Prototype RIA Crawler Implementation Details

Mustafa Emre Dincturk

We implemented our crawler as a prototype of IBM® Security AppScan® [2]. AppScan is an automated web scanner that is used to detect security vulnerabilities and accessibility issues in web applications. To build our crawler, we made use of some of the existing functionalities in AppScan such as the JavaScript execution engine and the DOM equivalence algorithm. The crawling strategies are implemented as a separate module that can be called from AppScan. Figure 1 shows the simplified RIA crawler architecture.

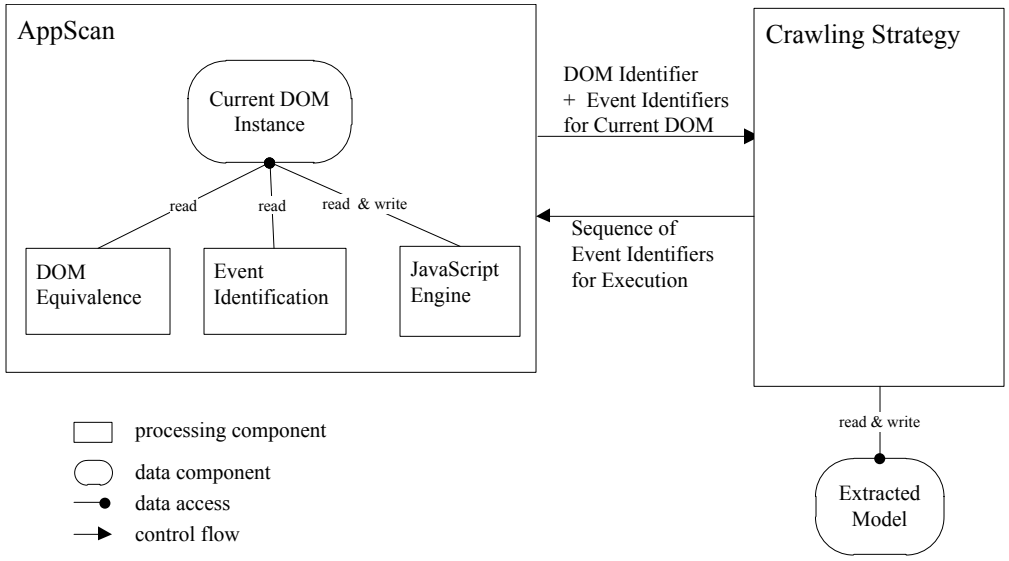

Figure 1: RIA Crawler Architecture

The AppScan component performs the functionalities of a browser. It is able construct the initial DOM for a given URL, execute JavaScript using

its JavaScript Engine sub-component and perform the required manipulations to update the current DOM. AppScan component also contains the sub-components for Event Identification and DOM equivalence. The Event identification component detects the DOM elements that have enabled events on the current DOM and produces identifiers for these events. The DOM equivalence component generates the DOM identifier for the current DOM.

When the AppScan component is given a URL to crawl, it loads the page corresponding to the URL, constructs the DOM and then generates the set of event identifiers for the events found in the DOM and generates the DOM identifier. (Details of how event and DOM identifiers are generated are explained below). Then the DOM identifier and the list of event identifiers are passed to the crawling strategy. The crawling strategy then decides on an event to explore and returns an event sequence. The event sequence consists of the event appended to a (possibly empty) transfer sequence, which will take the crawler to the DOM from where the event will be explored. The AppScan component executes the received sequence of events and returns the control back to the crawling strategy providing the event identifiers and the DOM identifier for the reached DOM. Whenever the crawling strategy takes control, it updates the extracted model according to the result of the last event exploration. This process continues until there is no unexplored event left and hence a model for the URL is extracted.

1 DOM Events and Event Identification

An important component for a RIA crawler is the algorithm to identify the events in the DOM. To compute an event identifier, we need the ability to identify the DOM element, the type of the event (for example whether it is a mouse event like a onmouseover or onclick, a key stroke) and the event handler (the JavaScript function that will be executed when the event is triggered). Identification of the DOM element is needed since events are associated with DOM elements. The event type is needed since the same DOM element can react to multiple types of events. The event handler is needed since it defines what happens when the event occurs. As we explain shortly, it is possible to change the event handlers that are registered to a DOM element through JavaScript execution. That means the same DOM element may react to the same event type differently if its event handlers are changed.

1.1 Event Registration Methods

The evolution from simple HTML pages to RIAs has resulted in three different ways to register an event handler to a DOM element.

1. **As inline HTML:** The oldest way is to specify the event handler as an attribute of the HTML element in the HTML document. For example, the following shows a `div` element that reacts to mouse click events. The event handler is registered using the onclick attribute of the HTML element. The value of the attribute is the event handler, in this case a call to a JavaScript function named doSomething.
   `<div id="id1" onclick="doSomething()">Text content</div>`
   For anchor `a` elements, it is also possible to trigger JavaScript code when the link is clicked by using the href attribute as follows.
   `<a href="javascript:'doSomething()'">Text content</a>`
   The href attribute is normally used to specify the URL that needs to be loaded when the anchor is clicked; however, browsers interpret a string that is prefixed by the keyword 'javascript:" as JavaScript code as in this example.

2. **Assignment via JavaScript:** Later, browsers allowed registration of event handlers by assigning the event handler as a property of the DOM element through JavaScript. This method made dynamic registration of event handlers possible. The following example registers the function named doSomething as an onclick event handler to the element with id "id1"
   ```
   <script type="text/javascript">
   document.getElementById("id1").onclick = doSomething;
   </script>
   ```

3. **Using DOM Event Specification:** The most recent and advanced way of event registration is through using the event registration model that is introduced in DOM Level 2 specification. Unlike the previous methods, which only allow a single event handler for each type of event, in this model, any number of event handlers can be added for each type of event using the `addEventListener` method via JavaScript. (Microsoft implements a slightly different version of this model in IE.) When multiple event handlers are registered to an event type, all the registered event handlers are run one after the other when the event occurs. The following example registers two event handlers to the onclick

event of the element with id "id1".

```
<script type="text/javascript">
var element = document.getElementById("id1");
element.addEventListener = ("click", doSomething);
element.addEventListener = ("click", doAnotherThing);
</script>
```

In this example, when the element is clicked both JavaScript functions, `doSomething` and `doAnotherThing`, will be executed. In this model, it is also possible to remove a previously registered event handler using the `removeEventListener` method.

1.2 Implementation

Our crawler supports all three types of event registrations methods mentioned. AppScan component is able to provide us the set of DOM elements that have registered event handlers. The event handlers that are registered using method 1 is easily detectable since they are part of the HTML. For DOM elements that have events registered using methods 2-3, AppScan uses its JavaScript Engine to keep track of which DOM elements have registered event handlers.

Once all the elements with event handlers are known, an event identifier needs to be generated for each element and event type. In the current implementation the event identifier is a string that consists of a DOM element identifier and an event type identifier. As the DOM element identifier, we are using the string representation of the HTML element and its descendants (retrieved using the outerHTML property of the element) and the type of event. For example, for the following HTML anchor element which has an event handler registered using `href` attribute

```
<a href="javascript:‘doSomething()’"><img src="something.jpg" /></a>
```

the produced event identifier looks like

```
<a href="javascript:‘doSomething()’"><img src="something.jpg" /></a>~~HREF
```

where `~~` is a delimiter separating the DOM element identifier and the event type identifier. The event type identifier `HREF` shows that the element has an event registered to it using its `href` attribute. In this example, the event handler `doSomething()` information is part of the DOM element identifier.

For event handlers registered through JavaScript (methods 2 and 3) the event identifier also contains the hash of the event handler(hash of the definition of the javascript function) For event handlers registered through HTML

(method 1), the event handler information is already part of the DOM element identifier as we have seen in the example above.

1.2.1 Event Types

Our crawler currently supports a subset of the mouse events and the href events (which are also mouse events since they are triggered when an anchor is clicked). The mouse events that are considered are mouseover, mouseenter, mousedown, mouseup, click, dblclick, mouseout and mouseleave. Instead of considering all these mouse events as individual events we consider them as a single composite event that we call as GroupedMouseEvent. That is, if an element has any subset of the mentioned mouse events enabled, our crawler executes the corresponding event handlers in sequence as written above. The main reason for doing this is to better simulate a user's behaivour. When a human user wants to click on an element in the browser, she has to first move the mouse over to the element which will trigger the mouseover and mouseenter handlers and then she will be able to click it. In addition, a mouse click event handler is only triggered after the mousedown and mouseup handlers are executed. Similarly a double click is only triggered after the click event handler. It is also a typical user behavior to move the mouse away from the element once it is clicked. For this reason the sequence ends with running any mouseout and mouseleave event handlers.

In the near future, we are planning to introduce a new mouse event sequence in addition to the mentioned sequence. The new sequence will exclude the dblclick event handlers from the current sequence. Thus we will consider the double click and click as different events. (Until now none of our test websites had double click event handlers, so not considering them separately did not have any effect). Also the crawler currently considers only the left mouse button clicks (middle and right clicks are currently ignored). More complex mouse gestures (such as drag and drop) and key strokes are also not supported. Such event types could be included in the future.

2 DOM Equivalence

For DOM Equivalence, our crawler uses AppScan's DOM Equivalence algorithm [1] with a slight modification. The original algorithm implemented in AppScan produces an identifier for a given DOM by only considering the

underlying HTML structure without taking into account the events enabled in the DOM. Since this algorithm only considers the HTML structure, in the remainder we refer to it identifier produced by this algorithm as HTML ID. We believe it is important to make sure that the DOMs in the same equivalence class should have the same set of enabled events. For this reason, our DOM identifiers are the combination of the HTML ID produced by AppScan and an identifier generated for the set of enabled events in the DOM. The latter identifier is simply produced by first sorting the set of event identifiers enabled in the DOM into a list and then concatenating the individual event identifiers in the list.

2.1 Computing the HTML ID

The algorithm aims at identifying the pages with similar page structure by reducing the repeating patterns in a given HTML to reach a canonical representation of the HTML document such that the canonical representation will be the same for any other structurally equivalent document. The motivation comes from the observation that HTML pages often contain repeating patterns, especially when the content is listed using HTML tables and lists. For example, Figure 2 shows the rendering of an HTML table in the browser on the left and the corresponding HTML body on the right. As we can see, the HTML table body contains rows, <tr> elements, and each row contains columns, <td> elements.

When all the text and attributes are stripped from the HTML (leaving only the HTML tags), the subtree rooted at <tbody> looks like

```
<tbody>
    <tr><td></td><td><a></a></td><td></td></tr>
    <tr><td></td><td><a></a></td><td></td></tr>
    <tr><td></td><td><a></a></td><td></td></tr>
</tbody>
```

where each row <tr> follows the same pattern. The algorithm recognizes such patterns and reduces them. In this example the subtree would be like the following after the reduction.

```
<tbody>
    <tr><td></td><td><a></a></td><td></td></tr>
</tbody>
```

By eliminating such repetitions the algorithm considers two pages as equivalents when they have the same pattern repeated different number of times,

Figure 2: A page containing a table (left) and the body of the corresponding HTML document (right)

everything else being the same. For this example, the produced identifier will not change when the number of rows is changed in the table. In addition, once all the reductions are done in a subtree, the algorithm sorts the remaining tags so that the algorithm is not affected by reordering of the elements in the page.

The algorithm allows the user to configure which HTML tags and attributes to consider as well as whether to include the text content for the computation of the HTML ID. (The configuration we used for the experimental study is to include the text content, all HTML tags and none of the attributes). When provided an HTML page, the algorithm produces the identifier by applying the following steps:

1. The HTML is stripped out of anything that is not included in the user configuration.

2. Algorithm identifies a parent node whose children are all leaf nodes in the tree.

3. Algorithm traverses the leaf nodes and at each leaf node, it checks if the sequence of already traversed leaves and the current leaf node forms

a pattern. A pattern is detected if the sequence contains consecutive repeating elements. For example the sequence
<A><B><C><A><B><C>
contains the consecutive repeating pattern <A><B><C> whereas
<A><B><C><D><A><B> has no consecutive repeating pattern. Although <A><B> is repeated, the repetition is not consecutive.

4. When such a repeating pattern is detected, all the repetitions are eliminated.

5. When the last leaf node of the parent is processed, the reduced sequence is sorted and the parent node is turned into a leaf node containing the reduced sequence. For example, when the last leaf in the leaf node sequence <B><A><C><B><A><C> of the parent <Parent> is processed, the result would be a new leaf node
<Parent><A><B><C></Parent>
(i.e. a leaf node <Parent> with text "<A><B><C>").

6. Steps 2-5 are repeated until the stripped HTML is reduced to a single node. At this point the resulting node uniquely identifies the equivalence class of the HTML page. By hashing the content of the node, the HTML ID is produced.

References

[1] K.A. Ayoub, H. Aly, and J.M Walsh. Dom based page uniqueness identification. http://ip.com/patapp/CA2706743A1, 2010. [Online].

[2] IBM. IBM Security AppScan family. http://www-01.ibm.com/software/awdtools/appscan/. [Online].