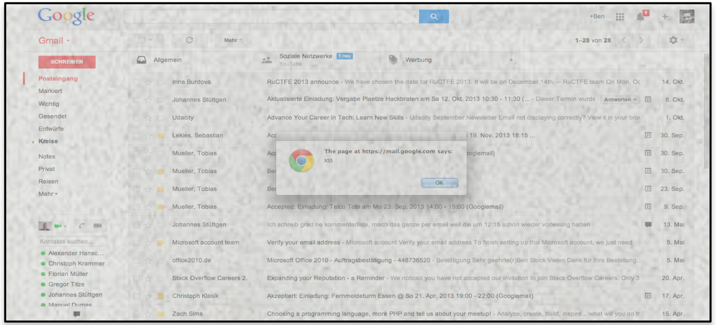
- Source: *location.hash*, Sink: *document.write*
- Intended usage:
  - http://example.org/#mypage
  - `<img src='//adve.rt/ise?hash=mypage'/>`
- Exploiting the vuln:
  - http://example.org/#'/><script>alert(1)</script>
  - ```
    <img src='//adve.rt/ise?hash='/>
    <script>alert(1)</script>
    '/>
    ```
    ```

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

Our motivation and contribution

- Perform Large-scale analysis of DOMXSS vulnerabilities
  - Automated, dynamic detection of suspicious flows
  - Automated validation of vulnerabilities
- Our 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

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Our approach: use dynamic taint tracking

- **Taint tracking**: Track the flow of *marked* data from source to sink
- **Implementation**: into Chromium (Blink+V8)
- **Requirements for taint tracking**
  - Taint all relevent values / propagate taints
  - Report all sinks accesses
  - **be as precise as possible**
    - taint details on EVERY character

Representing sources

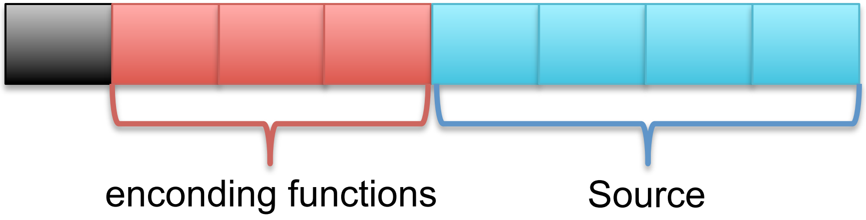
- In terms of DOMXSS, we have **14** sources
- additionally, **three** relevant, built-in encoding functions
  - escape, encodeURI and encodeURIComponent
  - .. **may** prevent XSS vulnerabilites if **used properly**
- Goal: store *source* + *bitmask of encoding functions* for each character

Representing sources (cntd)

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

7 bits < 1 byte

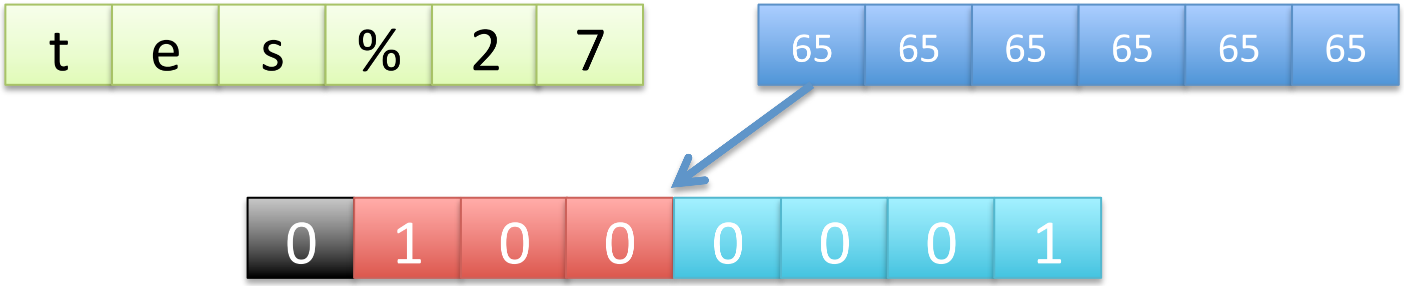
- ➔ 1 Byte sufficient to store source + encoding functions
  - encoding functions and counterparts set/unset bits
  - hard-coded characters have source 0



Representing sources (cntd)

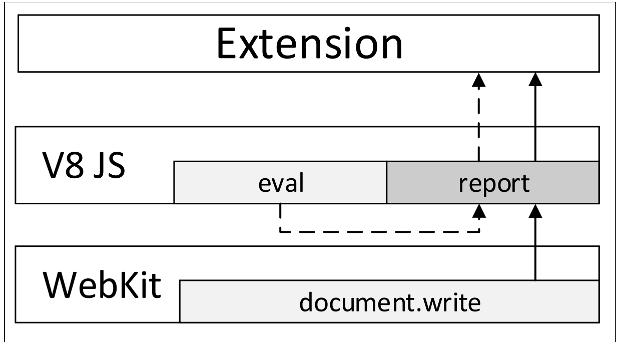
- Each source API (e.g. URL or cookie) attaches taint bytes
  - identifing the source of a char
  - `var x = location.hash.slice(1);`

  - `x = escape(x);`



Detecting sink access

- **Taint propagated through all relevant functions**
- **Security-sensitive sinks report flow and details**
  - such as text, taint information, source code location
- **Chrome extension to handle reporting**
  - keep core changes as small as possible
  - repack information in JavaScript
  - stub function directly inside V8

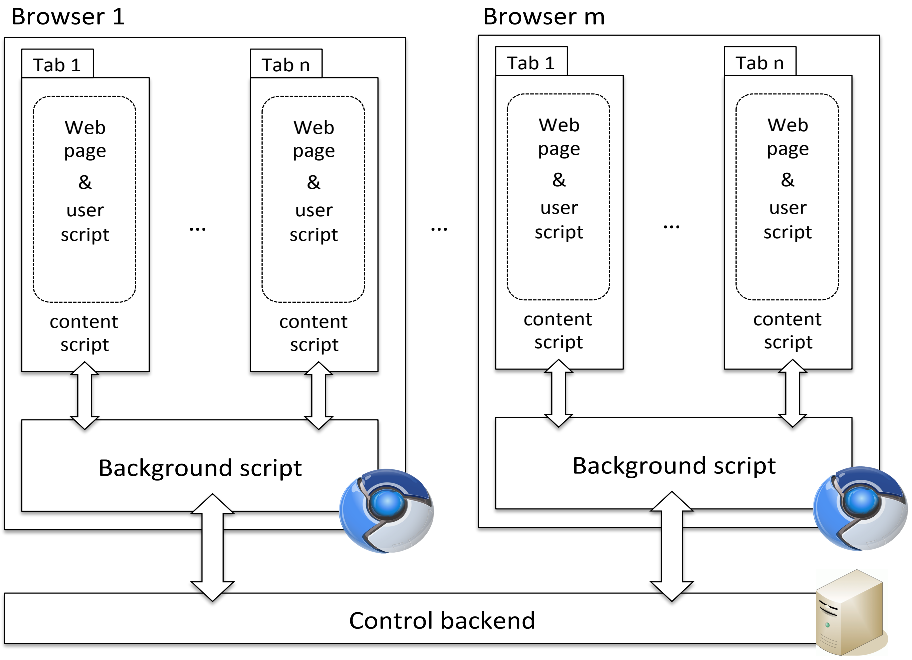


Empirical study on suspicious flows

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

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 possibly attacker-controllable input to security-sensitive sinks

Context-Sensitive Generation of Cross-Site Scripting Payloads

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Validating vulnerabilities

- 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("<img src='pic.jpg?hash="
          + location.hash.slice(1) + "'>");
```

    - Exploit:

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

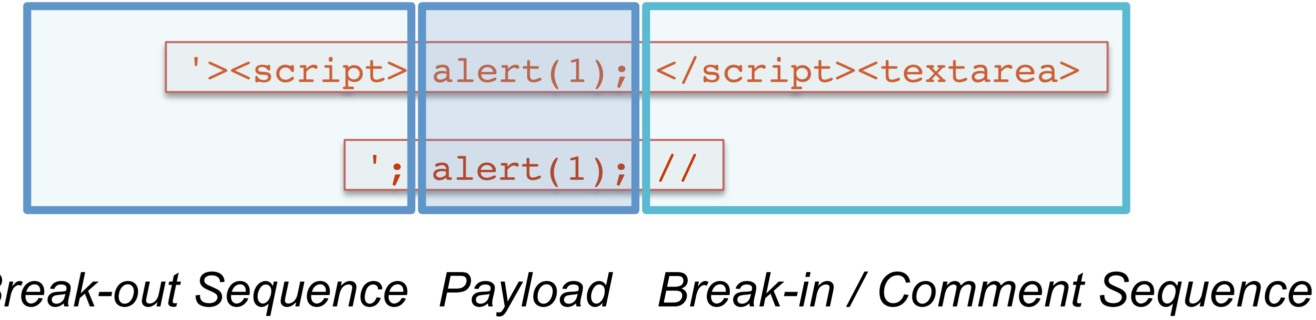
  - JavaScript Context
    - Vulnerability:

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

    - Exploit:

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

Anatomy of an XSS Exploit



- **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/ in our engine

```
eval('
function test() {
  var x = "http://example.org";
  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: “;}
  - Comment: //
  - **Exploit: ";}alert(1);//**
  - Visit: http://example.org/#";}alert(1);//

```
function test() {
  var x = "http://example.org";
}
alert(1);//“; doSomething(x); }
```

Validating vulnerabilities

- Our focus: directly controllable exploits
  - *Sinks*: direct execution sinks
    - HTML sinks (document.write, innerHTML ,...)
    - JavaScript sinks (eval, ...)
  - *Sources*: location and referrer
  - 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 unencoded flows: 1,825,598 ➔ **313,794**

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

Empirical study

- **Step 4: Exploit validation**
  - **69,987** out of **181,238** unique test cases triggered a vulnerability

- **Step 5: 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 one or more XSS problems
  - This number only represents the lower bound (!)

Limitations

- No assured code coverage
  - e.g. debug GET-param needed?
  - also, not all pages visited (esp. stateful applications)
- Fuzzing might get better results
  - does not scale as well
- Not yet looking at the „harder“ flows
  - found one URL ➔ Cookie ➔ eval „by accident“

Summary

- We built a tool capable of **detecting** flows
  - taint-aware Chromium
  - Chrome extension for crawling and reporting
- 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

Thank you very much for your attention!

Ben Stock
@kcotsneb
ben.stock@fau.de

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

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
Encoded	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
Encoded	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
Encoded	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
Encoded	64,78%	52,81%	83,99%	57,69%	**1,57%**	30,31%	