Testing Web Applications

Jeff Offutt

http://www.cs.gmu.edu/~offutt/

SWE 642
Software Engineering for the World Wide Web

Joint research with Ye Wu, Blaine Donley, Xiaochen Du, Wuzhi Xu, and Hong Huang

Outline

1. Testing Web Applications is Different
2. HtmlUnit
3. User Session Data
4. Bypass Testing
5. Atomic Section Modeling
General Problem

• Web applications are heterogeneous, dynamic and must satisfy very high quality attributes

• Use of the Web is hindered by low quality Web sites and applications

• Web applications need to be built better and tested more

Separation of Concerns in Web Apps

• Presentation layer  HTML, output and UI

• Data content layer  Computation, data access

• Data representation layer  In-memory data storage

• Data storage layer  Permanent data storage
Differences in Testing Web Software

- Traditional graphs do not apply
  - Control flow graph
  - Call graph
- State behavior is hard to model and describe
- All inputs go through the HTML UI – low controllability
- Hard to get access to server-side state (memory, files, database) – low observability
- Not clear what logic predicates can be effectively used
- No model for mutation operators on web software

Example Problem 1

Yikes, did they take my money or not?

Examples from Blaine Donley
Example Problem 2

Why, and what does this mean?

Example Problem 3

Why should I trust you enough to try again?
New Essential Problems of Web Software

1. Web site software is extremely loosely coupled
   - Coupled through the Internet – separated by space
   - Coupled to diverse hardware and software applications
   - Web services will dynamically couple with other services after deployment – without human intervention!

2. Web software services offer dynamically changing flow of control
   - Web pages are created by software on user request
   - The interaction points (forms, buttons, etc.) vary depending on state: the user, previous choices, server-side data, even time of day
   - Examples: amazon.com, netflix.com, washingtonpost.com
   - Finding all screens in a web app is an undecidable problem

Extremely Loose Coupling

- **Tight Coupling**: Dependencies among the methods are encoded in their logic
  - Changes in A may require changing logic in B
- **Loose Coupling**: Dependencies among the methods are encoded in the structure and data flows
  - Changes in A may require changing data uses in B
- **Extremely Loose Coupling (ELC)**: Dependencies are encoded only in the data contents
  - Changes in A only affects the contents of B’s data
Ramifications of ELC

- Web applications encourage ELC
  - Physical separation of hardware and software makes ELC necessary
  - XML supports ELC
- ELC has some non-obvious affects
  - Software modules can dynamically integrate with others if they use the same data structures
  - EJBs can be inserted into Web applications, which can immediately start using them

Dynamic Flow of Control

How can we ensure the reliability of this type of system?
Example Problem 4

Fuhgetabout it ... I'm going to barnesandnoble.com

Example Problem 5

Oh yeah?? I'm definitely pushing BACK!
Example Problem 6

What if I do?
And ... how long should I wait?

Example Problem 7

Doh!!! Shoot the designer!
Outline

1. Testing Web Applications is Different
2. HtmlUnit
3. User Session Data
4. Bypass Testing
5. Atomic Section Modeling

JUnit to HtmlUnit

- JUnit is a very simple “framework” for automating unit-level Java tests
  - Makes calls to methods in a Java class, reports results
- HttpUnit was built on top of JUnit to automate web application tests
- HtmlUnit was created to be simpler, higher level, and provide more support for JavaScript
What is HtmlUnit?

- The developers call it a “headless browser”
- Open source
- Used by many professional testers
- A tool to help automate tests for web applications

What Does HtmlUnit Do?

- A test driver program
- Java class with methods that contain:
  - Target URL
  - Form data
- Rather than testing a web app by typing and clicking, test inputs are written into a program
  - Easy to run many tests at once
  - Fewer mistakes—tests can be checked
  - Tests can be saved and re-run
  - More tests can be created by copy, paste, and modify
Example HtmlUnit Use

```java
@Test
public void htmlUnitWebsiteFromGoogle() throws Exception {
    WebClient webClient = new WebClient(BrowserVersion.FIREFOX_3);

    HtmlPage startPage = webClient.getPage("http://www.google.com");
    assertEquals("Google", startPage.getTitleText());

    HtmlInput queryField = (HtmlInput) startPage.getElementsByTagName("q").get(0);
    queryField.setValueAttribute("HtmlUnit");

    HtmlElement button =
        startPage.getFirstByXPath("//input[@value = "I'm Feeling Lucky"]");
    HtmlPage p2 = button.click();
    assertEquals("HtmlUnit – Welcome to HtmlUnit", p2.getTitleText());
}
```
What Kinds of Errors?

- Any functional errors in the back-end software
- JavaScript errors
- HTTP errors: 404, 500, ...
- Problems in the “WUI” … the web user interface
  - Incorrect HTML
  - Malformed URLs
  - Incorrect headers
  - …

What Is Missing?

The hard part is deciding what data to supply to the web application …
User Session Data

- The server logs all requests from users
- By default, form values are not kept
- But if saved, the logs can provide the raw resources to create test values during regression testing
  - Configure the web server
  - Add JavaScript to invoke server-side logging script
Using Session Data

• Exact replays of users previous sessions to check for changes in the software’s behavior

• Mix multiple sessions together to test new combinations and scenarios

• Replay sessions in parallel for concurrency testing

• Modify the session sequence of requests, but use the old data to fill in the form fields

Limitations

• Does not include “new” values, just old values

• Users tend to follow likely scenarios most of the time—the hard faults are hiding behind rarely used input data
  – These are called “happy paths”

• Limited help with new functionality

• This is purely client-based (“black box”) … what can we test if we analyze the source?
Outline

1. Testing Web Applications is Different
2. HtmlUnit
3. User Session Data
4. Bypass Testing
5. Atomic Section Modeling

Abbreviated HTML

```html
<FORM >
  <INPUT Type="text" Name="username" Size=20>
  <INPUT Type="text" Name="age" Size=3 Maxlength=3>
  <P> Version to purchase:
  ...
  <INPUT Type="radio" Name="version" Value="150" Checked>
  <INPUT Type="radio" Name="version" Value="250">
  <INPUT Type="radio" Name="version" Value="500">
  <INPUT Type="submit" onClick="return checkInfo(this.form)"
  <INPUT Type="hidden" isLoggedIn="no">
</FORM>
```
Bypass Behavior

• Extremely loose coupling …

• combined with the stateless protocol …

• allows users to easily bypass client-side checking:

Users can save and modify the HTML

Saved & Modified HTML

<FORM >
<INPUT Type="text" Name="username" Size=20>
<INPUT Type="text" Name="age" Size=3 Maxlength=3>
<P> Version to purchase:
  …
<INPUT Type="radio" Name="version" Value=150>
<INPUT Type="radio" Name="version" Value=250>
<INPUT Type="radio" Name="version" Value=500 Checked>
<INPUT Type="submit" onClick="return checkInfo (this.form)"
<INPUT Type="hidden" isLoggedIn= "no" >
</FORM>
SQL Injection

User Name: turing' OR '1'='1 Password: enigma' OR '1'='1

Original SQL:
SELECT username FROM adminuser WHERE username='turing' AND password = 'enigma'

“injected” SQL:
SELECT username FROM adminuser WHERE username='turing' OR '1' = '1' AND password = 'enigma' OR '1' = '1'

Bypass Testing

• This example illustrates how users can “bypass” client-side constraint enforcement
• Bypass testing constructs tests to intentionally violate constraints
  – Eases test automation
  – Checks robustness
  – Evaluates security
• Preliminary results
  – Rules for constructing tests
  – Successfully found errors in numerous Web apps
Applying Bypass Testing

Validating input data on the client is like asking your opponent to hold your shield in a sword fight

- Analyze HTML to extract each form element
- Model constraints imposed by HTML and JavaScript
- Rules for data generation:
  - From client-side constraints
  - Typical security violations
  - Common input mistakes

Types of Client Input Validation

- Client side input validation is performed by HTML form controls, their attributes, and client side scripts that access DOM
- Validation types are categorized as HTML and scripting
  - HTML supports syntactic validation
  - Client scripting can perform both syntactic and semantic validation

<table>
<thead>
<tr>
<th>HTML Constraints</th>
<th>Scripting Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Length <em>(max input characters)</em></td>
<td>• Data Type <em>(e.g. integer check)</em></td>
</tr>
<tr>
<td>• Value <em>(preset values)</em></td>
<td>• Data Format <em>(e.g. ZIP code format)</em></td>
</tr>
<tr>
<td>• Transfer Mode <em>(GET or POST)</em></td>
<td>• Data Value <em>(e.g. age value range)</em></td>
</tr>
<tr>
<td>• Field Element <em>(preset fields)</em></td>
<td>• Inter-Value <em>(e.g. credit # + exp. date)</em></td>
</tr>
<tr>
<td>• Target URL <em>(links with values)</em></td>
<td>• Invalid Characters <em>(e.g. &lt;,../&amp;,)</em></td>
</tr>
</tbody>
</table>
**Example Client-Side Constraint Rules**

- Violate size restrictions on strings
- Introduce values not included in static choices
  - Radio boxes
  - Select (drop-down) lists
- Violate hard-coded values
- Use values that JavaScripts flag as errors
- Change “transfer mode” (get, post, …)
- Change destination URLs

**Example Server-Side Constraint Rules**

- Data type conversion
- Data format validation
- Inter-field constraint validation
- Inter-request data fields (cookies, hidden)
### Example Security Violation Rules

<table>
<thead>
<tr>
<th>Potential Illegal Character</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>Empty String</td>
<td></td>
</tr>
<tr>
<td>Commas</td>
<td>,</td>
</tr>
<tr>
<td>Single and double quotes</td>
<td>' or &quot;</td>
</tr>
<tr>
<td>Tag symbols</td>
<td>Tag symbols &lt; and &gt;</td>
</tr>
<tr>
<td>Directory paths</td>
<td>..../</td>
</tr>
<tr>
<td>Strings starting with forward slash</td>
<td>/</td>
</tr>
<tr>
<td>Strings starting with a period</td>
<td>.</td>
</tr>
<tr>
<td>Ampersands</td>
<td>&amp;</td>
</tr>
<tr>
<td>Control character</td>
<td>NIL, newline</td>
</tr>
<tr>
<td>Characters with high bit set</td>
<td>254 and 255</td>
</tr>
<tr>
<td>Script symbols</td>
<td>&lt;javascript&gt; or &lt;vbscript&gt;</td>
</tr>
</tbody>
</table>

### Test Value Selection

- **Challenge:**
  - How to automatically provide effective test values?
- **Semantic Domain Problem (SDP)**
  - Values within the application domain are needed
  - Enumeration of all possible test values is inefficient
- **Possible Solutions**
  - Random Values (ineffective – lots of junk)
  - Automatically generated values (very hard)
  - Taking values from session log files (feasible but incomplete)
  - Tester input (feasible)
- **Our tool used an input domain created by parsing the interface and tester input**
Real-World Examples

- atutor.ca
- nytimes.com
- bankofamerica.com

- Atalker
- Us-markets
- ATM locator, Site search

- demo.joomla.or Poll, Users
- mutex.gmu.edu
- comcast.com
- Login form

- bank of america.com

- atm locator, Site search
- Site search

- phpMyAdmin
- yahoo.com
- ecost.com
- Main page, Notepad, Composer, Detail submit,
- Set Theme, Search reminder, Shopping cart control
- SQL Query, Weather Search

- dbstats

- brainbench.com

- homepage, submit request
- mutex.gmu.edu
- google.com

- phpMyAdmin
- login form

- brainbench.com

- submit request

- myspace.com

- events & music

- barnesandnoble.com

- cart manager

- amazon.com

- item dispatch, handle buy

- Pure black-box testing means no source (or permission) needed!

Output Checking

- (V) Valid Responses: invalid inputs are adequately processed by the server
  (V1) Server acknowledges the invalid request and provides an explicit message regarding the violation
  (V2) Server produces a generic error message
  (V3) Server apparently ignores the invalid request and produces an appropriate response
  (V4) Server apparently ignores the request completely

- (F) Faults & Failures: invalid inputs that cause abnormal server behavior (typically caught by web server when application fails to handle the error)

- (E) Exposure: invalid input is not recognized by the server and abnormal software behavior is exposed to the users

- These do not capture whether the valid responses corrupted data on the server
Results

Outline

1. Testing Web Applications is Different
2. HtmlUnit
3. User Session Data
4. Bypass Testing
5. Atomic Section Modeling
Dynamic Execution of Web Apps

• Parts of the program are generated dynamically
• Dynamic web pages are created from user requests
• Different users will see different programs!
• Users can make unexpected changes to the flow of control
  – *Operational transitions* : Transitions NOT based on an HTML link: back-button, URL rewriting, refresh

*The potential flow of control cannot be known statically.*

Control Flow Graphs in Web Applications

• Many testing criteria on non-Web software rely on a static control flow graph
  – Edge testing, data flow, logic coverage …
  – Also slicing, change impact analysis, …
• Static control flow graphs cannot be computed for Web applications!
• But all the pieces of the web pages and the programs are contained in the software …
Atomic Sections

PrintWriter out = response.getWriter();

out.println (<HTML>)
out.println (<HEAD><TITLE>title</TITLE></HEAD>)
out.println (<BODY>)

if (isUser) {
    out.println (<CENTER>Welcome!</CENTER>);
    for (int i=0; i<myVector.size(); i++)
        if (myVector.elementAt(i).size > 10)
            out.println (<p><b>myVector.elementAt(i)</b></p>);
        else
            out.println (<p>myVector.elementAt(i)</p>);
    } else {
    }

out.println (<</BODY></HTML>);
out.close ();

Atomic Sections Defined

• A section of HTML with the property that if any part of the section is sent to a client, the entire section is
  – May include JavaScript
  – All or nothing property
• An HTML file is an atomic section
• Content variable: A program variable that provides data to an atomic section
• Atomic sections may be empty
Composite Sections

- Atomic sections are combined to create dynamically generated web pages
- Four ways to combine:
  1. Sequence: $p_1 \cdot p_2$
  2. Selection: $(p_1 \mid p_2)$
  3. Iteration: $p_1^*$
  4. Aggregation: $p_1 \{p_2\}$
     - $p_2$ is included inside of $p_1$
- The previous example produces:
  $p \to p_1 \cdot ((p_2 \cdot (p_3 \mid p_4)^*) \mid p_5) \cdot p_6$
- Composite sections can be generated automatically

Modeling Dynamic Interaction

- Interactions are classified into three types of transitions:
  1. Link Transition: An HTML link
  2. Composite Transition: Execution of a software component causes a composite section to be sent to the client
  3. Operational Transition: A transition out of the software’s control
     - Back button
     - Refresh button
     - User edits the URL (URL rewriting)
     - Browser reloads from cache
Modeling Web Applications

- Intra-component Level: Abstract description of each component in the Web application

- Inter-component Level: A graphical representation of the entire Web application

Intra-Component Level

For each component:
- Start page
- Atomic sections
- Composite sections
- Composition rules
- Transition rules
Inter-Component Level

- A Web Application Graph (WAG)
  - Nodes are web components
  - Edges are transitions
- Three types of transitions
  1. Static links
  2. Dynamic links
  3. Forwarding links
- Annotations on links
  - Type of HTTP request
  - Data being transmitted as parameters
- Current State: static variables and session information

Test Criteria

- Tests can be applied at intra- and inter-component level
- Tests are created by deriving sequences of transitions among the Web software components and composite sections
Composite Section Test Criteria
Intra-Component

1. All productions in the grammar
   - Multiple forms for each software component
   - Each atomic section used at least once
2. Each selection used once
   - Every form element
3. Each possible aggregation
4. MCDC type coverage of conditions on productions
   - Based on predicates from the software that separate atomic sections

WAG (Inter-Component) Tests

- **L1**: Evaluate static link transitions
  - One test generated for each form
- **L2**: L1 with two extensions
  - Values entered with URL rewriting
  - Multiple tests for each form
- **L3**: Operational transitions
  - Starting on non-initial pages, no subsequent transitions
- **L4**: Operational transitions
  - L1 tests with one operational transition at end
- **L5**: L4 + tests to traverse every transition out of the final page
Empirical Evaluation: Testing STIS

- STIS helps users keep track of arbitrary textual information
- 18 JSPs, 5 Java classes, database
- Atomic sections derived automatically
  - Parser works on Java servlets, JSPs, Java classes
- WAG derived by hand
- Form data chosen by hand
- 109 total tests
Results from Testing STIS

<table>
<thead>
<tr>
<th>Failure Category</th>
<th>L1</th>
<th>L2</th>
<th>L3</th>
<th>L4</th>
<th>L5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of tests</td>
<td>29</td>
<td>21</td>
<td>7</td>
<td>19</td>
<td>33</td>
</tr>
<tr>
<td>1. Pages displayed without authentication</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>2. Records added without authentication</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3. Runtime failures (unhandled exceptions)</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>Total number of failures</td>
<td>0</td>
<td>3</td>
<td>5</td>
<td>11</td>
<td>6</td>
</tr>
</tbody>
</table>

Previous web tests | 109 tests

Found 25 naturally occurring failures

Atomic Sections Summary

- Atomic sections fundamentally model Web applications
  - Allow the Web app form of CFGs

- Can also be used for
  - Maintenance
  - Design modeling / evaluation
  - Change impact analysis (slicing)
  - Coupling of Web application components
Open Questions

• How to define data flow?
  – DU-pairs cannot be determined statically – uses cannot always be found
• Issues not handled:
  – Session data
  – Multiple users
  – Concurrency
  – Input data
  – Output validation

Conclusions

• The Web provides a new way to deploy software
• The new technologies means that old testing techniques do not work very well
• New tools and techniques are being developed
  – HtmlUnit
  – User session based
  – Bypass testing
  – Atomic section modeling
  – Data flow testing
• Most are still in the research stage
• Most companies test web software very badly
We still have many problems to solve ...