Automated Security Testing of Web Widget Interactions

Cor-Paul Bezemer
Delft Univ. of Technology
& Exact Software
The Netherlands
c.bezemer@tudelft.nl

Ali Mesbah
Delft Univ. of Technology
The Netherlands
A.Mesbah@tudelft.nl

Arie van Deursen
Delft Univ. of Technology
The Netherlands
Arie.vanDeursen@tudelft.nl

ABSTRACT

We present a technique for automatically detecting security vulnerabilities in client-side self-contained components, called web widgets, that can co-exist independently on a single web page. In this paper we focus on two security scenarios, namely the case in which (1) a malicious widget changes the content (DOM) of another widget, and (2) a widget steals data from another widget and sends it to the server via an HTTP request. We propose a dynamic analysis approach for automatically executing the web application and analyzing the runtime changes in the user interface, as well as the outgoing HTTP calls, to detect inter-widget interaction violations.

Our approach, implemented in a number of open source Atusa plugins, called Diva, requires no modification of application code, and has few false positives. We discuss the results of an empirical evaluation of the violation revealing capabilities, performance, and scalability of our approach, by means of two case studies, on the Exact Widget Framework and Pageflakes, a commercial, widely used widget framework.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Reliability, Security, Verification

Keywords

Web applications, security testing

1. INTRODUCTION

Web applications increasingly rely on client-side processing to enable a user experience comparable to that of desktop applications. Using Ajax [2], the classic sequence of static HTML pages is replaced by a single page, which is dynamically adjusted at the client side by JavaScript code that directly manipulates the browser's DOM tree. Communication with the server is asynchronous, and typically affects small parts of the DOM tree only [13].

Furthermore, Ajax pages can be composed from independent user interface components, often called web *widgets*. These widgets are mini-applications composed of a chunk of code that can be embedded in an HTML page, and run independently and next to each other, providing dynamic content and functionality for a wide variety of tasks such as showing the latest news headlines, weather predictions, or a list of new email messages. Highly visible examples include Mashup sites such as iGoogle,[1] Netvibes,[2] and Pageflakes,[3] which allow users to select widgets from a catalog, and customize and host them on their own start page. They also provide APIs for external developers to build and include new widgets in the widget catalogs.

As any program code, widgets can be used for malicious purposes. Security becomes an important aspect when third-parties are allowed to include new widgets in public catalogs. Example scenarios include when a malicious widget changes the content of another widget to trick the user into releasing sensitive information, or even worse, listens to the account details a user enters in another widget (e.g., PayPal or Email widgets) and sends the data to a malicious site.

Current protection-based solutions aim at avoiding the problem by disallowing inter-widget interactions, at the cost of limiting functionality. Traditional detection-based approaches are generally static analysis-based, which has limitations in revealing faults and violations in the distributed runtime behavior of modern rich web applications.

Testing modern web applications for security vulnerabilities is far from trivial. In this paper, we propose a new dynamic analysis approach that automatically detects and reports inter-widget interaction violations. We have extended and used our Ajax testing tool Atusa [14], to implement the approach in a number of security plugins, called Diva. In addition, we present the results of an evaluation, performed to assess the effectiveness and performance of our approach. The results of our case studies performed on two external widget frameworks are positive; We believe the applicability of our approach is not limited to Ajax-based widgets, and can be used to test the security of any web application in which inter-widget interactions pose security threats.

This paper is further organized as follows. In the next section, we discuss two types of security vulnerabilities that

[1] `http://www.igoogle.com`
[2] `http://www.netvibes.com`
[3] `http://www.pageflakes.com`

we focus on in this paper. In Section 3, we present our overall approach to automatically detect inter-widget security violations. In Sections 4 and 5 we describe the details of the automated detection methods. Section 6 covers the implementation of the methods and Section 7 reports on the results of our evaluation with two case studies. We conclude our paper with a discussion, related work, and concluding remarks.

2. BACKGROUND

Placing interactive applications from multiple external vendors on a single web page requires a framework and a collection of widgets. The framework offers APIs and guidelines for developers to create new widgets. Each implemented widget must adhere to the framework guidelines, and can be hooked into a framework web page as a subtree, and executed without requiring additional compilation, in a mobile code style [1].

The traditional way of placing content from multiple external vendors on one page is by using `IFrames`. The advantage of `IFrames` is that they provide an isolated environment for the widgets because the Same Origin Policy (SOP) [15] constrains them from accessing other parts of the page. The main problem with using `IFrames` is, however, that they do not provide real integration on the page. `IFrames` impose undesired limitations, for instance on, the page layout, use of CSS style sheets of the framework, and rich widget interactions such as resizing.

An alternative for overcoming such limitations is to place each widget inside a `DIV` container [18]. The trade-off of using `DIV` containers is the lack of SOP support. Hence, widgets that are placed in `DIV` containers run in the same execution environment and are, therefore, capable of accessing and manipulating other widgets' properties. This leaves the developers with the difficult choice between security and functionality. Existing widget frameworks appear to be avoiding the security problems by either completely disallowing inter-widget interaction, or putting the responsibility of possible risks of using external widgets on the shoulders of the end users.

Inter-widget interactions are desirable because they can provide highly interactive functionality, by for instance allowing to update a shopping basket widget, when an item is selected in a product items widget. Therefore, placement in `DIV` containers is preferred over `IFrames`.

Various violation scenarios in `DIV`-based malicious widgets can be sketched. In this paper, we focus on the following two generalized types:

V1 A malicious widget (MAL) changes the DOM subtree of another innocent widget (VIC), e.g., MAL changes the action URL of a form in VIC so that when the user submits the form, the content is sent to a malicious site.

V2 MAL reads data from VIC and sends it to a server using an HTTP request, e.g., user logs into an email account using VIC, MAL logs the entered username and password and sends it to a remote server.

Although security testing is a broad term, current web application security research is mainly focused on preventing and detecting Cross-site Scripting (XSS) and SQL injections. However, we believe that V1 and V2 cover a wide range of security violation scenarios that are increasingly becoming important as they threaten the safety of modern interactive web applications, in which widgets are `DIV`-based rather than `IFrame`-based and require dedicated attention.

3. OUR APPROACH

Web widgets are generally placed on a single-page web interface [13] using AJAX technology. The goal of our approach is to automatically identify security vulnerabilities in web widgets. To achieve this goal, our approach performs a dynamic analysis of the AJAX web application, by crawling through the various states and analyzing the widgets' behavior.

In our previous work [12], we proposed a new type of web crawler, called CRAWLJAX, capable of exercising client-side code, detecting and executing doorways (clickables) to various dynamic states of AJAX-based applications within browser's dynamically built DOM. While crawling, CRAWLJAX infers a state-flow graph capturing the states of the user interface, and the possible event-based transitions between them, by analyzing the DOM before and after firing an event.

More recently, we proposed an approach for automatic testing of AJAX user interfaces, called ATUSA [14]. ATUSA is based on the crawling capabilities of CRAWLJAX and provides data-entry point detection and (pre-, in-, and post-crawling) plugin hooks for testing AJAX applications through generic and application-specific invariants that serve as oracle to detect faults.

In this paper we propose to extend and use ATUSA for automatically spotting security problems in widget interactions. For each generalized violation scenario (V1, V2), we first present our method for detecting the violation and then discuss the plugin that implements the approach.

In order to find DOM change violations (V1), we first need to automatically detect each widget's boundary in the DOM tree. Once the boundaries are defined, we can analyze the elements receiving events and the actual changes taking place on the DOM tree to decide whether a state change is a violation.

For HTTP request violations (V2), the main challenge is in coupling each outgoing request with the corresponding DOM element, from which it originated. Once we know which element is causing the request, we can analyze the behavior and decide whether a violation has occurred.

To detect the widget in which the violation was initiated, we need to consider ways in which the state of a web page can be changed. In AJAX-based applications, the web page usually changes as a result of an event fired on an element with an event listener. Note that performing a full page refresh updates the page as well but is rarely required in current single-page web interfaces. In our approach, we assume a violation is initiated by an event (e.g., `click, mouseover, dblclick`) fired on a DOM element.

In the next two sections, our methods for detecting V1 and V2 are presented respectively.

4. DETECTING INTER-WIDGET CHANGE VIOLATIONS

The first violation scenario that we address involves a malicious widget MAL trying to make a change to the DOM subtree belonging to an innocent widget VIC. Below, we

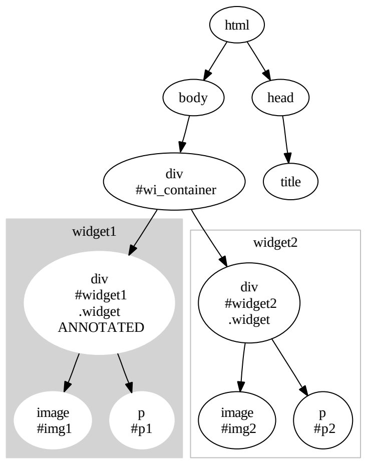

Figure 1: Example DOM tree of a widget framework (# = id, . = class).

first discuss the nature of widget boundaries, after which we explain how we can reverse engineer them, and how these boundaries can be used to detect illegal DOM changes.

4.1 Widget Boundaries

A widget is a subtree of the DOM. Widgets cannot be nested, and do not share DOM elements. The top (root) element of a widget comprises its boundary with the top level framework DOM elements.

An example is shown in Figure 1: Two widgets are displayed, with root elements having id's `#widget1` and `#widget2`. The framework element is the parent of these two widgets, having id `#wi_container`.

In many frameworks, it is straightforward to identify the boundary between a widget and the framework part of a DOM tree, often by means of the `class=widget` attribute. With the goal of defining a standard for automated widget testing, we are an advocate of this policy. For example, in Figure 1, the root of each widget holds a class attribute `.widget`.

Not all frameworks, however, require this explicit identification. In those cases, a number of heuristics can be used to determine if a particular DOM node constitutes a widget boundary. First, some elements are certain not to belong to any boundary. Typical examples include `HTML` or `BODY` elements, or nodes that are known to belong to the given framework (the `#wi_container` in the example).

In addition to that, some elements can be given a certain probability that they constitute a boundary. An example is a `DIV` element, which can, but need not be a widget boundary. These probabilities are defined by the tester after manual inspection of the DOM.

4.2 Reverse Engineering Boundaries

In order to identify widget boundaries automatically, we propose the use of *boundary rules*. Each rule provides a *pattern* as well as a rating. Example rules for Figure 1 are displayed in Figure 2.

The pattern syntax is equal to the pattern used to configure the elements to crawl in CRAWLJAX [14], resulting in rules of the form *`tag:{attr=value}|rating`*.

The rating is an indicator for the likelihood that a given pattern will match a boundary. Possible values include `LOW`, `MEDIUM`, `HIGH` and `VERY_HIGH`, as well as `EXCLUDE`. The latter option indicates that a node mating a given pattern cannot be a widget boundary.

```
html:{}|EXCLUDE; body:{}|EXCLUDE; div:{}|LOW;
div:{class=widget}|VERY_HIGH; threshold = MEDIUM
```

Figure 2: Widget boundary rules for the example widget (separated by ;).

Algorithm 1 IdentifyWidgetBoundary(fingerNode)

```
1: $rules \leftarrow$ getIdentificationRules()
2: $node \leftarrow$ fingerNode
3: $bestMatchedNode \leftarrow$ null
4: $highestRating \leftarrow$ null
5: while $node \neq$ null do
6:     $r \leftarrow$ node.match(rules)
7:     if $r \neq$ EXCLUDE && $r > highestRating$ then
8:         $bestMatchedNode \leftarrow node$
9:         $highestRating \leftarrow r$
10:    end if
11:    if $r =$ VERY_HIGH then
12:        break
13:    end if
14:    $node \leftarrow node$.getParent()
15: end while
16: if $highestRating \geq$ getThreshold() then
17:     $bestMatchedNode$.annotate()
18:     return $bestMatchedNode$
19: end if
20: return null
```

In order to determine the widget boundary for a given node n, we traverse the tree upwards from n to the root of the full DOM tree. Every ancestor we encounter on our way up we match with the given rules. The node with the highest ranking is marked as widget boundary, provided the ranking is higher than a minimal threshold. The corresponding algorithm is displayed in Algorithm 1.

4.3 Detecting Violations

The final step is using the widget boundaries to identify DOM Change Violations. To that end, we need to identify elements triggering a change, and elements affected by a change. We define an *active element* as the element on which the last event was fired.

The widget boundary of the widget that initiated the DOM change can be automatically detected using Algorithm 1 with the active element as input.

To detect the widget in which the DOM change took place, first any changed node(s) must be determined by comparing the DOM tree before firing the event with the DOM tree after firing the event. After detecting the node(s) in which the DOM change took place, an analysis of the DOM change can be carried out. Assuming the DOM is successfully annotated during the detection of the widget boundary of the active element, Algorithm 2 can be used to automatically analyze the DOM changes.

5. DETECTING REQUEST VIOLATIONS

Our second violation scenario describes the generalized situation in which a widget reads data from another widget, and sends it to a remote server using HTTP requests. We first offer some examples illustrating this violation (Section 5.1). Then, we explain how we can capture the widget from which a request originates (Section 5.2), and how we can use that to compare the request to a list of trusted re-

Algorithm 2 IsDomChangeViolation(changedNode)

```
Require: The DOM tree is annotated successfully during the widget
    boundary detection of the active element.
1: $node \leftarrow changedNode$
2: while $node \neq$ null do
3:     if $node$.isAnnotated() then
4:         return false
5:     end if
6:     $node \leftarrow node$.getParent()
7: end while
8: return true
```

quests for the given widget (Section 5.3). In addition, we explain how these lists can be generated automatically.

5.1 Request Violation Examples

A simplified example of JavaScript code that causes an HTTP request violation is given in Figure 3.

```
// code in MAL
url = "http://www.domain.com/mal_script.aspx";
vic = document.getElementById("VIC.inputField");
vic.onkeydown = sendData(url, event.getKey());
```

Figure 3: MAL attaches an event listener to VIC, sending every user input in VIC to a remote server.

It is important to realize that the attachment of the `onkeydown` event handler does not affect the DOM tree of VIC and, therefore, can not be detected using Algorithm 2. In this example, VIC makes an HTTP request every time a key is pressed in its input field, which, based on VIC's original functionality, could be a violation. A second example is given in Figure 4, in which the event which triggers the request is attached to MAL itself. Because MAL reads data from VIC, this is also a violation.

```
// code in MAL
function getValue() {
 return document.
  getElementById("VIC.inputField").value;
}

url = "http://www.domain.com/mal_script.aspx";
mal = document.getElementById("MAL.button");
mal.onclick = sendData(url, getValue());
```

Figure 4: MAL reads data in VIC and sends it to a remote server through an HTTP request.

This type of violation is very difficult to detect by static analysis, because an event handler can be generated and attached to an element at runtime in JavaScript. In addition, it is difficult to detect by which widget an event handler was attached to an element. Our approach uses dynamic analysis and assumes that MAL has to send the data read from VIC to a remote server. Otherwise, the data never leaves the client and hence no real violation can occur. We detect HTTP request violations by analyzing all outgoing HTTP requests.

5.2 HTTP Request Origin Identification

The main challenge of detecting the origin widget of a request is to couple the request to the DOM element from which it originated. This is not a trivial task, since HTTP requests do not carry information about the element that triggered the request. To be able to analyze HTTP requests, all requests must be intercepted. For this purpose, we propose to place an HTTP proxy between the client browser and the server, which buffers all outgoing HTTP requests.

```
// before annotation
<script src="javascript.js"></script>
// after annotation
<script src="javascript.js?requestForProxyId=123"
        requestForProxyId="123"></script>
```

Figure 5: Example annotation of the unique attribute annotation proxy plugin.

The only way to attach information about DOM elements to an HTTP request, without affecting the behavior of the web server handling the request, is by adding data to the request query string (e.g., `?wid=w23&requestForProxyId=123`). This data should be selected carefully, to ensure it does not interfere with other parameters being sent to the server. If the request parameters contain the value of a unique attribute, such as the element's ID, it can be extracted and used to identify the element in the DOM. Enforcing all HTTP requests to contain a value with which the origin widget can be detected requires having mechanisms for the enforcement of a unique attribute in each DOM element, and the attachment of the unique attribute of the originating element to outgoing requests.

First we need to consider ways HTTP requests can be triggered in Ajax-based web applications.

Static Elements. HTTP requests triggered by the `src` attribute of an static element, for instance in a `SCRIPT` or `IMG` element in the source code of the HTML page, are sent immediately when the browser parses them. This leaves us no time to dynamically annotate a unique value on these elements, as the requests are sent before we can access the DOM. The solution we propose is to use the proxy for intercepting responses as well. The responses can be adjusted by the proxy to ensure that each element with a `src` attribute is given a unique identifying attribute.

Figure 5 shows an example of annotating a `requestForProxyId` attribute on a `SCRIPT` element. Note that the attribute is annotated twice: in the URL so that it reaches the proxy, and as an attribute for easy identification on the DOM tree using XPath when the violation validation process is carried out.

Dynamic Elements. The `src` attribute of an element that is dynamically created on the client through JavaScript and added to the DOM tree, can also trigger an HTTP request. Annotating attributes through the proxy has limitations for this type of request, since elements that are added dynamically on the client-side are missed. During dynamic annotation these elements are missed as well, because the request is triggered before the element can be annotated. Because we assume every element has a unique attribute in our approach, requests triggered from dynamically generated elements can be detected easily as they do not contain a unique attribute. We believe dynamically generated elements with a `src` attribute are rare in modern web applications, and since this attribute should point to, for instance, a JavaScript file or image, the HTTP request they trigger should be easy to verify manually by a tester. Therefore, all requests made from elements which are not annotated, should be flagged as suspicious and inspected by the tester.

Algorithm 3 IsHttpRequestViolation(request)

```
Require: Access to the DOM history list.
Require: Elements in the DOM are annotated with a unique attribute.
Require: Request contains the annotated attribute of the DOM element.
1: $id \leftarrow request$.getQueryString().getElementId()
2: $activeNode \leftarrow domHistory$.findElement($id$)
3: $widget \leftarrow$ IdentifyWidgetBoundary($activeNode$)
4: $list \leftarrow widget$.getAllowedUrlList()
5: $url \leftarrow request$.getUrl()
6: return !$list$.contains($url$) || isSuspicious($url$)
```

Ajax Calls. HTTP requests sent through an Ajax call, via the `XMLHttpRequest` object, are the most essential form of sending HTTP requests in modern single-page web applications [2]. These requests are often triggered by an event, e.g., *click, mouseover*, on an element with the corresponding event listener. Note that this type of elements could also be created dynamically, and therefore proxy annotation is not desirable. Hence, we propose to dynamically annotate such elements. To that end, we annotate a unique attribute on the element right before an event is fired. Note that this annotation is easiest to implement by means of aspects, as explained in Section 6.

After the annotation, the attribute (and its value) must be appended to all HTTP requests that the event triggers. To that end, we take advantage of a technique known as *Prototype Hijacking*[17], in which the Ajax call responsible for client/server communication can be subverted using a wrapper function around the `XMLHttpRequest` object. During the subversion, we can use the annotated attribute of the element, on which the event initiating the call was fired, to add a parameter to the query string of the Ajax HTTP call.

It is possible that the annotated origin element is removed from the DOM by the time the request is validated. To avoid this problem, we keep track of the DOM history. After an event is fired, and a DOM change is occurred, the state is saved in the history list. Assuming the history size is large enough, a request can always be coupled to its origin element, and the state from which it was triggered, by searching the DOM history.

5.3 Trusted Requests

After detecting the origin widget of a request, the request must be validated to verify whether the widget was allowed to send this request. To this end, a method must be defined for specifying which requests a widget is allowed to make.

Our approach uses an idea often applied in firewall technology, in which each application has an allowed list of URLs [10]. For each widget, we can automatically create a list of allowed URLs by crawling it in an isolated environment.

This way, every request intercepted by the proxy can be assigned to that specific widget. At the end of the crawling process, the proxy buffer contains all the requests the widget has triggered. This list can be saved, edited by the tester, and retrieved during the validation phase of a request. In addition, it is possible for a tester to manually flag URLs in the list as suspicious. If during the validation process a request URL does not exist in the allowed URL list of its origin widget, or if the URL is flagged as suspicious, we assume the widget does not have permission to trigger the request and thus an HTTP request violation has occurred.

Assuming a request contains the annotated attribute of the origin element, Algorithm 3 can be used to automatically detect the *origin widget* of the request and report HTTP request violations.

Note that this approach also works for requests that do not originate from a widget, but from a non-widget element instead. By crawling the framework with only an empty widget, an allowed URL list can be created for the framework. A request which originates from an element that does not have a widget boundary will be validated against the allowed URL list of the overall framework.

6. IMPLEMENTATION

We have implemented our methods for detecting DOM change violations and HTTP request violations as two separate plugins, called Diva (Detecting Inter-widget Violations with Atusa), for our Atusa Ajax testing and crawling infrastructure. Diva is available through our web site[4] as an open source project.

The full architectural processing view, of the crawler as well as the security plugins, is shown in Figure 6. Two *in-Crawling* plugins are shown in particular: *DOM Violations*, and *HTTP violations*. Furthermore, it shows the integrated proxy between the embedded browser and the web server. The implementation details of Atusa and the underlying Ajax crawler, Crawljax, and the role of the remaining components such as the robot, state machine, controller, and DOM analyzer are explained in [14, 12].

The first issue to consider for the implementation of Diva is the way Atusa handles the DOM. Atusa uses three DOM objects:

- The *realDom* - The runtime internal browser DOM,
- The *domBeforeEvent* - A copy of the *realDom* before the event is fired,
- The *domAfterEvent* - A copy of the *realDom* after the event is fired.

The DOM change violation detection plugin compares the *domBeforeEvent* and the *domAfterEvent* using the Diff function of XmlUnit[5] to find the changed nodes. XmlUnit returns a list of references to nodes in the *domAfterEvent* object, while the crawler returns a reference to the *active element* in the *domBeforeEvent* object. As mentioned earlier, our approach requires both references to be in the same object. Therefore, we extended Atusa using AspectJ to annotate the widget boundary of the active element in the *realDom* with a unique attribute, right before it fires an event on it. We have chosen to annotate the widget boundary rather than the active element itself, because we assume widgets cannot be removed during the crawling session. This implies that the widget boundary is available in every DOM state, while the active element may be removed or changed. When *realDom* is copied into *domAfterEvent*, the annotation is copied along and the active element can be retrieved using an XPath query for the annotated unique attribute. Using this approach, references to the widget boundary of the active element and the changed nodes can be found in the same DOM object.

The minimum rating an element must match, to be included in the detection process, can be configured. It is also

[4] `http://spci.st.ewi.tudelft.nl`

[5] `http://xmlunit.sourceforge.net`

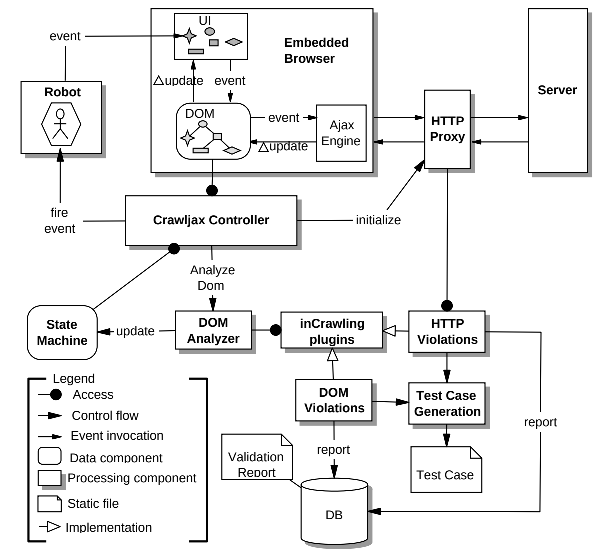

Figure 6: Processing view of our security violation detection approach.

possible to use wildcards in the value specification of a rule, e.g. `div:{id=widget%}`.

For dynamic injections as described in Section 5.2, Atusa was adapted, through aspects, to inject a unique attribute into the element on which the next event will be fired, right before the event is fired.

For our request violation detection approach, we have integrated an HTTP proxy into Atusa. We use the proxy functionality of WebScarab,[6] which is an open source Java proxy. Our integration allows initializing and accessing the proxy through the Crawljax controller.

Di Paola and Fedon [17] have proposed a method for subverting Ajax calls by diverting the `XmlHttpRequest`, called *Prototype Hijacking*. Unfortunately, this approach does not work in Internet Explorer since the browser does not allow prototype manipulation of the XmlHttpRequest object. To overcome this limitation, we subvert an instance of the object instead of the prototype itself. This solution is, however, not generic, i.e., depending on the underlying Ajax framework and the way the calls are made, different subversion code is required. We have currently implemented diversion code for the ASP.NET and jQuery Ajax frameworks. The JavaScript wrapper code is automatically added to the page trough the proxy.

The output of Diva consists of a description of the violation and the event sequence which has caused it. The description of a DOM change violation contains a textual representation of the DOM change found, generated by XmlUnit. The description of an HTTP request violation contains information about the origin widget, the requested URL, and the name of the list against which it was checked. In addition, a Selenium[7] test case is generated for each detected violation automatically. These test cases can be used by the tester to replay and investigate the detected violations manually.

[6] `http://www.owasp.org/index.php/Category:OWASP_WebScarab_Project`

[7] `http://seleniumhq.org`

Widget pair	DOM change in VIC	Description
D1	Form action URL	An event fired in MAL changes the specified element in VIC
D2	Link description	
D3	Link destination	
D4	Background colour	
D5	Image source	
D6	Framework element	An event fired in MAL changes a framework element, e.g. the `href` of a menu item
Widget pair	**Event attached to VIC**	**Description**
H1	Key press	MAL attaches an event handler, which sends an HTTP request, for the specified event to an element in VIC
H2	Focus	
H3	Submit	
H4	Click	
H5	Mouse movement	
Widget pairs	**Description**	
D*	All DOM violations on one page	
H*	All HTTP violations on one page	
DH*	All DOM and HTTP violations on one page	

Table 1: Widget pairs causing DOM change and HTTP request violations by MAL.

7. EMPIRICAL EVALUATION

To assess the the usefulness and effectiveness of our approach and the corresponding implemented plugins, we conducted two case studies following Yin's guidelines [21]. Our evaluation addresses the following research questions:

RQ1 What are the violation revealing capabilities of the two approaches for DOM change and HTTP request violation detection?

RQ2 How well does our analysis perform, with respect to crawling time and the number of states?

RQ3 How scalable is our approach, with respect to the number of widgets that can be verified at the same time?

7.1 Subject Systems

Our experimental subject systems consist of EWF (Exact Widget Framework) and Pageflakes.

EWF. Exact Software[8] is currently researching the possibilities of building software solutions using widgets. As a proof of concept, a widget framework is being prototyped, which is referred to as EWF throughout this paper. One of the key components of the EWF will be the personalized start page for each user, on which widgets from a catalog can be placed using a drag and drop mechanism. The prototype Ajax-based widget framework is implemented by the Research and Innovation (R&I) team of Exact Software in ASP.NET and runs on IIS 7.

Pageflakes. Pageflakes is a commercial Ajax-based start page, which allows its users to place widgets on their personalized pages. Pageflakes is a closed source framework, which means we are not able to inspect or adapt the framework code. In Pageflakes, widgets are called flakes and at the time of writing over 240,000 flakes were available. Pageflakes also provides APIs for external developers to create their own widgets.

Note that both EWF and Pageflakes use `DIV`-based containers for their widgets.

7.2 Setup

We designed a number of scenarios, each consisting of two widgets, a malicious widget (MAL) and a victim widget (VIC). Two groups of widget pairs were designed, as shown in Table 1. The first group contained six cases (D1-D6),

[8] `http://www.exactsoftware.com`

each causing a DOM change violation in VIC by MAL, e.g., an event fired on MAL changes the action URL of a form, or the background color of an element in VIC. The second group contained five cases (H1-H5), each causing an HTTP request violation, e.g., MAL attaches an event listener to VIC so that the contents of a form field in VIC are sent to the server by MAL after a `keypress` event.

Both studies were performed on an Intel Core 2 6400 2.13 Ghz CPU with 2 GB RAM running Windows XP. We configured Atusa properties to use Internet Explorer 7 as the internal browser, included the `A, Input, Img, Button, Label` tags for crawling, and the `click, mouseover, mousedown, mousemove, mouseup, blur, focus, keydown, keypress, keyup`, and `select` as the possible events for the robot.

In order to test the violation revealing capabilities of our approach (RQ1), we measure the number of violations seeded and detected. We also measure the number of false positives to test the reliability of the output of our plugins. To test the performance of our approach (RQ2), we measure the crawling time for each case, and the number of clickables and DOM states found by Crawljax. The set of clickables contains all elements which were not explicitly excluded by configuration and on which firing an event resulted in a new DOM state [12]. Finally, to test the scalability of our approach (RQ3), we place multiple widget pairs on the page at the same time (case D*, H* and DH* in Table 1).

We have configured the maximum crawling time for each case at 60 minutes. If the crawling process was not finished by that time, the execution was stopped and the output was analyzed.

7.3 Execution

EWF. To prevent bias, 11 EWF widget pairs, including their malicious behaviour, were implemented by a member of the Exact R&I team, following the requirements in Table 1. Because our focus was on inter-widget interactions, all elements in the menu section and elements which may remove or hide a widget were excluded from the crawling process. We have manually determined the items to exclude by inspecting the DOM, which could be done within 30 minutes. We defined the rules for EWF widget boundary identification (e.g., `div:{class=ex widget}|VERY_HIGH`).

For each test run, a widget pair was manually placed on the page, after which, we ran Atusa on the personalized page. Different events were then fired on the widgets automatically and every time a change was detected, our plugins analyzed the DOM tree and the HTTP requests to detect security violations. To test our approach with multiple widgets on a page, we also carried out the test with all widget pairs (DH*) on the same page.

Pageflakes. For the second subject, we implemented the same scenarios, D1-H5, using the Pageflakes API, which provides convenience JavaScript methods for event management and DOM manipulation for creating *flakes*. After defining the widget boundary identification rules (e.g., `div:{class=flake}|VERY_HIGH`), the tests were run using the same setup as for EWF. Flake pairs were manually placed on the page, after which they were automatically tested through Atusa.

In both cases, for widget pairs D1 to D6, the DOM change violation detection plugin was enabled, and for widget pairs H1 to H5, the HTTP request violation detection plugin. During the execution of all widget pairs (DH*) both plugins were enabled.

Widget pair(s)	Violations seeded	Violations detected	False Positives	Time (s) w.o. plugins	Time (s) w. plugins	Overhead time (%)	Clickables	States
D1	3	3	0	122	110	-9.8	3	4
D2	3	3	0	106	90	-15.0	3	4
D3	3	3	0	107	93	-13.0	3	4
D4	3	3	0	106	90	-15.0	3	4
D5	3	3	0	175	159	-9.1	3	4
D6	4	4	0	126	110	-12.7	4	5
D*	19	19	0	552	538	-2.5	19	20
H1	1	2	1	74	75	1.4	3	4
H2	1	2	1	55	55	0.0	2	3
H3	-	-	-	-	-	-	-	-
H4	1	2	1	51	51	0.0	1	2
H5	1	2	1	62	61	-1.6	4	5
H*	4	8	4	156	158	1.3	10	11
DH*	23	43	10	703	760	8.1	29	30

Table 2: Results of the EWF case study.

7.4 Results

Tables 2 and 3 show our measurements for the EWF and Pageflakes case studies, respectively. The scenario with all widgets on the page (DH*) required more crawling time on Pageflakes than the maximum time allowed (annotated with * in Table 3).

The first run on the EWF case reported a large number of false positives. After inspection, we found that the used jQuery UI[9] library in EWF, attaches `onmousedown` event handlers to many elements on the page that cause DOM changes for each fired event. To avoid these unnecessary false positives, Atusa was adapted to filter DOM changes of this type. During the second run, we found that widget pair H3 could not be validated due to invalid HTML code, which caused a problem after the JavaScript subversion code injection. During this process, the HTML is parsed, the code is added, and serialized. Therefore, a requirement of Diva is that widgets contain valid HTML. Our proposal is to run the regular HTML validity checker plugins in Atusa first [14], and reject widgets that contain invalid HTML code.

Since Pageflakes does not allow the use of `<form>` elements, widget pair D1 could not be implemented for that subject system. This is also the reason that one violation less was seeded in widget pair D6.

RQ1. Going back to our first research question, RQ1, the measurements from the two tables show that our approach can successfully detect DOM change violations, D1 through D6, without any false positives, on both EWF and Pageflakes.

The results of widget pairs H1 to H5 (excluding H3) show a false positive for each widget pair in the EWF case study. Closer inspection reveals that all these false positives were caused by an implementation characteristic of the EWF. In EWF, JavaScript code for each widget is loaded in the `BODY` section of the page rather than inside the corresponding widget. Therefore, although the JavaScript request will appear in a widget's allowed URL list, the request is coupled with the framework page rather than the widget. This framework behaviour violates the boundary of a widget and makes it impossible to decide whether a violation was intro-

[9] `http://jqueryui.com`

Widget pair(s)	Violations seeded	Violations detected	False Positives	Time (s) w.o. plugins	Time (s) w. plugins	Overhead time (%)	Clickables	States
D1	-	-	-	-	-	-	-	-
D2	3	3	0	263	230	-12.5	29	30
D3	3	3	0	107	223	118	29	30
D4	3	3	0	289	387	34	29	30
D5	3	3	0	358	368	2.8	29	30
D6	3	3	0	289	279	-3.5	30	31
D*	15	15	0	1676	2031	21.2	149	150
H1	1	1	0	161	158	-1.9	16	17
H2	1	1	0	179	178	-0.6	20	21
H3	-	-	-	-	-	-	-	-
H4	1	1	0	177	177	0.0	20	21
H5	1	1	0	175	179	2.3	20	21
H*	4	4	0	908	920	1.3	80	81
DH*	19	19	0	3600*	3600*	0.0*	161	162

Table 3: Results of the Pageflakes case study.

duced by a widget or framework action. Our proposal is to adjust the framework so that it respects the widget boundary. The same type of false positives are being recognized in the case with all widget pairs (DH*). The Pageflakes framework adds the code of a widget within the widget boundary. Hence, no such false positives for the Pageflakes case study were found. Based on these observations, we can conclude our approach is successful in detecting inter-widget interaction violations (V1, V2). In addition, for EWF, it was interesting to see that more violations were detected by our plugins than deliberately seeded in the test with all widgets on the page (DH*). This was because widget pairs H1 to H5 displayed a message after an HTTP request was successfully sent in EWF, which caused a DOM change violation. These violations were not detected sooner, because H1 to H5 were ran only with the HTTP request violation detection plugin enabled until then.

RQ2. To be able to answer RQ2, we measured the crawling time of widget pairs with and without plugins. In some cases, running ATUSA on widget pairs without plugins requires more crawling time than with plugins. One possible explanation could be that crawling involves firing events on the web application in the browser, and waiting for the response from the server. Possible variations in network and server processing latency could result in the observed oscillations. Therefore, the performance times should be seen as an indication only. Note that the results also indicate that our analysis of the DOM changes and HTTP request does not introduce much overhead and the whole process of crawling and analyzing the widgets can be carried out within a couple of minutes, on average.

RQ3. As far as RQ3 is concerned, we tested our approach with more than twenty widgets placed on the page, at the same time, and measured the number of detected violations and the required crawling time. The results show that all the seeded violations could be detected in a reasonable amount of time. However, it is important to consider that not all types of elements were included in the crawling process. Including all elements in the crawling process may add considerably to the crawling time.

8. DISCUSSION

Completeness. A security testing approach that is not complete, may have limited applicability, as it may miss many security violations while testing. The first fact to consider is that our approach uses a crawler. Therefore, if CRAWLJAX is not able to find a certain state, our approach is not able to test that state either. This problem is inherent to dynamic analysis approaches that use a crawler.

Our current implementation is capable of testing widgets which contain valid (X)HTML only. From a security perspective, this is a sound requirement. In our case it is imposed by the HTML serializer used during the JAVASCRIPT injection process, as explained in Section 7.4.

JAVASCRIPT timers can delay the effect of an event. Because we use the element on which the last event was fired to analyze a violation, it is important that effects of that event are applied before the next event is fired. If this is not the case, an effect may be coupled to another event than it was caused by, resulting in invalid analysis.

When a content proxy is used to make a request, i.e., a proxy on the server makes a request for a widget and forwards the response, it is difficult to detect an HTTP request violation when both widgets have the content proxy in their allowed URL lists.

Security. In addition to the completeness, it is important that a security approach itself cannot be bypassed easily. The first observation is that widgets execute mostly on the client-side of an application using JAVASCRIPT. Testing the client-side of AJAX applications is a challenging and new research area [14]. This is especially the case for testing client-side security of widgets. A possible security threat in our approach could be that a malicious user creates a widget, which contains code to attach a node to another widget, with the goal of bypassing our widget boundary identification algorithm. Since this requires a DOM change in the other widget, our DOM change violation detection algorithm should be able to recognize it. Another possible security issue could be that a widget subverts AJAX calls again, after our subversion code. An example of this is a subversion, in which the `requestForProxyId` of another element than the active one is added to the HTTP request. Another possibility is that a widget tries to steal and move the `requestForProxyId` attribute, with the goal of bypassing the origin widget detection algorithm. A possible solution could be to implement a verification mechanism, which generates a different (random) attribute name each time, and verifies that the node is correctly annotated before and after firing an event.

A malicious widget may attach an event handler to a widget VIC, which sends a request to a URL in the allowed URL list of VIC. This may lead to a vulnerability similar to cross-site request forgery [7]. This is a problem that should be handled by the widget developer rather than by our approach, as this is more of a server-side than a client-side issue. Widget developers should make sure that appropriate security measures are taken when a widget performs a critical action on the server, such as deleting an account, to make sure other widgets cannot easily counterfeit this action.

Scalability. We have shown our approach is capable of analyzing more than twenty widgets on the same page. In an industrial framework, many more widgets may require testing. Our approach requires the widgets to be analyzed to be on the same page. An expectation is that the maximum number of widgets on the page is limited by either the framework or browser memory. A solution to this problem may be the creation of subsets of widgets to test, for example by grouping them based on their functionality.

Another aspect of scalability is the number of testers which can use our approach at the same time. It is important to realize that each tester should use his own framework account, and place the widgets under test on his own personal test page. Because Atusa and our plugins require client-side installation only, our approach does not limit the number of concurrent testers. Therefore, this number is only limited by the number of allowed concurrent users in the framework, or by the web server.

Different Applications. Our approach requires no modification of web application code and, therefore, is very generic and works on any DOM-based web setting. The only framework-specific aspect of our implementation is currently the part that the Ajax calls have to be subverted as explained in Section 5. The applicability of our approach is not limited to Ajax-based widgets, and can be used to test the security of any web application in which inter-element interactions pose security threats. In fact, inter-element interactions between any type of web elements can be tested by changing the configuration settings and rules for widget boundary detection.

Our DOM change violation detection method has proved to be useful in other areas as well. One example is that many developers working with complex Ajax libraries such as jQuery, make mistakes, in which their actions may affect the whole page instead of one element or widget. Our plugin can help developers to detect such errors automatically.

For our HTTP request violation detection plugin we have added an HTTP proxy to Atusa. This proxy can be used in a variety of ways. Some examples are the detection of external JavaScript usage and links to external domains. Our approach, can also be used to explicitly deny access to certain domains or URLs, for example because they are known to contain phishing pages. A final application of the proxy in combination with Atusa, is the possibility of checking for dead links, or broken Ajax calls. Traditional dead links checkers cannot handle Ajax applications because they cannot crawl dynamic DOM-based content. Atusa can verify the validity of links and Ajax calls by analyzing the HTTP response code for each request on the proxy.

Threats to Validity. We have discussed some of the issues concerning the external validity of our empirical evaluation in the above discussion on completeness, security, and scalability. As far as the internal validity is concerned, we included a JUnit test suite for our plugins, to minimize coding errors. In addition, we have implemented a simple widget framework for our test suite, to validate the behavior and functionality of our plugins.

The requirements for widget pairs used in the cases were designed based upon known exploits in web application security research, for example by analyzing the effects of XSS or phishing attacks, e.g., many phishing attacks try to lure a user into sending the contents of a form to a URL specified by the malicious user, which corresponds with widget pair D1 in Table 1, 'Changing the action URL of a form'.

With respect to *reliability*, Atusa and the two plugins are open source and we intend to make the Pageflakes case available through our website, making the case fully reproducible. As far as the EWF case is concerned, because of the proprietary nature of the project, we are not allowed to make it publically available.

A threat to the validity of our results is the simplicity of the widget pairs implemented for the case studies. Although the pairs exhibit realistic behavior and represent real security vulnerabilities, the event sequences required to reach a violated state in our case studies are short. More case studies with published and more complex widgets are needed.

A final validity threat is that we have explicitly excluded some not-widget elements from the crawling process, such as menu items. Although such elements were not part of the widgets, excluding them may have led to considerably shorter crawling times.

9. RELATED WORK

Related work can be largely divided into two groups: prevention and detection approaches for web security analysis and testing.

Prevention Techniques. A common approach in current research is to place each widget in a sandbox to restrict inter-widget interaction and functionality. Examples of such approaches are SubSpace [6] and MashupOS [19], which introduce new HTML elements for giving a widget more freedom than an `IFrame`, and less than a `DIV`. Our approach places a virtual sandbox on the widget by disallowing interaction outside its boundary, without the requirement for changes to the HTML standard or browsers.

Using a proxy between the client and server is another common approach. Noxes [10] is a client-side firewall, which has access rules for browser connections, and for each page a list of allowed domains is defined. Because Ajax-based web widgets follow the single-page model, only one list can be used for each application in this approach. Our approach allows for the specification of such lists on widget level rather than application level, which makes our approach better applicable for Ajax-based applications, especially widget frameworks.

Scott *et al.* [16] propose a proxy-based approach to define security policies for validating or transforming HTTP request parameters. Their proxy embodies strong type checking or input validation. This approach differs from ours in that security policies must be defined manually for each parameter, while our approach does not require this, using a generic detection mechanism instead.

Halfond *et al.* [3] propose a method for dynamic prevention of SQL injection attacks by using positive tainting and instrumenting code to analyze a SQL query, right before it is sent to the database.

Detection Techniques. Finding security vulnerabilities through static analysis is mainly focused on detecting Cross-Site Scripting (XSS) and SQL injections [5, 20, 9].

Gatekeeper [11] is a static approach for verifying whether a widget follows a certain security policy. The authors have introduced a safely statically analyzable subset of JavaScript, in which the widget must operate.

Static analysis techniques, however, have serious limitations when applied to modern web applications, since the runtime behavior cannot be understood merely by analyzing the server-side code.

An important group of security tools are black-box security scanners that dynamically access the web application using public interfaces only. SecuBat [8] and WAVES [4] are tools that aim at detecting SQL injection and XSS by using a crawler. Most crawler-based tools are currently agnostic to the runtime DOM tree changes, and cannot handle the client side of the web spectrum. Our approach, which is based on dynamic analysis, has access to the DOM at all times during crawling and, hence, can analyze the runtime behavior of the application.

10. CONCLUDING REMARKS

This paper presents the first dynamic approach for automated security testing of Ajax web widgets. We have proposed a method for automatic detection of two types of inter-widget interaction violations. This paper makes the following contributions:

- The definition of two types of security violations in inter-widget interactions, namely DOM change and HTTP request violations;
- An algorithm for finding DOM change violations by connecting elements to DOM widget boundaries;
- An algorithm for finding HTTP request violations by coupling requests to the triggering DOM element;
- Implementation of these algorithms in two open source plugins (called Diva) for the Atusa crawling and testing infrastructure for Ajax applications.
- An empirical evaluation of the violation revealing capabilities, performance, and scalability of the approach, by means of two case studies.

Future work encompasses research on finding the best subsets of widgets for testing, as it is not always desirable nor possible to place all widgets on the same page, at the same time. In addition, conducting further case studies, and developing methods and plugins for spotting other security vulnerabilities, such as cross-site scripting and SQL injections, in Ajax applications, form part of our future research. Finally, more research should be done on the effect of a content proxy on our approach, and on ways to automatically analyze requests made through such a proxy.

11. REFERENCES

[1] A. Carzaniga, G. P. Picco, and G. Vigna. Designing distributed applications with mobile code paradigms. In *Proceedings of the 19th International Conference on Software Engineering (ICSE'97)*, pages 22–32. ACM Press, 1997.

[2] J. Garrett. Ajax: A new approach to web applications. Adaptive path, February 2005. `http://www.adaptivepath.com/publications/essays/archives/000385.php`.

[3] W. G. J. Halfond, A. Orso, and P. Manolios. Using positive tainting and syntax-aware evaluation to counter sql injection attacks. In *Proceedings of the 14th International Symposium on Foundations of software engineering (FSE'06)*, pages 175–185. ACM, 2006.

[4] Y.-W. Huang, C.-H. Tsai, T.-P. Lin, S.-K. Huang, D. T. Lee, and S.-Y. Kuo. A testing framework for web application security assessment. *J. of Computer Networks*, 48(5):739–761, 2005.

[5] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo. Securing web application code by static analysis and runtime protection. In *Proceedings of the 13th international conference on World Wide Web (WWW'04)*, pages 40–52, New York, NY, USA, 2004. ACM.

[6] C. Jackson and H. J. Wang. Subspace: secure cross-domain communication for web mashups. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 611–620, New York, NY, USA, 2007. ACM.

[7] N. Jovanovic, E. Kirda, and C. Kruegel. Preventing Cross Site Request Forgery Attacks. *Securecomm and Workshops, 2006*, pages 1–10, 28 2006-Sept. 1 2006.

[8] S. Kals, E. Kirda, C. Kruegel, and N. Jovanovic. Secubat: a web vulnerability scanner. In *Proc. 15th int. conf. on World Wide Web (WWW'06)*, pages 247–256. ACM, 2006.

[9] A. Kiezun, P. J. Guo, K. Jayaraman, and M. D. Ernst. Automatic creation of SQL injection and cross-site scripting attacks. In *Proceedings of the 31st International Conference on Software Engineering (ICSE'09)*. IEEE Computer Society, 2009.

[10] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic. Noxes: a client-side solution for mitigating cross-site scripting attacks. In *Proceedings of the 2006 ACM symposium on Applied computing (SAC'06)*, pages 330–337. ACM, 2006.

[11] B. Livshits and S. Guarnieri. Gatekeeper: Mostly static enforcement of security and reliability policies for JavaScript code. Technical Report MSR-TR-2009-43, Microsoft Research, 2009.

[12] A. Mesbah, E. Bozdag, and A. van Deursen. Crawling Ajax by inferring user interface state changes. In *Proc. 8th Int. Conference on Web Engineering (ICWE'08)*, pages 122–134. IEEE Computer Society, 2008.

[13] A. Mesbah and A. van Deursen. A component- and push-based architectural style for Ajax applications. *Journal of Systems and Software*, 81(12):2194–2209, 2008.

[14] A. Mesbah and A. van Deursen. Invariant-based automatic testing of Ajax user interfaces. In *Proceedings of the 31st International Conference on Software Engineering (ICSE'09), Research Papers*, pages 210–220. IEEE Computer Society, 2009.

[15] J. Ruderman. The Same Origin Policy. `http://www.mozilla.org/projects/security/components/same-origin.html`, 2001.

[16] D. Scott and R. Sharp. Abstracting application-level web security. In *Proceedings of the 11th international conference on World Wide Web (WWW'02)*, pages 396–407, New York, NY, USA, 2002. ACM.

[17] G. F. Stefano Di Paola. Subverting Ajax. In *23rd Chaos Communication Congress*, 2006.

[18] W3C. The global structure of an html document. `http://www.w3.org/TR/REC-html40/struct/global.html`.

[19] H. J. Wang, X. Fan, J. Howell, and C. Jackson. Protection and communication abstractions for web browsers in MashupOS. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles (SOSP'07)*, pages 1–16. ACM, 2007.

[20] G. Wassermann and Z. Su. Static detection of cross-site scripting vulnerabilities. In *Proceedings of the 30th international conference on Software engineering (ICSE'08)*, pages 171–180. ACM, 2008.

[21] R. K. Yin. *Case Study Research: Design and Methods.* SAGE Publications Inc, 3d edition, 2003.