Fuzz Testing of Web Browsers

Presented by Renata Hodovan
Outline

• **Principles** of random testing
• **Pros & Cons** of fuzzing
• **How to** fuzz a browser?
• **Evaluation** of a real life framework
Principles of Fuzz Testing

• **Idea:** Stress testing the target with deformed inputs

• **Expected bugs (primarily):**
  - Implementation mistakes
  - Non-semantic issues

• **Automated testing**
Possible Issue Types

• **Stability** issues
  - Crashes
  - Memory corruptions
  - Hangs

• **Semantic** issues
  - Assertion failures
  - Output mismatch with an oracle
Pros & Cons

• **Pros**
  - It works! :-)  
  - Fast and cheap  
  - No need for source code  
  - Portable

• **Cons**
  - Smart fuzzing can be challenging
How to Fuzz a Browser?
What do You Need?

- Evil test generator algorithm
- Transfer mechanism
- Monitoring framework
Browser Fuzzing Framework

Test

Test

Test
Variations for Generators

- Random noise
- Mutation based
- Generation based
- Combination of the above
First Steps ...

- Tests with **random** character sequences
- **Pros:**
  - Fast and easy to implement
- **Cons:**
  - Mostly fails on the first checks
  - Not able to find complex errors
- Found bug in WebKit (Apple Safari)
  - &#6198;&#656
Mutation Based Approaches

- **Idea:** the most error-prone tests are the almost good ones
- Let's mess up existing tests!

**Pros:**
- Still easy to implement
- Much more effective than purely random

**Cons:**
- The variety of possible tests depends on the initial test set
Mutation Based Approaches

• Ingredients:
  - Existing test cases
  - Parser for the tests
  - Test domain (in)competence
Mutation Based Approaches

• Replace tokens with random contents:

<applet code="good.class"></applet>

<applet code="foo.bar"></applet>

(Google Chrome issue)
Generation Based Approach

- **Ingredients:**
  - Model/grammar describing the input format
    - E.g., in BNF format
  - Automatism that processes the description and generates a fuzzer
    - ANTLR
Generation Based Approach

Input grammar → Fuzzer grammar

ANTLR → JAVA

JAVA → HTML

JAVA → CSS

JAVA → JS
Generation Based Approach

• **Pros:**
  - Not bound to any input test set
  - Easier to extend
  - Increased coverage

• **Cons:**
  - Needs much more preliminary work
How to Obtain the Input Grammar?

- Make your hands dirty! **Write it yourself!**
- Extract it from **standards**
  - E.g., from XSD or IDL definitions
  - They can be processed automatically
- Extract from existing **test cases**
  - Uncover undocumented features
- **Combine** all of them
Further Challenges

• Grammars can only describe syntactic requirements but not semantic ones. E.g.,:
  - Variable matching
  - Using functions with “correct” parameter list
  - Building valid relations between XML nodes

• Solution:
  - Adding semantic information manually
    - E.g., using symbol tables
How to Obtain the Input Grammar?

• Make your hands dirty! **Write it yourself!**
• Extract it from **standards**
  ▶ E.g., from XSD or IDL definitions
  ▶ They can be processed automatically
• Extract from existing **test cases**
  ▶ Uncover undocumented features
• **Combine** all of them
function f_0(){
    for(var v_0 in [10]){
        try {
            for(var v_1 in [10])
                return;
        } finally {}
    }
}
for(var v_2 in f_0()) {}

(JavaScriptCore issue)
Features

• Supported languages
  ▸ HTML
  ▸ SVG
  ▸ MathML
  ▸ CSS
  ▸ JavaScript
  ▸ Combinations of the above

• Applied techniques
  ▸ Mutation
  ▸ Generation

• Grammar sources
  ▸ Hand-written
  ▸ Extracted from XSD, IDL, web standards

• Advanced features
  ▸ ID matching, self adapting weights
## Results in Numbers

<table>
<thead>
<tr>
<th>Engine</th>
<th>Number of bugs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Google Chrome</td>
<td>274</td>
</tr>
<tr>
<td>Apple Safari</td>
<td>257</td>
</tr>
<tr>
<td>Jerryscript (JS engine)</td>
<td>96</td>
</tr>
</tbody>
</table>
Questions?