Finding your Way in the Testing Jungle

A Learning Approach to Web Security Testing

Omer Tripp
IBM
Tel Aviv U.

Omri Weisman
(formerly)
IBM

Lotem Guy
(formerly)
IBM
detecting XSS in web applications using black box security testing
effectiveness = \frac{#successes}{effort}
Effectiveness

brute force

R5000
R2500
R1000
R500
R250
R100

expert selection

our approach

log scale

R

ISSTA 2013
USER

hello

SERVER

Server-side code: Input validation

query = query.replace("script","\"\")

HTML response

<h1>Search Results</h1>
<p>No results were found for the query:<br />
hello</p>
Server-side code: Input validation

query = query.replace("script","");
security vulnerability = PROOF = attack

XSS attacks: >500 Million

∀ element: 500 Million

to be tested for security

payloads

↑

thousands of parameters,
cookies, forms, ...
dynamic/learning based approach

searches

500,000,000

in 10 steps
### Simple Example

**Reflection:** `<frameset><frame src='[HERE]'></frame></frameset>`

**Sanitizer:** HTML Encode (&, "", ', <, >)

<table>
<thead>
<tr>
<th>#</th>
<th>Payload</th>
<th>Reflection</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>onload=&quot;alert(1)</code></td>
<td><code>&lt;frame src='</code>onload=<code>alert(1)</code>&amp;quot<code>&gt;</code>&lt;br&gt;<code>&lt;/frame&gt;</code></td>
<td><strong>Learn:</strong> avoid &quot;</td>
</tr>
</tbody>
</table>
### Simple Example

**Reflection:** `<frameset><frame src=`[HERE]`>`</frame></frameset>`

**Sanitizer:** HTML Encode (&, ', ', <, >)

<table>
<thead>
<tr>
<th>#</th>
<th>Payload</th>
<th>Reflection</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>onload=&quot;alert(1)&quot;</code></td>
<td><code>&lt;frame src=&quot;</code> onload=&amp;quot&gt;alert(1)&quot;<code>&gt;</code></td>
<td>Learn: avoid &quot;</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>&lt;/frame&gt;</code></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td><code>onload='alert(2)'</code></td>
<td><code>&lt;frame src=&quot;</code> onload=&amp;#039&gt;alert(2)'<code>&gt;</code></td>
<td>Learn: avoid '</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>&lt;/frame&gt;</code></td>
<td></td>
</tr>
</tbody>
</table>
## Simple Example

**Reflection:** `<frameset><frame src='[HERE]'></frame></frameset>`

**Sanitizer:** HTML Encode (`, `), `<, >`

<table>
<thead>
<tr>
<th>#</th>
<th>Payload</th>
<th>Reflection</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>onload=&quot;alert(1)</code></td>
<td><code>&lt;frame src='</code>onload=&quot;alert(1)<code>&amp;quot;</code>&gt;`&lt;/frame&gt;</td>
<td>Learn: avoid &quot;</td>
</tr>
<tr>
<td>2</td>
<td><code>onload='alert(2)'</code></td>
<td><code>&lt;frame src='</code>onload=&amp;#039alert(2)&amp;#039<code>'&gt;</code>&gt;`&lt;/frame&gt;</td>
<td>Learn: avoid '</td>
</tr>
<tr>
<td>3</td>
<td><code>onload='alert(3)</code></td>
<td><code>&lt;frame src='</code>onload=<code>alert(3)</code>&gt;`&lt;/frame&gt;</td>
<td>Vulnerability found!</td>
</tr>
</tbody>
</table>
Tokenization: Example

\(<\text{script}>\text{alert}(1234)\text{</script}>\)
A Learning Approach

Knowledge base

Select and Send Test

Prune Search Space

Study Response

Learn Constraints
Vulnerability Types

Structural

Bypass
query = query.replace("script","'");

Structural

Bypass
Learning approach (informal)

Constraints = {}

Pick test from KB

Send test, find reflection in response

reflection == test?

YES

Test individual tokens

NO

Report vulnerability and finish

Learn constraints
Grammar: Derivation Rules

\[ jsPayload \rightarrow [\text{window[location]}=\text{jsLocation}] \mid \text{alert(probe)} \mid \text{probe}+\{\text{jsFunc:alert}\} \mid \text{eval('ale'+rt+'('prober')')} \mid \text{eval("ale"+"rt"+"("prober")")} \mid \text{eval(‘ale`+`rt`+`(`prober`))} \]

\[ \text{location} \rightarrow 'location' \mid "location" \mid `location` \]

\[ \text{jsLocation} \rightarrow ... \]

\[ \text{jsFunc} \rightarrow \text{toString} \mid \text{valueOf} \]

\[ \text{probe} \rightarrow ... \]
Empirical Testing
Empirical Testing

- 15,552 benchmark test cases

- Compared:
  - XSS Analyzer
  - Brute force
  - Random samplings:
    - \{100, 200, \ldots, 1000\} U \{1500, 2000, \ldots, 5000\}
  - “Expert selection” by IBM Security AppScan®

- COVERAGE: How many vulnerabilities were found?
- PERFORMANCE: How many requests were sent?
Results

- IBM AppScan: 43% Coverage, 40 Requests
- R100: 55% Coverage, 67 Requests
- R250: 64% Coverage, 148 Requests
- R500: 70% Coverage, 263 Requests
- R1000: 75% Coverage, 481 Requests
- R2500: 80% Coverage, 1062 Requests
- R5000: 84% Coverage, 1724 Requests
- brute force: 100% Coverage, 2301 Requests
- XSS Analyzer: 99% Coverage, 10 Requests

ISSTA 2013
Grammar Completeness Impact

- How important is it to have a really complete grammar?

- We compared:
  - Full grammar
  - Grammar where all whitespaces other than space are excluded
  - Grammar without any browser-specific exploits
  - Brute force
Results: Grammar Completeness

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Vuln.s</th>
<th>Coverage</th>
<th>Requests (avg.)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Total</td>
<td>Succ.</td>
</tr>
<tr>
<td>brute force</td>
<td>10356</td>
<td>100%</td>
<td>2301</td>
<td>95</td>
</tr>
<tr>
<td>full</td>
<td>10245</td>
<td>99%</td>
<td>10</td>
<td>6</td>
</tr>
<tr>
<td>limited whitespaces</td>
<td>10116</td>
<td>98%</td>
<td>10</td>
<td>6</td>
</tr>
<tr>
<td>w/o browser</td>
<td>8162</td>
<td>79%</td>
<td>12</td>
<td>7</td>
</tr>
</tbody>
</table>

Table 2: Statistics on the behavior of XSS Analyzer when run over a partial grammar
Contributions:
1. Industry grade dynamic learning black-box XSS testing algorithm
2. Grammar representation of test payloads
3. Implementation and evaluation

See more discussion, detail in the paper
Finding your Way in the Testing Jungle

A Learning Approach to Web Security Testing

Omer Tripp
IBM
Tel Aviv U.

Omri Weisman
(formerly)
IBM

Lotem Guy
(formerly)
IBM