Automated Security Testing of AJAX Web Widget Interactions

Master's Thesis

Cor-Paul Bezemer

Automated Security Testing of AJAX Web Widget Interactions

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Cor-Paul Bezemer
born in Den Haag, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Exact International Development B.V.
Poortweg 6
Delft, the Netherlands
www.exactsoftware.com

Automated Security Testing of AJAX Web Widget Interactions

Author: Cor-Paul Bezemer
Student id: 1149601
Email: c.bezemer@student.tudelft.nl

Abstract

Over the years AJAX, a technique for improving the responsiveness of web applications, has become increasingly popular. One of the results of AJAX is the development of a new type of web application component called web widget. Widgets are mini-applications which are placed next to each other on a web page. This has consequences for their security. In this report two security threats are explained. The first threat discussed is the case in which a widget changes the DOM of another widget. The second threat discussed is the case in which a widget steals data from another widget. We propose a dynamic approach for automatically detecting these issues. Our approach uses ATUSA, a testing framework capable of crawling AJAX applications, for which we have developed two security testing plugins. In this report we also evaluate our approach using three case studies. The first case study is conducted on test widgets, which we created for a simplified widget framework. The second case study is conducted on the Exact Widget Framework, a widget framework which is being prototyped by the Research and Innovation team of Exact Software. The final case study is performed on Pageflakes, an industrial, widely used widget framework. The results of these case studies show that our approach has high violation-detection capabilities with a low false positive detection rate.

Thesis Committee:

Preface

I started my graduation project in February 2008 at Exact Software. After a good year of hard work and what turned out to be a very interesting journey, I proudly present to you my Master's thesis.

First of all, I would like to thank all the people at Exact for offering a warm and inspiring environment for doing my research. From day one, all the Research & Innovation team members have made me feel more than welcome. I would especially like to thank Bart Platzbeecker, for sharing his knowledge and experience in the field of security, and for his input in our brainstorm sessions. I would also like to thank Toine Hurkmans for the weekly progress meetings with Mark and Bart, his feedback and supervision during my project.

Furthermore, I would like to thank Ali Mesbah for his supervision and many feedback and brainstorm sessions at the TU Delft. His input has always been inspiring and motivating, which has helped to take my research to a higher level. Finally, I would like to thank Arie van Deursen, for his feedback, ideas and support throughout my project.

Cor-Paul Bezemer
Delft, the Netherlands
March 9, 2009

Contents

List of Figures

Chapter 1

Introduction

Over the last couple of years there has been a shift in web application development. Web applications have changed from being static to being highly dynamic, using new Web 2.0 techniques. One of these techniques is Asynchronous JavaScript and XML (AJAX) [6], with which web applications can be made more interactive and user friendly. While in traditional web applications a complete page needs to be refreshed, AJAX makes it possible to refresh parts of the page. This contributes to the user experience and gives the user the feeling similar to a desktop application instead of a web application.

The advantages of AJAX have led to the development of web widgets. Widgets are mini-applications that run indepently or next to each other on a web page inside a browser. They can be used for a wide variety of tasks such as showing the latest news headlines or the weather. Widgets have gained much popularity over the last years because of web sites like iGoogle[1] and social network sites like Facebook[2]. These web sites allow users to select widgets from a catalog, configure them and host them on a personalized starting page. Such a starting page may contain multiple widgets, which makes it a powerful and convenient way of offering widgets to a user.

The introduction of AJAX and web widgets has had consequences for different areas of web application development. One of these areas is security. Security in software is concerned with maintaining the absence of unauthorized access while preserving normal behaviour for authorized requests [3, 23]. Another point security is concerned with is data protection. It is important that data which should be secure, such as private data, cannot be retrieved from the application by another user than the data is intended for. Web widgets are web applications and therefore suffer at least from the same security threats [10]. Unfortunately the nature of web widgets has introduced some new problems. Web widget frameworks place multiple widgets in the same environment on a web page. This may have serious consequences for the security and privacy of users of the framework.

In this thesis the security and privacy threats of running multiple web widgets in the same environment will be discussed. Also a method will be proposed for testing web widgets for these threats. The method uses ATUSA [29], which is based on CRAWLJAX [25],

[1]iGoogle: `http://www.igoogle.com`

[2]Facebook: `http://www.facebook.com`

a tool for automated navigation of AJAX applications. In order to explain the proposed method this report has the following structure. First background information about AJAX, web widgets and widget frameworks will be given. In Chapter 3 the problem dealt with in this report is explained in detail. In this chapter a motivation for the selected approach is given as well. In Chapters 5 and 6 an approach for detecting security threats of running widgets in the same environment is proposed. This approach is evaluated by three case studies in Chapter 7. In Chapter 8 a discussion is given of a number of aspects regarding our approach. After this we give an overview of related work in this research area. Finally we will draw a conclusion about our research.

Chapter 2

Background

2.1 AJAX

Traditional web applications are based on a multi page interface model in which all interactions have to go through the server by means of synchronous requests [26]. The result of this is that after every interaction the user has to wait for the new web page to be loaded (see Figure 2.1). This does not give the user the feeling of working with a responsive application because it lacks the responsiveness of a desktop application [6]. It also wastes bandwidth because the complete page is sent by the web server, although usually only a small part of the page changes [38]. To overcome this problem, a new technique called AJAX has been introduced.

Asynchronous JavaScript and XML (AJAX) is a technique which allows to make asynchronous requests in a web application instead of the synchronous requests used in traditional web applications [6]. A consequence of this is that the user can continue viewing the page after a request has been made. The response is sent by the server as XML (or another format like JSON) and incorporated on the page by the client using JavaScript and the Document Object Model (DOM). The advantage of this is that parts of the page can be reloaded rather than only the complete page (see Figure 2.1). This is a serious improvement of the user experience since the responsiveness can increase compared to traditional web applications.

2.1.1 Consequences of AJAX for Web Application Development

Because AJAX allows an increase in user experience with minimal effort, many companies are migrating their pages to AJAX powered pages. This process, sometimes called ajaxification [27], requires some consideration because the asynchronous nature of AJAX has some consequences for the development of web applications. Interactions in traditional web applications require the page to be reloaded completely but with AJAX this is no longer necessary. Therefore, there is no need to develop complete pages but UI components can be developed instead [26]. This leads to the component-based structure of AJAX and the development of web widgets (see Chapter 2.2) rather than web pages.

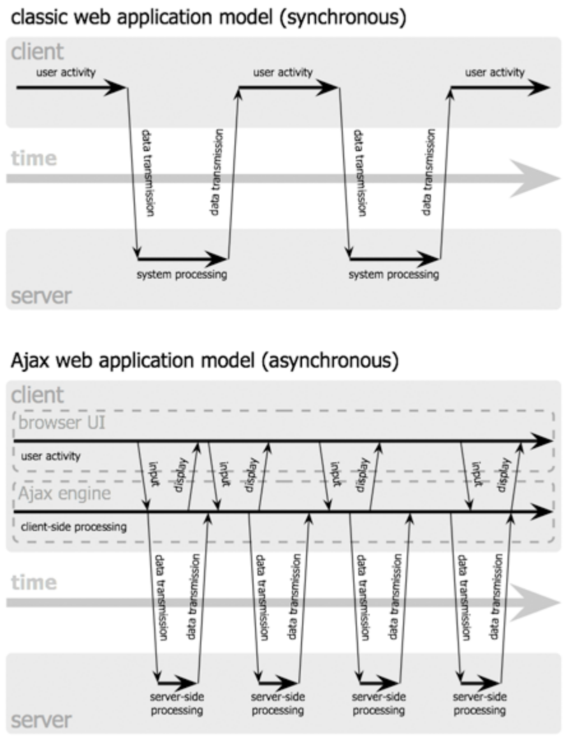


Figure 2.1: Interaction in a traditional and an ajaxified web application, originally taken from [6].

Another consequence is the difference in site navigation. In traditional web applications navigation is done through hyperlinks and the submission of forms. With AJAX, the set of clickable elements on a site is not limited to hyperlinks and may change with every click [28]. This is because the response of the AJAX call may add new content, containing clickable elements or AJAX calls, to the DOM. This 'hidden' content is not reachable through hyperlinks, but only through the DOM tree or instance of the client that made the AJAX call. Because not all content of an AJAX page is reachable through a URL, AJAX is based on a single page model. Software like web crawlers and test frameworks are usually not prepared for the single page model and do not keep track of the DOM to crawl web pages [39]. Therefore they need to be adjusted in order to reach the hidden content.

Finally, because AJAX relies heavily on JavaScript the web application must be tested on a variety of browsers and operating systems. This is because most browsers are known to have JavaScript compatibility issues as they do not all implement the same subset of JavaScript features [21, 22]. Various frameworks like the Google Web Toolkit[1] and Echo2[2] exist to help the web application programmer overcome these compatibility issues. These frameworks allow the developer to create AJAX applications in Java, which more develop-

[1]Google Web Toolkit: `http://code.google.com/webtoolkit/`

[2]Echo Web Framework: `http://echo.nextapp.com/site/echo2/`

ers are familiar with and does not have browser compatibility issues as it is a server-side language. When such an application is deployed, the framework converts the Java code to multi-browser JavaScript.

AJAX also introduces changes in the software architecture of an application. An architectural style for AJAX applications is discussed in the next section.

2.1.2 An Architectural Style for AJAX Applications

Mesbah and van Deursen [26] consider AJAX applications to be hybrids of web and desktop applications and believe that none of the existing architectural styles are suitable for representing them. A typical AJAX application exists of a client-side part, usually written in JavaScript and a server-side part, for example a Web service. They propose SPIAR [26], an architectural style for AJAX applications. In SPIAR three types of elements can be identified:

- **Processing elements** - The components that perform transformations on the data elements,
  - *Client browser* - The client browser processes the representational model of the web page and creates the user interface from it;
  - AJAX *engine* - The engine that handles all AJAX requests;
  - *Server application* - The application on the server that handles and responds to the HTTP requests made by the client browser;
  - *Service provider* - The logic engine of the server application;
  - *Delta encoder/decoder* - The component that encodes/decodes the delta-communication messages;
  - *UI components* - The set of server components that render the UI on the client;
- **Data elements** - The data that is used and transformed by the processing elements,
  - *Representation* - Any type of media;
  - *Representational model* - A runtime abstraction of the user interface, for example the DOM;
  - *Delta-communication messages* - Messages to exchange state changes between the server and client;
- **Connecting elements** - The elements that connect the components and enables them to communicate,
  - *Events* - An event is caused by a user action. It is propagated to the engine and the result on the server is the invocation of a service. Events form the basis of the action model in SPIAR;
  - *Delta connector* - Communication media connecting the server and client;

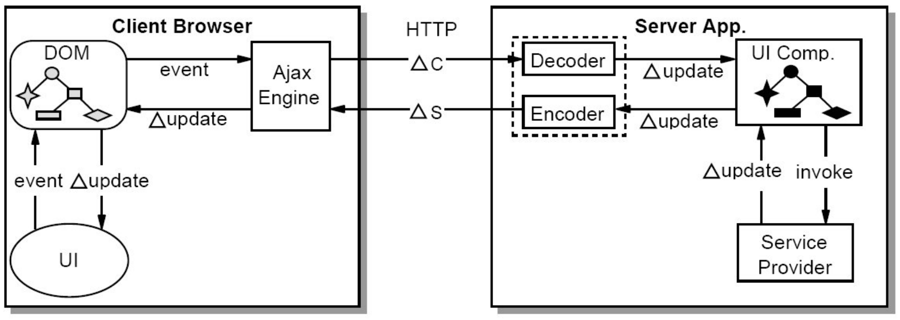


Figure 2.2: View of a SPIAR based architecture, originally taken from [26].

– *Delta update* - Updates the representation model on the client and the component model on the server.

SPIAR uses delta-communication to model the notifications of the changes in the client browser and server application. In this type of communication only the state changes are exchanged instead of the full page. As explained in this chapter, this is typical for communication in AJAX applications.

2.2 Web Widgets

The asynchronous nature of AJAX, and dynamic DOM of AJAX applications allow for new possibilities in web applications. It allows developers to update individual user interface elements rather than allowing full page updates only as in traditional web applications. A consequence of this is the shift from an application-based approach to a component-based approach. In this approach components, called web widgets or gadgets, are created instead of complete applications. Web widgets are AJAX-based components which run independently from each other inside any HTML-driven page. They are usually placed on a web page using one of the many available web widget frameworks. A selection of these frameworks is discussed in the next sections. In addition to placement on a web page, web widgets can be used on desktops using dashboard software such as the Microsoft Vista Sidebar or Apple's Dashboard. Finally, widgets are often convenient applications which require a small amount of space, both in memory usage and on the screen, which makes them suitable for usage on a mobile phone. In the remainder of this thesis we will focus on browser-based web widgets.

A disadvantage of web widgets is that they usually work on the platform they were developed for only. Web widgets of different platforms are usually not interchangeable. An exception to this are web widgets that are developed using the Universal Widget API[3] (UWA). The UWA was developed by NetVibes[4] and offers an interface for creating generic

[3]NetVibes Universal Widget API: `http://dev.netvibes.com`

[4]NetVibes: `http://www.netvibes.com`

web widgets that can run on different platforms like NetVibes, iGoogle and the iPhone[5].

In most cases web widgets can be developed by users of a framework. Since the introduction of web widgets, thousands of them have been created for a wide variety of tasks. Some examples are used to display the weather, to show news headlines and to play games. Often frameworks offer users a catalog where they can download predefined web widgets.

2.2.1 Web Widget Frameworks

Many vendors have been working on web widget frameworks during the last months. Most of them provide the same functionality and selecting one is a matter of taste. Some of the more advanced web widget frameworks are iGoogle, NetVibes, Pageflakes[6] and MyYahoo[7]. Social network sites like Hyves[8] and Facebook have also started to allow their users to create web widgets that can be used on their profile. One similarity these frameworks share is that the user is allowed to create client-side widgets only. No code written by a user will be executed on the server. This rule is a limitation for the developer but is established to improve the level of security of the web widgets that are created.

2.2.2 DIVs versus IFrames

As explained, widgets are placed on a web page next to each other by a widget framework. This can be done in two ways: using `DIV` containers and using `IFrames`. 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 (see Section 3.1) limits them or the framework from accessing each other's properties. The main problem with using `IFrames` is that they do not provide real integration on the page. `IFrames` impose limitations on the page layout and their contents do not use the CSS style sheet of the framework. They also do not allow a user to resize the widget [41].

A solution for these limitations is to place each widget inside a `DIV` container. `DIV` containers allow the user to resize widgets and they do not impose restrictions on the widget's layout and style. Unfortunately they do not provide an isolated environment for their execution. Widgets which are placed next to each other in `DIV` containers run in the same execution environment and are therefore allowed to manipulate each other's properties.

2.2.3 Mashups

A type of web application closely connected to widgets and widget frameworks is the mashup. The term mashup originates from the music industry, in which a mashup is a track that is created by combining two existing tracks [24]. In software technology a mashup is a web application in which multiple data sources and/or services are combined [5, 24, 42].

[5]Apple iPhone: `http://www.apple.com/iphone/`

[6]Pageflakes: `http://www.pageflakes.com`

[7]MyYahoo: `http://www.myyahoo.com`

[8]Hyves: `http://www.hyves.nl`

These data sources and services can be accessed through an API. ProgrammableWeb[9], one of the largest online directories for APIs and mashups, lists over 1150 APIs and over 3700 mashups at the time of writing.

One of the first and most famous mashups is HousingMaps[10]. HousingMaps combines data from classifieds site CraigsList[11] with the Google Maps[12] service. Housing offers are displayed on the map so users can see which offers are available in a certain area. This was an early example of a mashup and it became famous because of its effectiveness and simplicity.

Many large software companies are offering a (beta) framework for creating mashups. Yahoo! Pipes[13] is a tool to aggregate and manipulate data sources from the web [44]. A user can create a pipeline by connecting modules which can perform operations on data. When a data source is fed to this pipeline, all modules in it perform their operations on the data. The output contains the manipulated data, which can be used as a data source for other web applications [11].

While Yahoo! focuses on data manipulation, other companies are offering a more extensive solution for creating mashups. Microsoft and Google offer an online editor for creating widgets, mashups and web sites with their Popfly[14] and Google Mashup Editor (GME)[15]. After creating a widget, mashup or web site the owner may publish it so that other users of the framework can use it [8, 31]. After publishing the element the source code is always visible to the community. The goal of this is for the community to support itself by creating a large database of live examples.

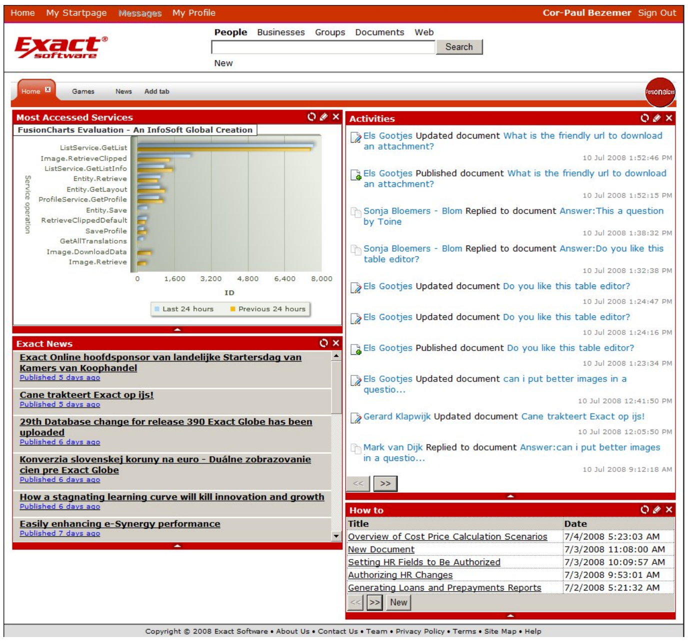
Intel takes a different direction with Mash Maker[16], a browser extension that can be used for creating mashups while browsing web sites. Mash Maker is a browser plugin which notifies users when a mashup for the site they are browsing is found in the database. Users can then choose to use one of the existing mashups or create their own.

Mashups are closely connected to widgets, because widgets which are placed on a personal homepage form a mashup. Therefore many of the problems and solutions which apply to mashups, also apply to widgets and vice versa.

2.3 Exact Widget Framework

The research done for this project was part of a research project at Exact Software[17]. Exact Software is a Delft-based company which was founded in 1984 and specializes in business applications. Exact has over 2500 employees in 40 different countries and their software is used in 125 countries.

[9]ProgrammableWeb: `http://www.programmableweb.com`
[10]HousingMaps: `http://www.housingmaps.com`
[11]CraigsList: `http://www.craigslist.com`
[12]Google Maps: `http://maps.google.com`
[13]Yahoo! Pipes: `http://pipes.yahoo.com`
[14]Microsoft Popfly: `http://www.popfly.com`
[15]Google Mashup Editor: `http://www.googlemashups.com`
[16]Intel Mash Maker: `http://mashmaker.intel.com/`
[17]Exact Software: `http://www.exactsoftware.com/`



Exact News

Exact Online hoofdsponsor van landelijke Startersdag van Kamers van Koophandel
Published 5 days ago

Cane trakteert Exact op ijs!
Published 5 days ago

29th Database change for release 390 Exact Globe has been uploaded
Published 6 days ago

Konverzia slovenskej koruny na euro - Duálne zobrazovanie cien pre Exact Globe
Published 6 days ago

How a stagnating learning curve will kill innovation and growth
Published 6 days ago

Easily enhancing e-Synergy performance
Published 7 days ago

Activities

Els Gootjes Updated document What is the friendly url to download an attachment?
10 Jul 2008 1:52:46 PM

Els Gootjes Published document What is the friendly url to download an attachment?
10 Jul 2008 1:52:15 PM

Sonja Bloemers - Blom Replied to document Answer:This a question by Toine
10 Jul 2008 1:38:32 PM

Sonja Bloemers - Blom Replied to document Answer:Do you like this table editor?
10 Jul 2008 1:32:38 PM

Els Gootjes Updated document Do you like this table editor?
10 Jul 2008 1:24:47 PM

Els Gootjes Updated document Do you like this table editor?
10 Jul 2008 1:24:16 PM

Els Gootjes Published document Do you like this table editor?
10 Jul 2008 1:23:34 PM

Els Gootjes Updated document can i put better images in a questio...
10 Jul 2008 12:41:50 PM

Gerard Klapwijk Updated document Cane trakteert Exact op ijs!
10 Jul 2008 12:05:50 PM

Mark van Dijk Replied to document Answer:can i put better images in a questio...
10 Jul 2008 9:12:18 AM

<< >>

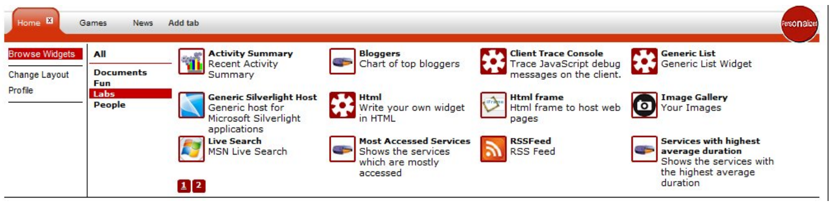
How to

Title	Date
Overview of Cost Price Calculation Scenarios	7/4/2008 5:23:03 AM
New Document	7/3/2008 11:08:00 AM
Setting HR Fields to Be Authorized	7/3/2008 10:09:57 AM
Authorizing HR Changes	7/3/2008 9:53:01 AM
Generating Loans and Prepayments Reports	7/2/2008 5:21:32 AM

<< >> New

Copyright © 2008 Exact Software • About Us • Contact Us • Team • Privacy Policy • Terms • Site Map • Help

Figure 2.3: Exact Widget Framework concept personalized start page MyStartpage.



Activity Summary Recent Activity Summary

Bloggers Chart of top bloggers

Client Trace Console Trace JavaScript debug messages on the client.

Generic List Generic List Widget

Generic Silverlight Host Generic host for Microsoft Silverlight applications

Html Write your own widget in HTML

Html frame Html frame to host web pages

Image Gallery Your Images

Live Search MSN Live Search

Most Accessed Services Shows the services which are mostly accessed

RSSFeed RSS Feed

Services with highest average duration Shows the services with the highest average duration

Figure 2.4: Exact Widget Framework concept widget catalog.

Exact is currently researching the possibilities of building software solutions using widgets. The goal of this is to allow to configure an application and the widgets it uses, with metadata. Because no programming is necessary, the configuration can be done by an expert without the help of a developer. As a proof of concept, a prototype community solution is being developed on a prototype widget framework, in which the community is allowed to read and publish content. Because the framework does not have an official name yet, it is referred to as the Exact Widget Framework (EWF) throughout this document. The prototype framework offers an environment for running and creating web widgets aimed towards enterprise situations. One of the key components of the prototype framework is be the personalized start page. Each user will have access to a personalized start page, 'MyStartpage', on which he can place widgets from a catalog using a drag and drop mechanism. A concept MyStartpage page on which widgets are placed is shown in Figure 2.3. On this page multiple tabs can be created, which help the user to place more web widgets on his page in an efficient way. The prototype framework also provides the possibility to create hosting pages, which are pages with a predefined collection of widgets. These hosting pages may, for example act as the page an anonymous user sees. They may also be placed on one of the tabs, allowing the user to download hosting pages with, for example only news or games widgets. A special feature of the prototype framework is the possibility to allow interaction between web widgets. This feature will allow a web widget to change or control the content of another web widget. An example is a news headline widget which loads the selected news item in a document editor widget while remaining active.

A concept widget catalog which was available at the time of writing is shown in Figure 2.4. The goal is to allow engineers to create and publish their own web widgets. At the time of writing it is unknown whether an API will be available for generating widgets, however the goal of Exact is to give developers as much freedom as possible, allowing them to create the client-side and server-side code for a web widget.

In the EWF widgets are placed inside `DIV` containers. As explained in Section 2.2.2, this means that the widgets do not execute in an isolated environment. In the next chapter the risks of this approach are explained.

Chapter 3

Problem Definition

Widget frameworks can place their widgets in `IFrames` or `DIV` containers. When widgets are being placed inside `IFrames`, their behaviour is limited by a browser policy, the Same Origin Policy [35]. For widgets in `IFrames`, the main limitation caused by this policy is that they are not allowed to access each other's properties. In the EWF, widgets are placed in `DIV` containers rather than `IFrames`, which has consequences for their security and privacy. When widgets are being placed inside `DIV` containers, the Same Origin Policy no longer applies and they are allowed to access and change other widgets' properties.

In this chapter first the Same Origin Policy is explained. After this, scenarios are given of security issues which rise when widgets are being placed in `DIV` containers. This chapter concludes with the definition of the problem researched in this thesis, and a brief description of the approach taken to solve this problem.

3.1 Same Origin Policy

In 1996 a browser design principle was introduced in Netscape Navigator 2.0 [35]. This principle, the Same Origin Policy (SOP), states that two documents with different origins should not be allowed to view or change each other's properties. As defined by Mozilla [35]:

> *"The same origin policy prevents a document or script loaded from one origin from getting or setting properties of a document from another origin."*

All of the currently available browsers implement this policy in one form or another. Table 3.1 gives an overview of outcomes for origin comparisons.

3.1.1 SOP Constraints

In this section scenarios of dangerous situations which would be possible without the SOP are described. In the scenarios page A is from a different origin than page B.

Scenario 3.1.1: Browser window manipulation

1. Page A opens a browser window with page B as content.

Table 3.1: Comparing origins with http://store.company.com/dir/page.html, originally taken from [35].

URL	**Outcome**	**Reason**
http://store.company.com/dir2/other.html	Success	
http://store.company.com/dir/inner/another.html	Success	
https://store.company.com/secure.html	Failure	Different protocol
http://store.company.com:81/dir/etc.html	Failure	Different port
http://news.company.com/dir/other.html	Failure	Different host

2. Page A changes the appearance of page B.
3. Page B looks different to the user than intended to by the owner.

Scenario 3.1.2: Frame content manipulation

1. Page A contains a frame with page B as content.
2. Page A changes the appearance of page B.
3. Page B looks different to the user than intended to by the owner.

Scenario 3.1.3: Cookie manipulation

1. Page A sets a cookie with sensitive data.
2. Page B opens the cookie from page A and changes the sensitive data.

Scenario 3.1.4: Data stealing

1. Page A contains a frame with page B as content.
2. Page A reads the data of page B.

3.1.2 JavaScript and the SOP

JavaScript is used on many web pages nowadays. The introduction of AJAX, which heavily relies on JavaScript as described in Section 2.1, has contributed greatly to this increase. JavaScript can be used to change the Document Object Model (DOM) of a page. The DOM is a model of all the elements, including their attributes, of a web page. By changing the DOM, JavaScript may for example manipulate styling properties or the action URL of a form. An example of this is displayed in Figure 3.1, in which the background color of an element is changed using JavaScript.

JavaScript has the ability to access more than just styling properties. Other properties available to JavaScript are the cookies, location and domain properties of the page. The

```
<html>
<head><title>Website</title></head>
<body><div id="div01"></div></body>
<script type="text/javascript">
document.getElementById('div01').style.backgroundColor = "green";
</script>
</html>
```

Figure 3.1: Changing the background color of an element.

way these properties may be changed or accessed is also limited by the SOP. JavaScript may not access a cookie from `google.com` when it is running from a page in the domain `tudelft.nl`. The domain JavaScript is running in is described by the document.domain property. JavaScript is allowed to change this property in a very limited way. While it is allowed to set document.domain to `google.com` from `mail.google.com`, it is not allowed to set the domain to `tudelft.nl` because of the SOP.

As described in the previous section, JavaScript may not manipulate the DOM of another site. However, it is possible to include external JavaScript files on a page. By doing this, the included JavaScript becomes part of the same execution environment and is therefore no longer restricted by the SOP when trying to access the DOM. The danger of this is that the developer should fully trust the owner of the external JavaScript file. If the owner decides to change it, the safety of the including page may be compromised.

3.2 Widget SOP Violation Scenarios

Although it is important to protect and respect the origin of a document, the SOP has its limitations. The most important limitation is the fact that two documents have either no or full access to each other's properties. This leaves developers with the difficult choice between security and functionality. In Chapter 9 a number of methods for controlling JavaScript DOM access are briefly introduced, but these all have their own problems.

Widgets demonstrate a special case of this limitation of the SOP. Their functionality requires them to run in the same environment, but their security requires them to run in different execution environments. This problem has traditionally been addressed in widget frameworks by running each (untrusted) widget in an `IFrame`. Another approach is warning the user that an untrusted widget will be placed on the page and that this may compromise his security. These approaches are both not desirable because they may limit functionality or compromise security. Therefore widgets in the EWF are placed inside `DIV` containers. In this section a number of realistic scenarios is given of dangerous situations which are possible when widgets are being placed inside `DIV` containers and no longer protected by the SOP. In Section 3.2.2 a generalization of these scenarios is made.

3.2.1 Scenarios of Possible Violations by Widgets

In this section scenarios of possible violations by widgets of the principle implemented by the SOP are presented. In all these scenarios the assumption exists that a user may publish his own widgets to the widget framework. Another assumption which is made is that data stealing is a violation only if the data is communicated to the server. Although data stealing may happen on the client only, we do not consider this to be a security issue because the stolen data can never reach the malicious user unless it is sent to the server. Two main actors play a role: the malicious user who publishes the malicious widget, and the victim user who downloads and uses the widget. The malicious widget is called MAL and the widget that is attacked by MAL is called VIC in the scenarios.

Scenario 3.2.1 : Stealing a password using a key logger

1. The malicious user creates MAL, which contains a key logger.
2. The malicious user publishes MAL.
3. The victim user places MAL and VIC on his page.
4. The victim user logs into Gmail using VIC.
5. The key logger in MAL logs the entered login information, saves it in a variable and sends this information to the malicious user using HTTP requests.

Scenario 3.2.2: Scraping private data without event

1. The malicious user creates MAL, which contains a screen scraper.
2. The malicious user publishes MAL.
3. The victim user puts MAL on his page.
4. MAL immediately scrapes private data from all widgets on the page, saves it in a variable and sends this data to the malicious user using HTTP requests.

Scenario 3.2.3 Changing the action URL of a form

1. The malicious user creates MAL, which contains code to change the action URL of a form in VIC.
2. The malicious user publishes MAL.
3. The victim user puts MAL and VIC on his page.
4. MAL changes the action URL of a form in VIC on the page.
5. The victim user enters private data in the form and submits it.
6. The form contents are posted to the URL entered by the malicious user.

Scenario 3.2.4 Scraping content after initiating event on VIC

1. The malicious user creates MAL, which contains code to scrape the content of VIC after a click event in VIC and sends it to the malicious user.
2. The malicious user publishes MAL.
3. The victim user puts MAL and VIC on his page.
4. The victim user performs a click event in VIC.
5. The content of VIC is sent to the malicious user using HTTP requests.

Scenario 3.2.5 Changing the description text of a link

1. The malicious user creates MAL, which contains code to change the description text of a link in VIC from 'Delete account' to 'Upgrade account for free'.
2. The malicious user publishes MAL.
3. The victim user puts the MAL and VIC on his page.
4. MAL changes the description text of a link in VIC.
5. The victim user clicks the link and executes another action than required.
6. The account is deleted by the victim user.

3.2.2 Generalization of the Scenarios

The scenarios described in the previous section can be generalized. These generalizations will be used throughout the remainder of this thesis to describe sets of violations. The generalization can be made as follows:

- **G1**: MAL changes the DOM of VIC (Scenarios 3.2.3, 3.2.5),
- **G2**: MAL steals data from VIC and sends it to the server using HTTP requests (Scenarios 3.2.1, 3.2.2, 3.2.4).

3.3 Problem Definition

In the future it will be possible for developers to build and publish their own web widgets, as is the case for many of the currently existing widget frameworks. These widgets may then be used by other users of the framework. Because the widgets eventually will be released to the public, it is important that they do not harm the server nor client they are running on in any way. The scenarios given in the previous section describe realistic cases of security issues in widgets which are published by malicious developers. Since users will lose confidence in a framework which does not guarantee their privacy and security, it is important to prevent the publishing of insecure widgets.

To guarantee the security of the widgets in the catalog, all widgets must be verified by Exact before they are being published. This verification process is time-consuming and therefore it should preferably be automated. Another good reason to automate this process is to be able to use regression testing after widgets or their security requirements have changed. The idealized process for verification is demonstrated by the following steps:

1. A developer uploads a widget to a test server.
2. The test server automatically decides if the widget conforms to a set of security requirements. In the rest of this report a widget will be referred to as 'accepted' if it does conform to this set of requirements and as 'rejected' if it does not.
3. If the widget is accepted, it is sent to the deployment server and the developer receives an ACCEPTED message. On the deployment server the widget is placed in the widget catalog which makes it available for other users of the framework. If the widget is rejected, an error report is returned to the developer together with a REJECTED message.
4. The widget is removed from the test server.

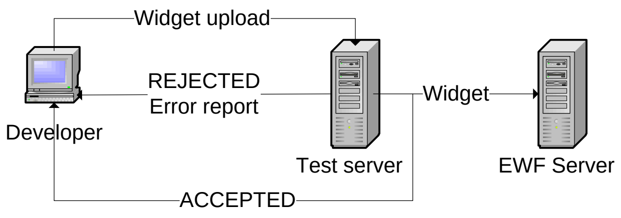


Figure 3.2: Preferred situation for publishing a widget.

Unfortunately current automated testing solutions are not yet mature enough to replace manual testing solutions. Therefore it is important to realize that an automated testing approach will be used to support the manual testing process. This thesis describes the process and result of making a first step towards such an automated security testing framework for supporting the security tester.

The goal of this thesis is to research approaches for detecting the violations described in the previous chapter. In order to come to such an approach, the following research questions will be addressed:

RQ1: What are the problems and difficulties of detecting these violations?

RQ2: Can we propose an automated approach for detecting these violations?

The goal of this thesis is to propose a method for detecting the violations defined by the generalized scenarios in Section 3.2.2, while taking the difficulties encountered during the research for RQ1 into account.

3.4 Motivation

Because web applications are becoming more popular, more security breaches are found on a regular basis. This is, for example demonstrated by the number of bugs posted to the Bugtraq[1] mailinglist. Detecting and preventing these security breaches is a difficult and time-consuming process. Automated approaches are being researched but they are far from replacing manual approaches. Therefore every attempt at improving automated security solutions can be valuable. This is especially the case for AJAX applications. Because AJAX is a new technique, relatively few research has been done which focuses on AJAX security. This research project could deliver one of the first automated security assessment approaches geared towards AJAX applications.

The focus of this project is on the widget interaction, which is a topic which has not had a large amount of attention in recent research yet. All existing widget frameworks appear to be avoiding the problem by completely disallowing inter-widget interaction rather than disallowing only certain behaviour for published widgets. This research could provide new insights from a different point of view on the subject.

For Exact the benefits are obvious as the goal of this project is to deliver a prototype of an automated testing framework, which can be used to assess the security of widgets before adding them to the widget catalog of the framework they are researching. The testing framework should be easily extendible so that testing modules for other software quality attributes will be easy to implement. Furthermore offering only secure widgets in their catalog can help improve the stability of the widget framework as it is more difficult to break functionality when widgets are secure. A final improvement is that an automated test framework is a scalable option: there is no significant number of employees needed to test the widgets, no matter how many there are being uploaded. Although manual testing will still be necessary, an automated test framework will help the tester to do the testing in an easier and faster way.

3.5 Selected Approach for this Project

In order to select the best approach for detecting SOP violations by widgets, a few important things must be considered. The first is the fact that widgets may heavily rely on the use of AJAX. As a consequence any techniques used must support AJAX and because AJAX relies heavily on JavaScript, any techniques used must support JavaScript as well. Unfortunately, because of the way JavaScript works, static analysis (see Chapter 9) is very difficult to achieve. JavaScript is a prototype-based language [33]. A prototype based language is an object-oriented based language without classes. Instead of class instances objects are clones of existing objects. These objects, called prototypes, may be changed during runtime, for example by adding or reimplementing methods. Although this offers a great deal of flexibility to the developer it makes placing constraints for static analysis very difficult and therefore we have decided to avoid static analysis of JavaScript as much as possible. A final

[1]Bugtraq mailinglist: `http://www.securityfocus.com/archive/1`

consideration which must be made is that the requirements of the tool state that it should be automated rather than manual (see Section 3.3).

After taking these considerations into account we have chosen the direction of our approach to be automated and dynamic (see Chapter 9). In order to use an automated and dynamic approach it is important to have access to a tool, which is capable of testing AJAX applications. ATUSA, discussed in Chapter 4, is a tool for automated testing of AJAX applications. ATUSA can be easily extended by creating plugins which makes it an excellent candidate. The creation of security-related plugins can also serve as a case study for the ATUSA plugin structure and as a proof of concept.

Chapter 4

ATUSA

Search engines like Google and Yahoo cover the part of the web that is called the 'publicly indexable part', which means the part that is reachable by following hyperlinks [28]. AJAX pages often contain content that is not reachable through hyperlinks but through other clickable elements like `DIV` containers and `SPAN` elements. Elements may also become clickable at runtime, for example by attaching an `onclick` handler. Because search engines do not index content reachable through these elements it is the responsibility of the programmer to make the content visible in another way. In addition to search engine visibility, the hidden content problem exists in automated testing solutions. AJAX pages can be made more accessible for search engines and testing solutions in two ways [28]:

- **Graceful degradation** - Design the web site as desired and add options, like <noscript> tags, for user-agents that do not support all the options,
- **Progressive enhancement** - Build the web site as simple as possible and enhance for users that support the enhanced functionality.

These approaches limit the use of AJAX and preferably a new solution should be found. Mesbah et al. propose the tool ATUSA [29], designed for automatically testing AJAX applications. ATUSA uses a crawler, CRAWLJAX [25, 28], to reveal hidden content of AJAX web sites. CRAWLJAX builds a state-flow graph[1] to model the user interface state changes. This state-flow graph is created incrementally, taking the initial state as root. From this root the set of candidate clickables is identified. Note that these elements are called clickables but that any type of event such as `onmouseover` and `onmousedown` may be fired at them. This set may be identified in three ways:

- **Full auto scan mode** - The candidate clickables are identified based on their HTML element names, for example all div and span elements,
- **Annotation mode** - Candidate clickables are identified based on their annotation *crawljax="true"*. This mode allows clickables to be explicitly included or excluded from the crawling process,

[1] In this state-flow graph the vertices represent states and the edges represent clickables.

- **Configured mode** - The crawler is configured to identify certain elements using a domain-specific language.

After the set of candidate clickables is identified the page is loaded in an embedded browser and a robot fires an event on each item in the set. For each event the resulting DOM is compared to the DOM before the click. If the DOM has changed the state is recorded in the state-flow graph; if it is an existing state a pointer to the corresponding vertex is returned, otherwise a new state is added. For easy comparison a hash code is calculated for each DOM state. After the state is recorded in the state-flow graph an edge representing the clickable that caused the transition is added between the previous and current state.

Although crawlers are used by search engines in particular, they can be useful in other areas. Another area in which crawlers are being used on a regular basis is security [13, 18]. Many security tools have a crawler equipped to help them detect entry points by crawling the contents of a web page during the simulation of an attack. Most of these equipped crawlers should be adapted in order to support AJAX applications. The approach used by CRAWLJAX may be used as a foundation for adapting these crawlers.

4.1 ATUSA Plugin Framework

The latest version of ATUSA allows for the implementation of validation plugins [29]. These plugins must be written in Java, like ATUSA, and follow the design guidelines of the Java Plugin Framework (JPF)[2]. They have access to the active crawling session maintained by CRAWLJAX with the goal of validating the application after every event fired. The crawling session contains session-specific data like a reference to the embedded browser, the DOM of the current state and the state-flow graph of CRAWLJAX. Figure 4.1 depicts the processing view of ATUSA, CRAWLJAX and the plugins.

It is possible to create three types of plugins. These three types will be explained in the following sections. Each plugin must implement the Validation interface. This interface contains two methods:

- `validate()`: Perform the validation. This method has access to the current crawl session,
- `getReport()`: Returns a report of the actions performed by the plugin.

During this thesis project a number of security-related plugins will be implemented for ATUSA. They are described in Chapter 5 and 6.

4.1.1 Pre-crawling Validation Plugin

This type of plugin may access the crawling session before the crawling process has started. This may be required for a number of reasons. One of these reasons is that a website may require a user to log in before each session. By creating a pre-crawling plugin it is

[2]Java Plugin Framework project: `http://jpf.sourceforge.net/`

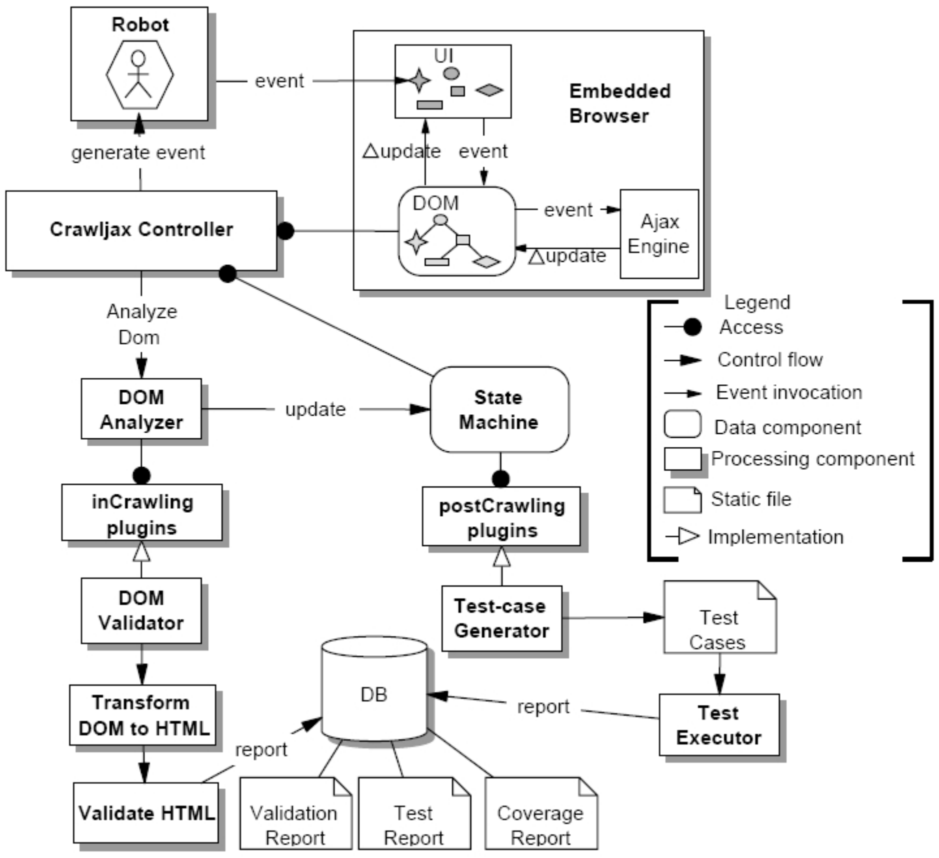


Figure 4.1: Processing view of ATUSA, originally taken from [29].

possible to enter the login details into a form on the page and submit that form before the automated crawling process starts. Another reason could be to initialize a logging system. The `validate()` method of this plugin is called once, right before the start of the crawling process.

4.1.2 In-crawling Validation Plugin

In-crawling plugins have access to the crawling session during the crawling process. After each DOM change the `validate()` method of each in-crawling plugin is executed. Possible applications of in-crawling validation are invariant checking and JavaScript error detection.

4.1.3 Post-crawling Validation Plugin

Post-crawling plugins have access to the crawling session after the crawling process has finished. The application of this type of plugin is very broad because the crawling session now contains a complete state-flow graph of the web application. This offers the plugin developer numerous possibilities like the creation of a sitemap or the analysis of unexpected paths in the application. Another possibility is to generate test cases from the state-flow graph [29]. The `validate()` method of post-crawling plugins is called exactly once, right after the crawling process has finished.

Chapter 5

Automatic DOM Change Violation Detection

Generalized scenario G1 from Section 3.2.2 describes the set of violations in which a malicious widget changes the DOM of another widget. These violations, in this thesis referred to as *DOM change violations*, can be detected on the client-side only. This makes detection of them difficult, as a user session must be simulated in order to detect the violations. Such a user session can be simulated with ATUSA, the tool which will be used in our approach to automatically detect DOM change violations.

In this chapter first the concept of our approach for automatically detecting DOM change violations will be explained. After this the implementation details of our approach as an ATUSA plugin will be discussed.

5.1 Concept

To propose an approach for automated detection of DOM change violations, it is important to define what a DOM change violation exactly is. We propose the following definition:

Definition 1 *A **DOM change violation** occurs when a widget's DOM tree is changed by another widget.*

To detect a DOM change violation we need to determine the following:

- The widget in which the change was initiated,
- The widget in which the DOM change took place,
- Whether the widget which initiated the change and the changed widget are different widgets.

In this chapter we propose an approach for determining these issues.

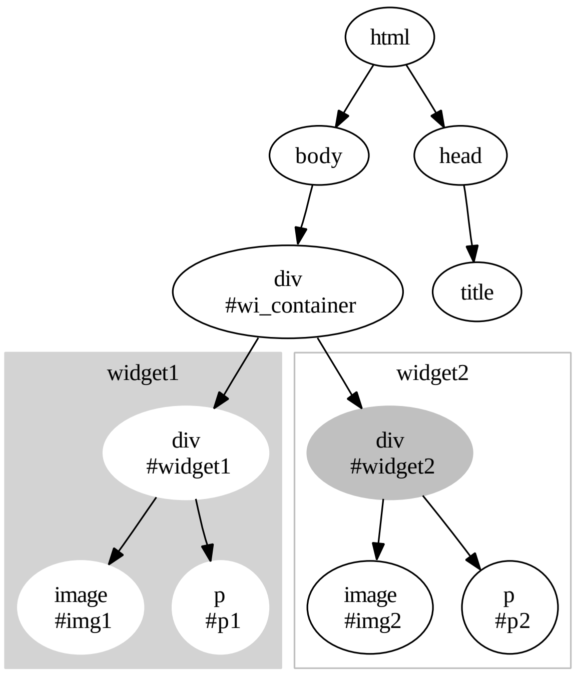


Figure 5.1: Example DOM tree for a widget framework.

5.1.1 Widget Boundary Detection

In order to be able to detect in which widget an action took place, and to be able to decide whether two detected widgets are the same, we need to define a clear and nonambiguous way of specifying a widget. To do this, it is important to realize exactly what a widget is. Figure 5.1 depicts the DOM tree of a simplified widget framework with two widgets. The elements which belong to `widget1` are `#widget1` and its child nodes, just as the elements of `widget2` are `#widget2` and its child nodes. The elements of `widget1` and `widget2` can be grouped by placing a boundary around them. This observation can be generalized into the following:

- Every widget has a boundary which encloses all its elements,
- Every element may only access the properties of the elements within the boundary of the widget it belongs to.

The widget boundary can be used as a foundation to decide whether two detected widgets are the same. We define a *widget boundary* to be the following:

Definition 2 *A **widget boundary** is the DOM node closest to the root of the DOM tree the nodes of that widget may manipulate.*

Definition 3 *The **widget boundary of a node** Node1 is the same as the widget boundary of the widget to which Node1 belongs.*

The widget boundaries in Figure 5.1 are the nodes labeled `#widget1` and `#widget2`. Automatic detection of these boundaries is not trivial. Although it is possible to determine these by, for example manually adding attributes in the code this is not preferable since this would require the tester to have access to the widget framework source code. This would make testing closed source widget frameworks like iGoogle impossible. A better way is to exploit the tree structure of the DOM. By exploiting the tree structure of the DOM and the characteristics of HTML it is possible to make the following assumptions while searching for the widget boundary of node *Node1*:

- A node either has only one or no widget boundary,
- If *Node1* belongs to a widget, the widget boundary of *Node1* always is either *Node1* or an ancestor of *Node1*,
- A widget boundary is likely to be a `DIV` element.

The first assumption is a consequence of the structure of a widget framework. An element on a page in a widget framework can either be inside a widget or outside a widget, for example navigation elements or framework code. Assuming that widgets cannot be placed inside each other, this means that in the DOM each node also has exactly one or no widget boundary.

The second assumption is a consequence of the tree structure of the DOM. Because each widget is a DOM subtree, we know that each widget element is either a node or the root node of that subtree. This leads to the assumption that the widget boundary of a node, when it exists, is either the node itself or an ancestor of the node.

The final assumption is a consequence of the structure of HTML and widget frameworks. As explained in Section 2.2.2, widgets are usually placed inside `DIV` elements or `IFrames`. Since we assume widgets are not placed inside `IFrames` in the frameworks under test, we may assume they are being placed inside a `DIV` element. It is however important to remain flexible in this assumption because this may change in the future, for example because of the introduction of new HTML elements [42].

By taking the above mentioned assumptions into account, it is possible to define a method for deciding to which widget a node belongs by identifying its widget boundary. In theory the node itself and all its parent nodes may be the widget boundary, but in practice it is possible to filter out many candidate nodes. In Figure 5.1 it is easy to see that the `html`, `body` and `#wi_container` nodes are not widget boundaries although they are ancestor nodes of each widget node. By defining rules we can exclude these items from the boundary identification process. In addition, in most widget frameworks, it is possible to identify a widget boundary with high confidence. An example of this is the Exact Widget Framework (Section 2.3) in which, at the time of writing, a widget boundary always has the class attribute `ex_widget`. An important consideration when using such rules is that the identification of a widget boundary may not always be unambiguous. It is possible that more than one node will be identified as a widget boundary when rules overlap. An example of this is the rule `class=widget`, which will match the node `<div id=widget1 class=widget>` as well as the node `<a id=link class=widget>`. Therefore identification using rules will result in a list of possible widget boundaries. In order to assign the confidence with which a rule matches a widget boundary, identification strength values can be added. These values can be one of the following: `LOW`, `MEDIUM`, `HIGH`, `VERY_HIGH` and `EXCLUDE`. A rule marked with `EXCLUDE` indicates that a node which matches it must be excluded from the identification process.

Applying these ideas results in an algorithm for defining to which widget a node belongs, described by Algorithm 1.

Algorithm 1 IdentifyWidgetBoundary(fingerNode)

```
$rules \Leftarrow getIdentificationRules()$
$node \Leftarrow fingerNode$
while $node \neq null$ do
   $identificationStrength \Leftarrow node.match(rules)$
   if $identificationStrength \neq null$ && $identificationStrength \neq EXCLUDE$ then
      $widgetBoundaries.add(node, identificationStrength)$
   end if
   $node \Leftarrow node.getParent()$
end while
```

Figure 5.2 demonstrates the result of the identification process for node *p1* using the rules as defined in the figure. In the figure *#widget1* has been identified as a widget boundary with very high confidence. By defining a different set of rules for every framework it is possible to identify widget boundaries in every widget framework with high confidence.

5.1.2 DOM Change Violation Detection

To recall the beginning of this chapter, the following needs to be determined to detect a DOM change violation:

- The widget in which the change was initiated,
- The widget in which the DOM change took place,
- Whether the widget which initiated the change and the changed widget are different widgets.

In our approach, we assume a DOM change is initiated by an event fired on a DOM element. It is also possible to initiate a DOM change with a full page refresh, but this is rarely required in current single-page web interfaces. Other ways of initiating a DOM change are discussed in Chapter 8. To determine the widget in which the DOM change was initiated, the node on which the last event was fired must be determined first. The *active element*, the element on which the last event was fired, can be retrieved from CRAWLJAX as described in Appendix B.1.

Definition 4 *The **active element** is the element on which the last event was fired.*

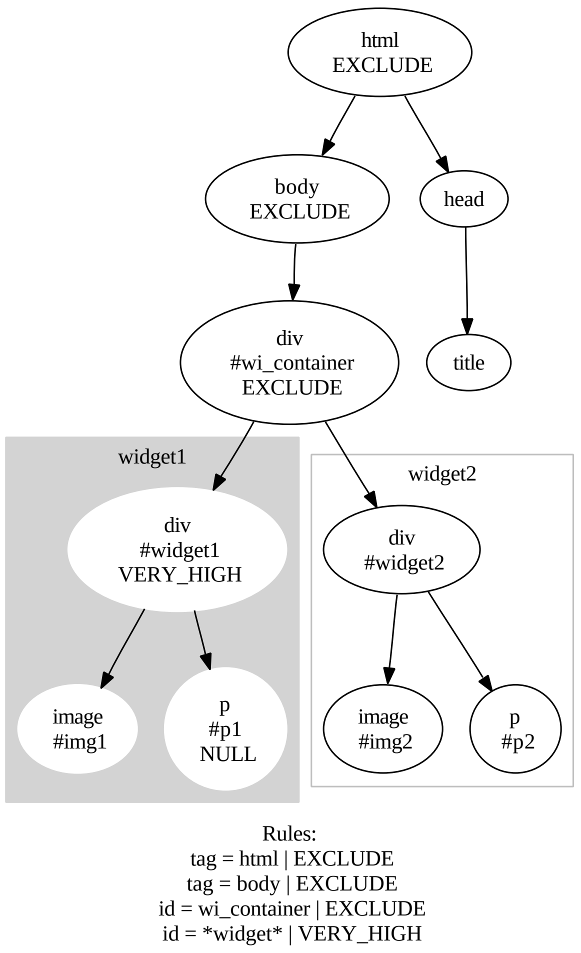


Figure 5.2: Widget boundary identification process for *#p1*.

It is possible to determine the widget boundary of the *active element* using Algorithm 1. The node(s) in which the DOM change(s) took place can be determined by comparing the DOM before the event and the DOM after the event is fired. Using Algorithm 1 with each changed DOM node as input, the widget(s) in which the DOM change(s) took place can be determined.

By comparing the widget boundaries of the widget which initiated the DOM change and the widget(s) in which the DOM took place, we can automatically decide whether a DOM change violation occurred. We propose Algorithm 2, `IsDomChangeViolation`, for automatically deciding whether a DOM change violation occurred. If the detected widget boundaries are the same, the DOM change was in the same widget as the initiation of the change, which is allowed. If the detected boundaries are different, the DOM change was in another widget than it was initiated in, which is a DOM change violation. In Section 5.2 the implementation of this approach for automatic detection of DOM change violations with

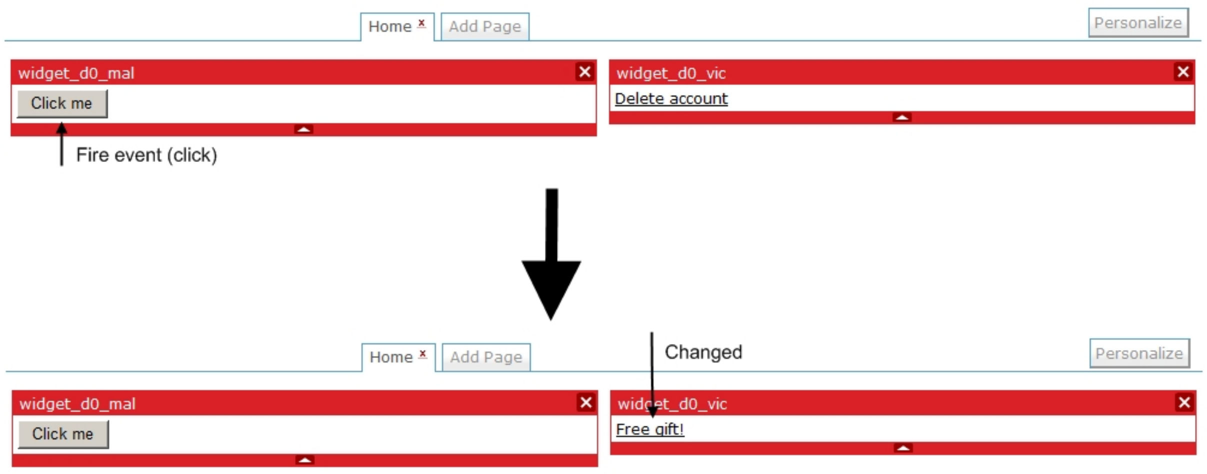


Figure 5.3: Example scenario: Changing the description of a link.

ATUSA is explained. In the next section an example scenario of a DOM change violation detection using Algorithms 1 and 2 is given.

Algorithm 2 IsDomChangeViolation(activeElement, changedNode)

```
$eventWidget \Leftarrow IdentifyWidgetBoundary(activeElement)$
$changedWidget \Leftarrow IdentifyWidgetBoundary(changedNode)$
**if** $eventWidget \neq changedWidget$ **then**
    **return true**
**else**
    **return false**
**end if**
```

5.1.3 Example Scenario

One of the scenarios from Section 3.2 which describes a DOM change violation is Scenario 3.2.5, in which an event fired on an element in a malicious widget changes the description of a link in the victim widget. Figure 5.3 demonstrates this scenario. When a user clicks on the ‘Click me’ button, the description of the link ‘Delete account’ changes to ‘Free gift’. The (simplified) DOM tree for this situation, before the event is fired, is given in Figure 5.4.

The event will be fired on the `button 'Click me'` node, and the `a 'Delete account'` node will change because of this event. By using Algorithm 2 we can show this is a DOM change violation. First a set of identification rules for identifying a widget boundary must be declared. Without these rules it is not possible to automatically detect a DOM change violation, because the algorithm would not have a notion of widget boundaries in that case.

From inspection of Figure 5.4 can be deduced that the `html`, `body` and `#wi_container` nodes are not to be included in the identification process because they do not belong to

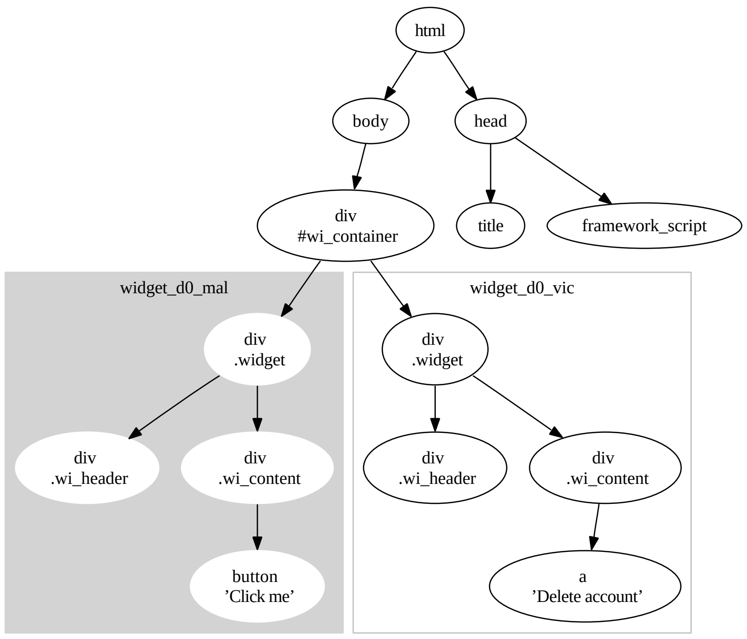


Figure 5.4: Example scenario: Changing the description of a link, simplified DOM tree.

```
tag = html | EXCLUDE
tag = body | EXCLUDE
id = wi_container | EXCLUDE
class = widget | VERY_HIGH
```

Figure 5.5: Rules for widget boundary identification in Figure 5.4.

widgets. We can also deduce that the nodes labeled `.widget` are very likely to represent a widget boundary. Therefore we run Algorithm 2, `IsDomChangeViolation`, with the rules defined in Figure 5.5.

The first step is to identify the widget in which the event was fired. This can be done using Algorithm 1, `IdentifyWidgetBoundary`, with node `button 'Click me'` as input, which will result in node `div .widget` in the `#widget_d0_mal` area. Note that during the node traversal in the identification process none of the nodes of `widget_d0_vic` were traversed.

The next step is to identify the widget in which the DOM change took place. The result of comparing the DOM before and after the event is the node labeled `a 'Free gift'`. Using Algorithm 1 with this node as input, the widget boundary is identified as the `div .widget` node in the `#widget_d0_vic` area. This boundary is not equal to the boundary of the widget in which the event was fired, which means a DOM change violation was detected.

5.2 Implementation

The approach for automatically detecting DOM change violations discussed in 5.1 was implemented as a plugin for ATUSA. In this section the implementation details are discussed.

5.2.1 DOM Handling in CRAWLJAX

To implement Algorithms 1 and 2, for widget boundary detection and DOM change violation analysis in ATUSA, it is important to consider how CRAWLJAX handles the DOM. CRAWLJAX has access to three DOM objects:

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

These objects are created as depicted in Figure 5.6. The internal browser DOM is not recorded, but it is directly accessible using reflection as described in Appendix B.2. It is important to realize that *domBeforeEvent* and *domAfterEvent* are both copies of the internal browser DOM and therefore the following conditions hold:

- A reference to a node in *domBeforeEvent* does not point to a node in *domAfterEvent* and vice versa,
- A reference to a node in *domBeforeEvent* or *domAfterEvent* does not point to a node in the internal browser DOM and vice versa,
- A change made in a DOM object will exist in that specific object only.

5.2.2 Widget Boundary Detection

As described in Algorithm 2, for DOM change violation detection, we need to identify two widget boundaries. The first, the boundary of the widget which initiated the DOM change, can be detected using Algorithm 1, `IdentifyWidgetBoundary`, with the active element as input. CRAWLJAX returns a reference to the active element, which is a node in *domBeforeEvent*.

The second widget to detect is the changed widget. To find this widget, the changed node(s) must be detected. An excellent library for XML/HTML comparison is XmlUnit[1]. XmlUnit takes a control document (*domBeforeEvent*) and a test document (*domAfterEvent*) as input. The test document is compared to the control document, and all nodes in the test document which differ from the control document are returned. This means that the result of the comparison of *domBeforeEvent* and *domAfterEvent* is a list of nodes, which are members of the *domAfterEvent* object. Figure 5.7 shows an example of XmlUnit output for the comparison of two DOM instances. Note that it is possible to use *domAfterEvent*

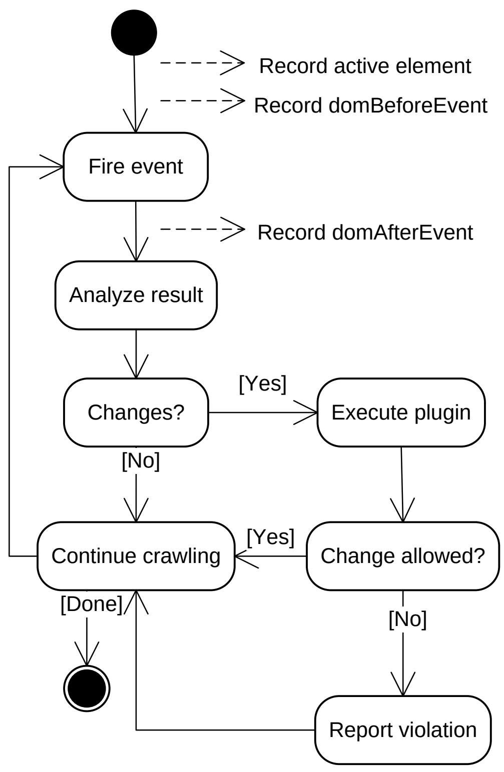
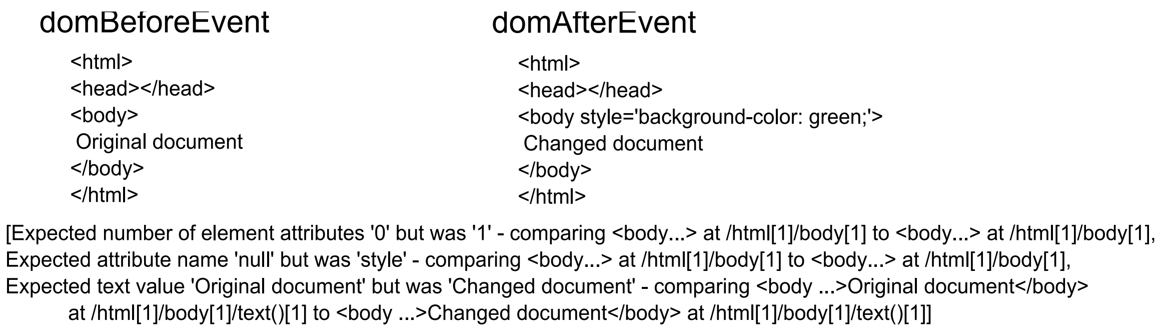


Figure 5.6: DOM recording in the crawling process.



```
domBeforeEvent                          domAfterEvent
    <html>                                  <html>
    <head></head>                           <head></head>
    <body>                                  <body style='background-color: green;'>
     Original document                       Changed document
    </body>                                 </body>
    </html>                                 </html>
[Expected number of element attributes '0' but was '1' - comparing <body...> at /html[1]/body[1] to <body...> at /html[1]/body[1],
Expected attribute name 'null' but was 'style' - comparing <body...> at /html[1]/body[1] to <body...> at /html[1]/body[1],
Expected text value 'Original document' but was 'Changed document' - comparing <body ...>Original document</body>
    at /html[1]/body[1]/text()[1] to <body ...>Changed document</body> at /html[1]/body[1]/text()[1]]
```

Figure 5.7: Example XmlUnit output.

as control document and *domBeforeEvent* as test document, but that this would lead to confusing and counterintuitive error messages.

Although the two widget boundaries can be identified using Algorithm 1, they are members of different objects, which means they cannot be naively compared. A possibility would be to compare the attributes of the widget boundary nodes. This would work in frameworks in which each node can be uniquely identified based on its attributes but this is certainly not always the case. Another problem with this approach is that it would fail if a widget contains a node with attributes equal to the widget boundary.

This leads to the observation that two boundaries must be references in the same DOM object to successfully compare them. This can be achieved in two ways: by identifying the changed element(s) in *domBeforeEvent* or by identifying the active element in *domAfterEvent*. The first way often fails because it is likely that the changed node does not exist nor can be identified in *domBeforeEvent*. Therefore the best method is to identify the active element in *domAfterEvent*. Because this element may also not exist in *domAfterEvent*, we decided to identify the widget boundary of the active element in *domAfterEvent* instead. The idea is that the widget boundary should exist after firing an event, assuming that widgets are not being removed during the session.

A way to identify an element in the DOM is by identifying it based on its unique attribute values. As we have stated, in general not all elements in a widget framework can be uniquely identified by their attribute values. Therefore such a unique value must be annotated to use this solution. In CRAWLJAX, it is possible to access and change the *realDom*. After annotating a unique attribute value in the widget boundary in *realDom*, this value will be copied along when *domAfterEvent* is recorded. Since the widget boundary now contains a known unique value, it can be identified in *domAfterEvent*, and it can be compared to the widget boundary of the changed widget. The details of attribute annotation implementation are discussed in Appendix B.4. We define the following for use throughout the rest of this thesis:

Definition 5 *An* ***annotated DOM*** *is a DOM in which one or more unique attributes were annotated with the purpose of identifying the elements after an event is fired.*

Definition 6 *An* ***annotated node*** *is a node in which one or more unique attributes were annotated with the purpose of identifying the elements after an event is fired.*

5.2.3 DOM Change Violation Detection

After the annotation of the unique attribute to identify the active element in *domAfterEvent*, and the analysis of the changed node(s), the DOM change violation detection algorithm can be executed. For ease of implementation a small adjustment was made to Algorithm 2. Instead of identifying the widget boundary for each changed node, and comparing it with the widget boundary of the active element, the violation analysis is done during the identification process. The idea is that, while traversing the DOM tree to the root from a node, the traversal should come across a node which was annotated during the identification process

[1]XmlUnit: `http://xmlunit.sourceforge.net/`

of the widget boundary of the active element. If this is not the case, a DOM change violation has occurred. Algorithm 3 describes the steps of the implemented method for automatically detecting DOM change violations.

Algorithm 3 AnalyzeDomChangeViolation(changedNode)

```
Require: The widget boundary of the active element is annotated in the DOM tree of changedNode
  $rules \Leftarrow getIdentificationRules()$
  $node \Leftarrow changedNode$
  while $node \neq null$ do
    if $node.isAnnotated()$ then
      return $noViolation$
    end if
    $node \Leftarrow node.getParent()$
  end while
  return $violation$
```

To prevent the detection of a high number of false positives, a mechanism for excluding sets of active elements from the analysis was implemented. This mechanism allows for the configuration of rules for excluding elements, for example because they belong to the menu of a widget framework. We define these elements as trusted elements.

Definition 7 *A **trusted element** is an element which belongs to the framework.*

An event on such a trusted element may cause a DOM change but since it was not caused by an inter-widget interaction, our approach does not apply for validating the change. These rules must be defined using the same format as CRAWLJAX uses for tag exclusion.

5.2.4 Selenium Test Suite Generator

In order to be able to easily replay and inspect a violation, a Selenium test case is generated for each detected violation. Selenium[2] is a web application testing system, which allows for the recording and replaying of test cases. Test cases are stored in HTML format, which means they can also be generated in other ways than using the recorder. SeleniumIDE is a Firefox plugin which allows for easy replay of a Selenium test case. Selenium test cases are generated using a template and the Apache Velocity[3] template engine. When a violation is detected, a test case is generated using the event sequence, which caused the violation. The event sequence exists of a list of XPath queries and events fired.

[2]Selenium: `http://seleniumhq.org/`

[3]Apache Velocity: `http://velocity.apache.org/`

Chapter 6

Automatic HTTP Request Violation Detection

Generalized scenario G2 in Section 3.2.2 describes the set of violations in which a widget attaches an event, which initiates an HTTP request, to another widget. An example of this is a key logger, which can be implemented by attaching an `onkeydown` event to the input field of a form. As explained in Section 3.2.1, the logged keys must be communicated to the server. If this does not happen, the data cannot reach the malicious user. Therefore the action will only be classified as a violation if it results in an HTTP request. In this chapter this type of violation, called *HTTP request violation*, and our approach for detecting it will be explained.

6.1 Concept

In order to propose an approach for automatically detecting HTTP request violations, it is important to define exactly what an HTTP request violation is. We propose the following definitions:

Definition 8 *An **HTTP request violation** occurs when an HTTP request originates from a widget which was not allowed to send that request.*

Definition 9 *A widget w is the **origin** of a request r if either an event fired on an element in w or the `src` attribute of an element in w triggers r.*

An example of code which causes an HTTP request violation is demonstrated in Figure 6.1. In this example MAL, the malicious widget, attaches an event handler to an input field in VIC, the victim widget. This event handler sends the key pressed for each key stroke in that input field to the URL specified by MAL. Based on VIC's original behaviour, this could be an HTTP request violation.

To automatically detect the violation described in Figure 6.1, we need to be able to automatically detect that MAL attaches an event handler to an element of VIC. It is important to realize that this action does not affect the DOM tree of VIC, and can therefore not be detected using Algorithm 3, `AnalyzeDomChangeViolation`.

```
// code in MAL
var url = "http://www.maliciousdomain.com/script.aspx";
var vic = document.getElement("VIC.inputField");
vic.onkeydown = sendData(url, event.getKey());
```

Figure 6.1: Example code which causes an HTTP request violation.

One method for automatic detection would be to statically analyze the JavaScript of every widget, but as explained in Section 9.1 this is not preferred. A different method is to exploit the assumption that data which is never communicated to the server remains safe. If a malicious user wants to steal the data, it should be sent to the server. This means that instead of detecting the attachment of an event handler, it is possible to listen for requests made by a widget. Throughout the remainder of this thesis we assume a widget may make HTTP requests only, although our approach would be applicable to other request protocols (such as FTP). After making this assumption, the following must be determined to detect an HTTP request violation:

- The widget from which the request originates,
- Whether the widget has permission to trigger the request.

In the next sections we propose an approach for automatically detecting HTTP request violations. First the concept of our approach for identifying the origin widget of an HTTP request will be explained. Finally we will discuss our approach for deciding whether this widget was allowed to trigger the request. In Section 6.2 the implementation details of our approach in ATUSA are explained.

6.1.1 HTTP Request Identification

The main challenge of detecting the origin widget of a request is to couple the request with the element from which the request originated. This is not trivial because HTTP requests do not carry any information about the DOM or the element which made the request. In addition, an HTTP request can be triggered in many ways. The following are the most important:

- The `src` attribute of an element, for example in a `SCRIPT` or `IMG` element,
- An AJAX call which is attached to an element as an event handler.

The first step to be taken is to intercept all HTTP requests. This can be done using an HTTP proxy [2], which is placed between the client and web server. Figure 6.2 depicts this situation. The proxy can provide ATUSA with all requests made by the web application by maintaining a buffer of all intercepted requests. The challenge is to couple these requests with the element from which they were triggered.

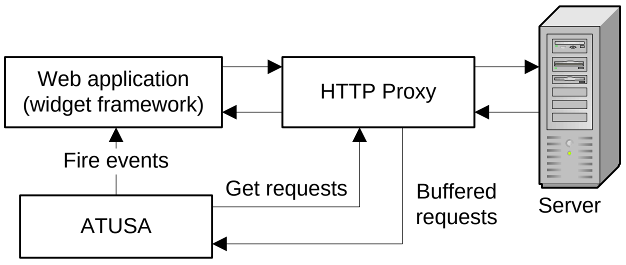


Figure 6.2: Situation with an HTTP proxy between client and server.

Because ATUSA only calls its in-crawling plugins after a DOM change, it is not possible to assume that all requests in the proxy buffer were caused by the last event fired by ATUSA. Many events which trigger requests may be fired before a DOM change occurs, which means that the buffer may contain requests which originate from another event than the one fired last. Therefore another method for identifying requests should be introduced.

The only way to attach information about the DOM to an HTTP request, without affecting the behaviour of the web server which handles the request, is by adding data to its query string (e.g. `?wid=123&id=widget1`). This data should be selected carefully to ensure it does not interfere with other parameters which are being sent to the server, for example by using a non-standard variable name. An example of data which could be added to the query string is the ID of the element that triggered the request. This data could then be extracted from the request and used by ATUSA, to identify the element from which the request was sent in the DOM. To be able to unambiguously identify an element, this approach requires all elements to have a unique ID. If elements are allowed to share IDs, we cannot automatically decide which of the sharing elements triggered the request. We propose Algorithm 4, `IdentifyRequestOriginWidget`, for identifying the widget boundary of the widget from which a request was sent, in a DOM in which all elements have a unique ID. This algorithm is used to identify the origin widget of each request in the proxy buffer, every time the HTTP request violation detection plugin is called by ATUSA.

Algorithm 4 IdentifyRequestOriginWidget(request)

```
Require: All elements in the DOM have a unique ID.
  $id \Leftarrow request.getQueryString().getElementId()$
  $activeNode \Leftarrow dom.getElement(id)$
  return $IdentifyWidgetBoundary(activeNode)$
```

6.1.2 HTTP Request Validation

After the widget from which the request was sent is identified, the request must be validated to see if the widget was allowed to trigger it. In order to do this, 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 URL list of URLs which may be requested [20]. In our approach a list of allowed request URLs is created automatically for each widget by running CRAWLJAX on that widget in an isolated environment. In this situation every request intercepted by the proxy can be assigned to that specific widget or the widget framework. At the end of the crawling process, the proxy buffer should contain all the requests the widget is allowed to trigger when it is placed on the personal homepage of a user. This list can be saved, and retrieved during the validation phase of a request. If during validation a request URL does not exist in the allowed URL list of its origin widget, the widget does not have permission to trigger the request and an HTTP request violation occurred. Algorithm 5 describes this idea.

Definition 10 *The **allowed URL list** of a widget contains all the URLs of the requests it is allowed to trigger throughout its lifetime.*

Algorithm 5 AnalyzeHttpRequestViolation(request)

```
Require: Request contains the unique ID of an element in the DOM.
  $widget \Leftarrow IdentifyRequestOriginWidget(request)$
  $list \Leftarrow widget.getAllowedUrlList()$
  if $list.contains(request.getUrl())$ then
    return $noViolation$
  else
    return $violation$
  end if
```

Note that this approach also works for requests which do not originate from a widget, for example because they are triggered by the framework. By running ATUSA on 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.

6.1.3 Example Scenario

One of the scenarios from Section 3.2 which describes an HTTP request violation is Scenario 3.2.1, in which a malicious widget attaches an `onkeydown` event handler to an input field of a form in the victim widget. When characters are entered into the input field, the event handler captures them and sends them to an URL specified by the malicious user. Figure 6.3 demonstrates this scenario.

The simplified allowed URL lists for the scenario depicted by Figure 6.3 are displayed in Figure 6.4. The allowed URL list of the framework was generated by recording all URLs requested during the execution of ATUSA on the framework with only an empty widget in it. The allowed URL lists of each widget was generated by recording the URLs while running ATUSA on them in an isolated framework environment, as explained in Section 6.1.2.

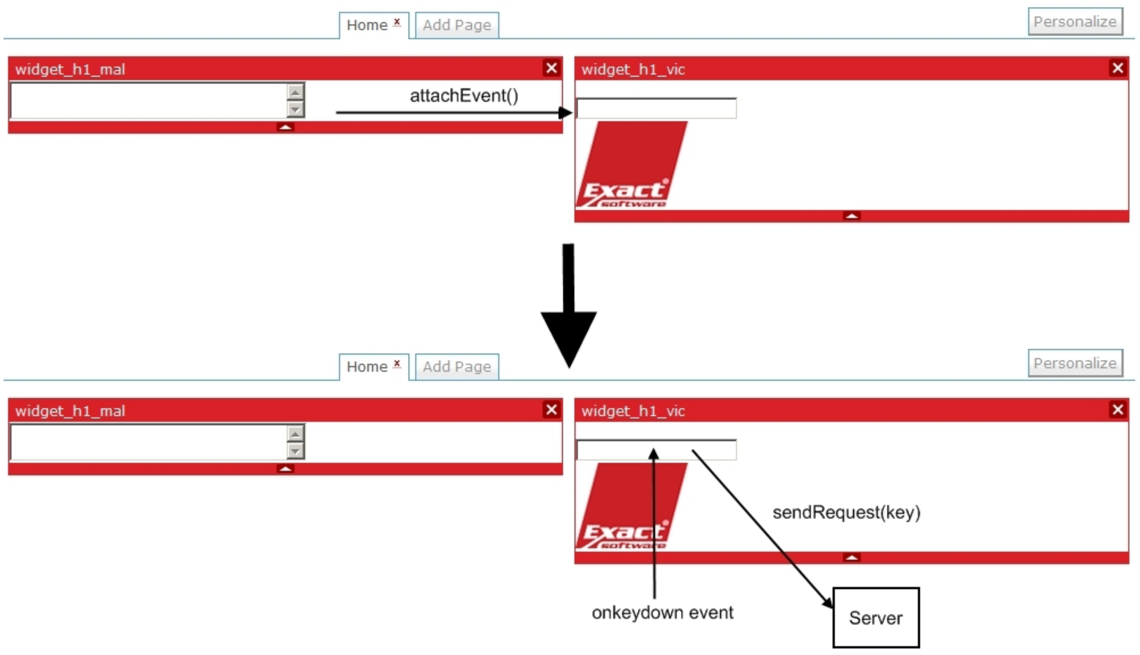


Figure 6.3: Example scenario: Stealing a password using a key logger.

```
AllowedList framework = {http://www.efw.com/widget_close.jpg,
                         http://www.efw.com/widget_bg.jpg,
                         http://www.efw.com/framework.js}
AllowedList widget_h1_vic = {http://www.efw.com/h1_vic/logo.jpg,
                             http://www.efw.com/h1_vic/widget_h1_vic.js}
AllowedList widget_h1_mal = {http://www.efw.com/h1_mal/logo.jpg,
                             http://www.efw.com/h1_mal/widget_h1_mal.js}
```

Figure 6.4: Allowed URL lists for the scenario depicted by Figure 6.3.

When a key is pressed in the input field of `widget_h1_vic`, the event handler attached by `widget_h1_mal` captures the event and sends the key to a URL specified by the creator of `widget_h1_mal`. Assuming a DOM change happens during the crawling process, ATUSA is able to detect this violation using our approach.

In this example scenario we assume each element has a unique ID, which is added automatically to each request which originates from that element. A simplified request made by the key logger is depicted by Figure 6.5. When this request is retrieved by ATUSA, Algorithm 4 for origin widget identification is started. First the ID is extracted from the query string. This ID is used to find the corresponding DOM element, in this case the input field in `widget_h1_vic`. Using this element the origin widget can be identified with Algorithm 1, `IdentifyWidgetBoundary`. Finally, by checking whether the allowed URL list for `widget_h1_vic` contains the requested URL, ATUSA can conclude the request is an

```
POST /widget_h1_mal HTTP/1.1
Content-Type: application/x-www-form-urlencoded
User-Agent: Mozilla/4.0
Host: www.malicious.com
Content-Length: 27
keycode=100&id=input_h1_vic
```

Figure 6.5: Request made by a key logger including the ID of the origin element.

HTTP request violation because the requested URL is not in the allowed URL list of the origin widget.

6.2 Implementation

The approach discussed in Section 6.1 was implemented as a number of plugins for ATUSA. In this section the implementation details of these plugins are discussed.

6.2.1 HTTP Request Identification

In order to automatically identify a HTTP request the following should be implemented:

- HTTP proxy for interception of all HTTP requests,
- Mechanism for the enforcement of unique attributes in each DOM element,
- Attachment of the unique attribute to HTTP requests.

These issues are implemented as a number of plugins for ATUSA. In this section the implementation details of these plugins are discussed.

HTTP Proxy

An important aspect of Algorithm 4 for identifying the origin widget of a request, is the ability to intercept all HTTP requests. This requires the implementation of an HTTP proxy. Many security testing and assessment approaches which use open source HTTP proxies exist [4, 20, 34, 36]. For the implementation of this project we decided to use the proxy functionality of WebScarab[1], a tool for the analysis of web applications. WebScarab has a plugin model, which allows for the creation of plugins to handle requests and responses passing through the proxy. We have integrated WebScarab into ATUSA in such a way that it can be started and configured through ATUSA.

To allow ATUSA to access the requests which pass the proxy, a request buffer plugin was implemented. This plugin buffers all requests until ATUSA retrieves the buffer. After the buffer is retrieved, the contents of the buffer are cleared and the buffer starts buffering

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

```
// before annotation
<script src="javascript.js"></script>

// after annotation
<script src="javascript.js?requestForProxyId=12345"
        requestForProxyId="12345"></script>
```

Figure 6.6: Example annotation of the unique attribute annotation proxy plugin.

again. In this way the buffer always contains all the requests made since the latest DOM change in the crawling process of ATUSA.

Attaching unique attributes

In order to decide whether an HTTP request was an HTTP request violation, a mechanism for coupling the request with its origin widget must be implemented. Algorithm 4, `IdentifyRequestOriginWidget`, indicates all elements in the DOM must have a unique ID. As explained in Section 5.1.1 this is not always the case.

To enforce a unique attribute in all DOM elements an annotation mechanism must be applied. To recall from Section 6.1.1, the most important ways to trigger an HTTP request are:

- The `src` attribute of an element, for example in a `SCRIPT` or `IMG` element,
- An AJAX call which is attached to an element as an event handler.

To add the unique attribute of the origin element to the request, these types of triggering HTTP requests should be treated in different ways. In order to avoid collision with existing attributes the attribute `requestForProxyId` will be added rather than the `id` attribute.

HTTP requests which are triggered by the `src` attribute of an element are usually not initiated by an event. Examples are the `<script>`, `<img>` and `<style>` elements. When the browser parses such an element, the corresponding HTTP request is executed immediately. Therefore, the only way of adding information to such a request is by manipulating the `src` attribute before the page reaches the browser. Because ATUSA fetches the page after the browser is done loading, it is not possible to annotate the value using ATUSA. To avoid this problem we have implemented a proxy plugin which parses a response and appends the unique value to all `src` attributes. For convenience the attribute is also annotated into the element because this simplifies identification of the element using XPath. Figure 6.6 shows an example annotation done by the proxy plugin.

HTTP requests made using an AJAX call are the most essential form of sending HTTP requests during navigation in modern single-page web applications [26], which widget frameworks are. These requests are often triggered by an event. Although they could be

```
var oldXhr = XmlHttpRequest;
function XmlHttpRequest(vars){
        var qStr = vars + "&requestForProxyId=";
        qStr += event.target.getRequesForProxyId();
        oldXhr(vars);
}
```

Figure 6.7: Simplified example of code used to subvert AJAX.

annotated using the same approach as used for `src` attribute annotations, this has a limitation. Because in this approach annotation is done by the proxy, elements which are added dynamically on the client-side are not annotated with a unique attribute. It is also difficult to trace a request back to the original situation from which the request was sent, because the annotated attributes remain constant throughout the crawling session. Because the DOM is dynamic in AJAX applications, many DOM changes may have been made before an AJAX call is triggered. To be able to trace a request back to the exact state from which it was sent, dynamic annotation is preferable. By using the same annotation technique as used during widget boundary detection (Section 5.1.1), this can be accomplished. The source code of ATUSA was extended using aspects in such a way that a unique attribute is annotated into the next active element, right before the event is fired. This has the advantage that a request on the proxy can be traced back to the original situation in which it was triggered, because each unique value exists in one DOM situation only. However, this does require to keep a history of DOMs (see Appendix B.5).

After annotation of the attribute into the element, the value must be appended to all HTTP requests the event triggers. We implemented a technique known as *Prototype Hijacking* [33], in which the AJAX call is subverted using a wrapper function. During the subversion a variable is added to the query string of the AJAX call. The value of this variable is retrieved, using JavaScript, from the element on which the event that caused the AJAX call was fired. Figure 6.7 shows a simplified version of the code used to subvert AJAX.

Di Paola and Fedon [33] propose a method of subverting AJAX by subverting the XmlHttpRequest[2] prototype. Unfortunately this approach does not work in Internet Explorer because IE does not allow prototype manipulation of the XmlHttpRequest object. Therefore an instance of the object should be subverted rather than the prototype itself. The consequence of this is that for every AJAX framework used, different subversion code is required. We have implemented subversion code for the ASP.NET AJAX[3] and jQuery[4] frameworks (see Appendix E). The subversion code is added automatically by a proxy plugin which adds the code to every HTML response page.

[2]The XmlHttpRequest is the JavaScript object used to send AJAX calls [6].
[3]Microsoft ASP.NET AJAX framework: `http://www.asp.net/ajax/`
[4]jQuery: `http://jquery.com/`

6.2.2 Allowed URL List Generation

An important step of HTTP request violation detection using our approach is the automated generation of allowed URL lists. The allowed URL list of a widget contains all URLs it may request during its existence on the page. This list is retrieved and used during the validation of requests. We have automated the generation of this list with a post-crawling plugin for ATUSA. This plugin fetches the request buffer from the proxy and generates a list of URLs from it. For this purpose a complete buffer mode, which buffers all requests since the beginning of the crawling process, was added to the buffer proxy plugin. By placing every widget on its own on the page, an allowed URL list can be generated for all widgets in the framework. These lists can be saved in a variety of ways, as long as they can be retrieved for a widget using information from the DOM. In our implementation we parse the widget title and use it as filename for the list. We assume that when a widget has permission to request a URL, the request is without limitations. Therefore we ignore parameters while generating the allowed URL list.

6.2.3 HTTP Request Validation

After the identification details are added to the requests, the final step of the HTTP request violation detection algorithm is validating the request. This process is implemented as an in-crawling validation plugin for ATUSA. Every time the plugin is started, the proxy buffer is requested. For each request in the buffer the `requestForProxyId` is extracted. The corresponding value is looked up in the DOM history (see Appendix B.5) using XPath, starting with the latest DOM state. If the element is found, the widget boundary is identified using Algorithm 1, `IdentifyWidgetBoundary`, and the allowed URL list for that widget is retrieved. Finally the retrieved list is used to validate the request.

Chapter 7

Empirical Evaluation

In this chapter our approach is evaluated, based on three performed case studies. The first case study is done on a simplified widget framework-like test suite, which was created with the purpose of ensuring the correct implementation of our approach. The second case study is performed on the EWF, the widget framework which is currently researched by Exact. The third case study is performed on Pageflakes, a popular industrial widget framework.

First the research questions which will be addressed during this evaluation will be stated. After this the setup of the system during the case studies will be explained. Finally we will discuss the case studies and their evaluation results.

7.1 Questions

The case studies were performed with the goal of answering the following research questions:

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

RQ2: How well does our analysis perform?

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

7.2 Test Setup

All case studies were performed on an Intel Core 2 6400 2.13 Ghz CPU with 2 GB RAM. Because ATUSA runs best on Windows XP[1], and the EWF runs only in Vista, we decided to run Windows XP in a virtual machine in Vista using Virtual PC 2007[2]. In this setting, ATUSA uses the XP version of Internet Explorer 7 as internal browser.

[1] ATUSA does run on Vista, but the package it uses to interact with the internal browser only works on Vista if the browser security settings are turned to the lowest level.

[2] Virtual PC 2007: `http://www.microsoft.com/virtualpc`

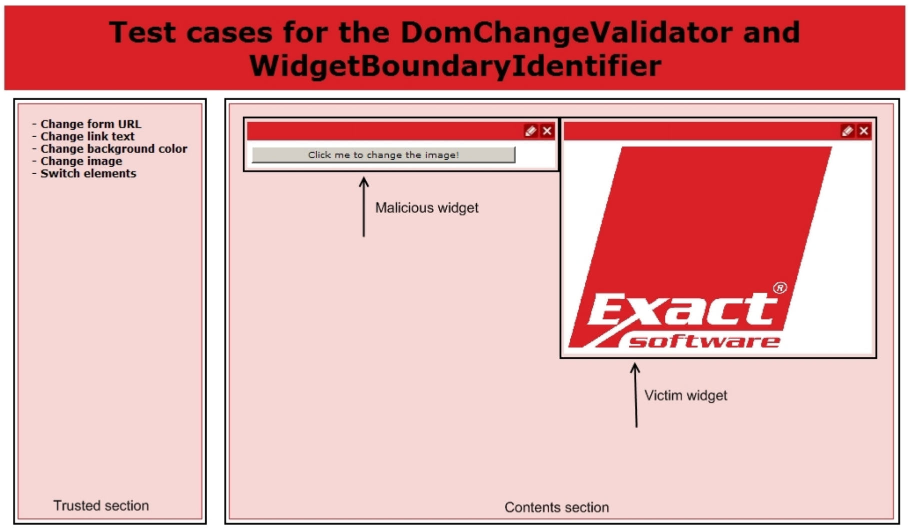


Figure 7.1: Screenshot of a DOM change violation widget pair in the simplified widget framework.

The widget frameworks used in the case studies were all implemented in ASP.NET. The simplified widget framework and the EWF were installed on a local IIS 7 server. The Pageflakes case study was performed using the production website of Pageflakes[3].

7.3 Case Study 1: A Simplified Widget Framework

In order to be able to test the correct behaviour of our implementation, a simplified widget framework was created. This framework does not allow dragging and dropping of widgets, but it produces the required structured output pages. Figure 7.1 shows a screenshot of the simplified framework. All elements in the menu section open two widgets in the contents section, using an AJAX call. Each widget pair contains a malicious widget (MAL) and a victim widget (VIC).

7.3.1 DOM Change Violation Widget Pairs

In the DOM change violation test widget pairs, every malicious widget contains an event handler which changes the DOM of its corresponding victim widget.

The following DOM change violation widget pairs were implemented for the simplified widget framework:

SD1: An event fired in MAL changes the action URL of a form in VIC

SD2: An event fired in MAL changes the link description of a link in VIC

[3]Pageflakes: `http://www.pageflakes.com`

```
<div class="widget" id="widget_image_1">
  <div>
    <img src="widget_header.jpg">
  </div>
  <div class="widget_contents">
    <input type="button" onclick="changeLogo();"
           value="Click␣me␣to␣change␣the␣image!"
           style="width:␣300px;" />
  </div>
</div>
<div class="widget" id="widget_image_2">
  <div>
    <img src="widget_header.jpg">
  </div>
  <div class="widget_contents" id="widget_contents">
    <img src="exact.jpg" id="logo" style="logo">
  </div>
</div>
```

Figure 7.2: DOM of the contents section of Figure 7.1.

SD3: An event fired in MAL changes the background color of VIC

SD4: An event fired in MAL changes an image in VIC

SD5: An event fired in MAL switches two DOM elements in VIC

Figure 7.1 shows the widget pair for case SD4, in which a click on the button in MAL changes the image in VIC. Figure 7.2 shows the DOM of the contents section. Widget boundaries in the simplified widget framework were designed to have the `class="widget"` attribute. The elements in the menu section were designed to have the `class="menu"` attribute. Therefore the rule `class=widget|VERY_HIGH` was added to the ATUSA widget boundary identification rules, and the `class=menu` rule was added to the trusted items configuration.

7.3.2 HTTP Request Violation Widget Pairs

To test the HTTP request violation detection, only one widget pair was implemented for the simplified widget framework. This is because for HTTP request violation detection, it is necessary to isolate a widget on the page to record its allowed URL list. This is difficult to achieve without dragging and dropping functionality, and therefore only one widget pair was implemented for this type of detection in this case study. The following HTTP request violation widget pair was implemented for the simplified widget framework:

SH1: MAL attaches an event handler to an element in VIC, which fires an HTTP request when the element is clicked

To run this widget pair, first an allowed URL list was generated for VIC by running it inside an isolated environment. The HTTP request violation widget pair was implemented as a new framework page, for the reason that if it was included in the DOM change violation test page, allowed URL lists for all these widgets had to be generated as well to prevent a high number of reported false positives. After generation of the allowed URL list, MAL was added to the page and the test was executed.

7.3.3 ATUSA Configuration

To perform the tests, most of the default configuration values of ATUSA were used. The significant configuration changes which were made are depicted by Figure 7.3. Note that because the test suite was designed with our approach in mind, there was no need to explicitly exclude any elements from the crawling process.

```
robot.events = onclick, onmouseover, onmousedown, onblur, onfocus,
               onkeydown, onkeypress, onkeyup, onmousemove,
               onmouseup, onselect
crawl.tags = a:{}, input:{}, img:{}, button:{}, label:{}, div:{}
crawl.tags.exclude = {}
atusa.plugins = nl.tudelft.swerl.login,
                # for cases SD1-SD5
                nl.tudelft.swerl.widgetInteractionDOMValidator
                # for case SH1
                nl.tudelft.swerl.httpRequestValidator
```

Figure 7.3: ATUSA configuration for the simplified widget framework case study.

7.3.4 Test Results

In order to get an indication of the overhead introduced by our approach, the widget pairs were first crawled by ATUSA without plugins. The number of candidate clickables (CC), clickables and states found during this run can also be used as a verification that plugins do not add states, for example because annotated attributes were not correctly removed. Table 7.1 shows the results of running ATUSA on the simplified framework without plugins. Table 7.2 shows the results of running ATUSA on the framework with plugins. The FP column depicts the number of reported false positives. For the widget pairs SD1-SD5 the DOM change violation detection plugin was enabled, for the SH1 widget pair the HTTP request violation detection plugin was enabled.

7.4 Case Study 2: Exact Widget Framework

As explained in Chapter 2.3, Exact Software is researching the possibilities of using widgets as building blocks for software solutions. A prototype of the framework was used to conduct a case study, to verify how our approach works on a framework with a more complex layout and functionality than our own framework.

Widget pair(s)	Time (ms)	CC	Clickables	States
SD1-SD5	753056	267	14	13
SH1	306167	108	5	6

Table 7.1: Results of running ATUSA on the simplified widget framework without plugins.

Widget pair(s)	Violations seeded	Violations detected	Real violations detected	FP	Time (ms)	CC	Clickables	States
SD1-SD5	5	5	5	0	768438	267	14	13
SH1	1	1	1	0	332024	108	5	6

Table 7.2: Results of running ATUSA on the simplified widget framework with plugins.

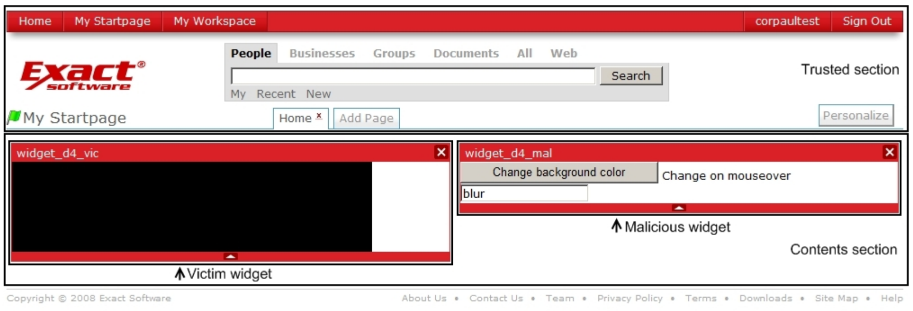


Figure 7.4: Screenshot of a DOM change violation widget pair in the Exact widget framework.

Figure 7.4 shows a screenshot of the EWF. Because dragging and dropping is allowed, testing isolated widget pairs, instead of the complete test suite, is more easily accomplished. It is very difficult to automatically drag widgets on the page. Therefore the widgets to test were manually placed on the page. This is possible, because the EWF remembers which widgets were on the page in the last session of a logged in user.

As in the first case study, all widget pairs contain two widgets, a malicious widget (MAL) and a victim widget (VIC). The widget pairs were designed by the author and implemented by a member of the Exact Research and Innovation team, to prevent bias of the test suite towards our approach. To test our approach with multiple widgets on a page, we have also conducted the test with all widget pairs on the same page.

7.4.1 DOM Change Violation Widget Pairs

The following DOM change violation widget pairs were implemented for the EWF:

D1: An event fired in MAL changes a form URL in VIC

D2: An event fired in MAL changes a link description in VIC

D3: An event fired in MAL changes a link href in VIC

D4: An event fired in MAL changes the background color of an element in VIC

D5: An event fired in MAL changes an image in VIC

D6: An event fired in MAL changes a DOM element outside MAL and outside VIC (a framework element)

Widget boundaries in the EWF have the `class="ex_widget"` attribute. Therefore the rule `class=ex_widget|VERY_HIGH` was added to the ATUSA widget boundary identification rules. Because our focus is on inter-widget interactions, all elements in the menu section were excluded from the crawling process. Other elements which were excluded, are elements which may remove or hide a widget.

7.4.2 HTTP Request Violation Widget Pairs

Because widgets can easily be dragged and dropped in the EWF, it is easier to test the HTTP request violation detection. This is because it is possible to isolate a widget on the page to generate its allowed URL list, when dragging and dropping is allowed. For this controlled experiment, only the allowed URL lists of the victim widgets were created. The following HTTP request violation widget pairs were implemented for the EWF:

H1: MAL attaches a key event handler to an input field in VIC

H2: MAL attaches a `focus` event handler to an input field in VIC

H3: MAL attaches an `onsubmit` event handler to a form in VIC

H4: MAL attaches an `onclick` event handler to a button in VIC

H5: MAL attaches a mouse movement event handler to an image in VIC

To run a widget pair, first an allowed URL list was generated for VIC by running it inside an isolated environment. After generation of the allowed URL list, MAL was added to the page and the test was executed.

7.4.3 ATUSA Configuration

To perform the tests, most of the default configuration values of ATUSA were used. The configuration changes which were made are depicted by Figure 7.5. Note that, in comparison with the first case study, `DIV` tags were not included in the crawling process. Because the EWF has a very complex page layout with many `DIV` elements, including them in the crawling process would add significantly to the crawling time. Because it is known in this controlled experiment that firing events on `DIV` elements would not expose violations, we decided to leave them out of the experiment (also see Section 8.1.4). The elements in the menu section were also explicitly excluded from the crawling process.

```
robot.events = onclick, onmouseover, onmousedown, onblur, onfocus,
               onkeydown, onkeypress, onkeyup, onmousemove,
               onmouseup, onselect
crawl.tags = a:{}, input:{}, img:{}, button:{}, label:{}
crawl.tags.exclude = a:{class=buttonedit}, a:{title=Close Layout},
                     a:{href=home}, a:{href=../home}, a:{href=myhome},
                     a:{href=person}, a:{href=myworkspace},
                     a:{id=ctl00_HeaderLoginView_SignOutButton},
                     a:{class=buttonclose}, a:{href=about}, a:{href=contact},
                     a:{href=team}, a:{href=privacy}, a:{href=tos},
                     a:{href=downloads}, a:{href=sitemap}, a:{href=help},
                     a:{class=buttonminimize}, a:{class=buttonmaximize,
                     input:{type=hidden}, input:{id=ctl00_SearchBoxinput},
                     input:{id=ctl00_SearchBoxbutton},
                     img:{id=ctl00_MainContent_titlebarViewer_image}
atusa.plugins = nl.tudelft.swerl.login,
                # for cases D1-D6 and D1-H5
                nl.tudelft.swerl.widgetInteractionDOMValidator
                # for cases H1-H5 and D1-H5
                nl.tudelft.swerl.httpRequestValidator
```

Figure 7.5: ATUSA configuration for the EWF case study.

Widget pair(s)	Time (ms)	CC	Clickables	States
D1	122349	124	3	4
D2	105215	120	3	4
D3	107299	120	3	4
D4	105625	120	3	4
D5	174616	124	3	4
D6	126446	152	4	5
D1-D6	552193	1557	19	20
H1	73538	116	3	4
H2	55071	84	2	3
H3	-	-	-	-
H4	51215	56	1	2
H5	61931	140	4	5
H1-H5	155608	550	10	11
D1-H5	702912	3207	29	30

Table 7.3: Results of running ATUSA on the EWF without plugins.

7.4.4 Test Results

Table 7.3 shows the results of running ATUSA on the EWF without plugins. Table 7.4 shows the results of running ATUSA with plugins. For widget pairs D1-D6 the DOM change violation detection plugin was enabled, for widget pairs H1-H5 the HTTP request violation detection plugin was enabled. During the execution of widget pairs D1-H5 both plugins were enabled.

Widget pair(s)	Violations seeded	Violations detected	Real violations detected	FP	Time (ms)	CC	Clickables	States
D1	3	3	3	0	110362	124	3	4
D2	3	3	3	0	90062	120	3	4
D3	3	3	3	0	92536	120	3	4
D4	3	3	3	0	89962	120	3	4
D5	3	3	3	0	158853	124	3	4
D6	4	4	4	0	109711	152	4	5
D1-D6	19	19	19	0	537829	1557	19	20
H1	1	2	1	1	74740	116	3	4
H2	1	2	1	1	54621	84	2	3
H3	-	-	-	-	-	-	-	-
H4	1	2	1	1	51365	56	1	2
H5	1	2	1	1	60950	140	4	5
H1-H5	4	8	4	4	157582	550	10	11
D1-H5	23	43	23+10	10	759685	3207	29	30

Table 7.4: Results of running ATUSA on the EWF with plugins.

7.5 Case Study 3: Pageflakes

The third case study we have performed was on Pageflakes. Pageflakes is a free service, which allows to place widgets on a personalized home page. Pageflakes is a closed source framework, which means we were not able to inspect or adapt the framework code. In Pageflakes, widgets are called flakes. At the time of writing over 240,000 flakes were available. Figure 7.6 shows a screenshot of a personalized homepage in the Pageflakes framework. A flake boundary can be detected using the same approach as used for widget boundary detection.

For our case study we have implemented the same widget pairs as described in Sections 7.4.1 and 7.4.2 using the Pageflakes API. This API provides convenience JavaScript methods for event management and DOM manipulation. The tests were run using the same setup as for the other case studies. Flakes were manually placed on the page, after which they were tested with ATUSA.

Flake boundaries are `DIV` containers with the `class="flake"` attribute. Therefore the rule `class=flake|VERY_HIGH` was added to the ATUSA widget boundary identification rules. All elements in the menu section were excluded from the crawling process. Other elements which were excluded, are elements which may remove or hide a widget.

7.5.1 ATUSA Configuration

The same configuration settings as in the EWF case study were used to conduct the case study on Pageflakes. The only change which was made was in the excluded elements. Figure 7.7 depicts the configuration of ATUSA for this case study. All tests were allowed a maximum crawling time of 1 hour.

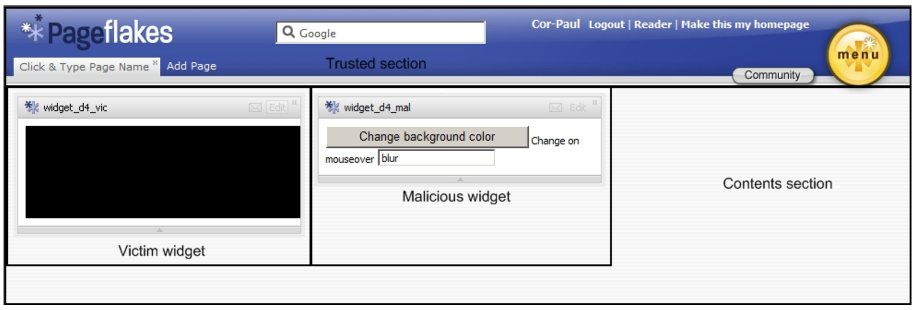


Figure 7.6: Screenshot of a DOM change violation widget pair in the Pageflakes framework.

```
robot.events = onclick, onmouseover, onmousedown, onblur, onfocus,
               onkeydown, onkeypress, onkeyup, onmousemove,
               onmouseup, onselect
crawl.tags = a:{}, input:{}, img:{}, button:{}, label:{}
crawl.tags.exclude = a:{href=%www%pageflakes%com%}, a:{id=confirm%},
                     a:{href=%Logout%}, a:{id=RssReaderLink},
                     a:{id=homeProfileLink}, a:{id=SetAsHomePageLink},
                     a:{id=CommunityButton}, a:{id=Start},
                     a:{id=NewTabLink}, a:{class=delete_page},
                     a:{href=%Page%aspx}, a:{href=%Flakes%aspx},
                     a:{href=%ProfileDirectory%aspx}, a:{class=%icon},
                     a:{href=%forums%pageflakes%com%},
                     a:{href=%company%pageflakes%com%tos},
                     a:{href=%company%pageflakes%com%privacy},
                     a:{href=%developers%pageflakes%com%}, a:{class=flake_title},
                     a:{class=confirm%}, a:{class=flake_name%}, a:{class=%tab},
                     a:{class=rss_number}, img:{id=icon%}, img:{class=logo},
                     input:{value=Close}, input:{type=hidden},
                     input:{id=domainSearchBox}, label:{id=confirmDialog%},
                     input:{id=confirm%}
atusa.plugins = nl.tudelft.swerl.login,
                # for cases D1-D6 and D1-H5
                nl.tudelft.swerl.widgetInteractionDOMValidator
                # for cases H1-H5 and D1-H5
                nl.tudelft.swerl.httpRequestValidator
```

Figure 7.7: ATUSA configuration for the Pageflakes case study.

7.5.2 Test Results

Table 7.5 shows the results of running ATUSA on the Pageflakes framework without plugins. Table 7.6 shows the results of running ATUSA with plugins. For widget pairs D1-D6 the DOM change violation detection plugin was enabled, for widget pairs H1-H5 the HTTP request violation detection plugin was enabled. During the execution of widget pairs D1-H5 both plugins were enabled. The scenario with all widgets on the page (D1-H5) required more crawling time than the maximum time allowed. Therefore the tests were stopped after one hour and the output until then was analyzed. For tests which required more crawling time, we have marked the candidate clickables, clickables and states with * in Tables 7.5

Widget pair(s)	Time (ms)	CC	Clickables	States
D1	-	-	-	-
D2	263391	1500	29	30
D3	107299	1500	29	30
D4	289375	1500	29	30
D5	358141	1530	29	30
D6	289375	1330	30	31
D1-D6	1675843	16797	149	150
H1	160938	833	16	17
H2	178641	1029	20	21
H3	-	-	-	-
H4	176984	1029	20	21
H5	175328	1029	20	21
H1-H5	907609	7857	80	81
D1-H5	3600000*	*	*	*

Table 7.5: Results of running ATUSA on the Pageflakes framework without plugins.

Widget pair(s)	Violations seeded	Violations detected	Real violations detected	FP	Time (ms)	CC	Clickables	States
D1	-	-	-	-	-	-	-	-
D2	3	3	3	0	229594	1500	29	30
D3	3	3	3	0	222703	1500	29	30
D4	3	3	3	0	387344	1500	29	30
D5	3	3	3	0	368156	1530	29	30
D6	3	3	3	0	279422	1330	30	31
D1-D6	15	15	15	0	2030719	16797	149	150
H1	1	1	1	0	158062	833	16	17
H2	1	1	1	0	177875	1029	20	21
H3	-	-	-	-	-	-	-	-
H4	1	1	1	0	177000	1029	20	21
H5	1	1	1	0	179406	1029	20	21
H1-H5	4	4	4	0	919687	7857	80	81
D1-H5	19	19	19	0	3600000*	*	*	*

Table 7.6: Results of running ATUSA on the Pageflakes framework with plugins.

and 7.6.

7.6 Evaluation Results

The evaluation results of the case study on the simplified widget framework show excellent results. All seeded violations were detected without false positives.

More interesting results were exhibited by the case studies on the EWF and Pageflakes. After the first run on the EWF a large number of false positives was reported. After inspection, we found that the jQuery UI[4] library, used in the EWF for dragging and dropping, attaches `onmousedown` event handlers to many elements on the page. These handlers cause DOM changes for each `onmousedown` event fired on these elements. To avoid these false

[4]jQuery UI: `http://jqueryui.com/`

positives, ATUSA was adapted to ignore DOM changes of this type. Tables 7.3 and 7.4 show the results of running ATUSA with these adaptions.

During the second run, a number of interesting results were found. The first is that widget pair H3 could not be validated. The widget pair did not contain valid HTML, which caused a problem after the JavaScript injection. During this process, the HTML is parsed and serialized. The serialization library converts all input to valid HTML, which breaks the functionality of the framework for invalid HTML. Therefore widget pair H3 was not included in the test results. Because the Pageflakes framework does not allow the use of `<form>` elements, widget pair D1 was not implemented for that case study.

Another interesting result concerns 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. This is because crawling involves communication with a server, which may be slow or overloaded. Therefore crawling times should be seen as an indication only. Note that this result also indicates that our implementation does not introduce much overhead.

The results of widget pair H1-H5 show a false positive for each widget pair in the EWF case study. All these false positives were caused by an implementation characteristic of the EWF and do not represent real false positives. In the EWF, JavaScript for each widget is loaded in the `body` section rather than inside the widget boundary. Therefore, although the JavaScript request will appear in a widget's allowed URL list, the request is coupled to the framework rather than to the widget. This is correct with regard to our approach, but incorrect with regard to the semantics of the framework. The same type of false positives are being recognized in the test with all widget pairs (D1-H5). The Pageflakes framework adds JavaScript of a widget inside the widget boundary. Therefore we did not detect such false positives during the Pageflakes case study. It was interesting to see that more violations were detected by our plugins than we deliberately seeded in the D1-H5 test in the EWF case study. This was because widget pairs H1-H5 displayed a message after an HTTP request was succesfully sent in the EWF case study. This caused the DOM change violation detection to detect a violation.

Revisiting the questions stated in the first section of this chapter, we can conclude our approach is succesful in detecting inter-widget interaction violations. Our approach is able to automatically detect DOM change violations and HTTP request violations, without detecting a high number of false positives. However, it is important to keep framework implementation decisions, such as usage of the jQuery UI library, in mind while analyzing the results.

All widget pairs were completed within reasonable time. This is also true for tests with more than twenty widgets on the page. However, it is important to consider that not all types of elements were included in the crawling process. Although including all elements in the crawling process may add considerably to the crawling time, this should not cause performance problems in a dedicated testing environment.

We have tested our approach with more than twenty widgets at the same time, which did not cause any problems. An observation is that combining widget pairs is faster than running each widget pair on its own. This is because framework elements which were not explicitly excluded have to be crawled only once, rather than for each widget pair. The test results show that our approach does not introduce much overhead compared to crawling without validation. Although this is promising for the scalability of our approach, more testing should be done to be able to draw a definitive conclusion about the scalability.

Chapter 8

Discussion

In Chapter 3 we have introduced a number of security issues in AJAX applications, widgets in particular. We have proposed and implemented an approach for automatically detecting two of such issues, DOM change violations and HTTP request violations, with ATUSA. In this chapter we discuss issues regarding our approach and its implementation.

8.1 Completeness

A security testing approach which is not complete may have limited applicability, as it may miss many security violations while testing. In this section completeness issues of our approach are discussed.

8.1.1 Crawler Usage

The first fact to consider is that our approach uses a crawler. Therefore, if ATUSA is not able to find a certain state, our approach is not able to test that state. This problem is inherent to dynamic approaches which use a crawler.

8.1.2 Valid (X)HTML

Our current implementation is capable of testing widgets which contain valid (X)HTML only. This is caused by the HTML serializer used during the JavaScript injection process, as explained in Section 7.6. Future research should include research on the HTML serialization mechanism to use.

8.1.3 Timers

The usage of timers in JavaScript may form a problem for our approach. 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. Timers may cause this because they can delay the effect of an event until the event is fired. Future work should contain more research on the effect of timers on our approach.

8.1.4 Item Exclusion

In our empirical evaluation we have excluded a number of elements from the crawling process. In addition, we have used a list of trusted items. In a controlled experiment this causes no problem, because it is known that those elements are not altered. However, in a production testing environment, it is necessary to check those elements and their effects, to ensure none of them have been altered by a malicious widget. An example of such a violation is the attachment of a key logger to the search field in the trusted section in Figure 7.4.

8.1.5 Discovery of Widget Pairs

In our test cases all widget pairs were known. In an industrial framework, many widgets may be available and placing only widgets on the page, that will be violated by the widget under test, is a difficult task. One possibility would be to place all widgets in the catalog on the page, but this may lead to performance problems.

8.1.6 Cross-site Request Forgery

In our approach for detecting HTTP request violations we use an allowed URL list for every widget. Using such a list means that another widget can still attach an event handler, which sends a request to a URL in the allowed URL list, to a widget. An example is depicted by Figure 8.1. Because `http://www.efw.com/post.php` is in the allowed URL list of `VIC`, the request triggered by the event attached by `MAL` is not detected as an HTTP request violation. This may lead to a similar vulnerability as the type of vulnerability known as cross-site request forgery (see Appendix D.3). This is a problem that should be handled by the widget developer rather than by our approach.

8.1.7 Multiple Browser Support

It is important for an automated testing approach to support multiple browsers, because of the browser incompatibilities discussed in Section 2.1.1. The current implementation of our approach supports Internet Explorer 7 only. Future plans are an implementation for Firefox, although there is one thing to consider regarding our approach. The implementation of our approach relies on the use of `window.event`, which holds the last event fired. This is the IE-specific way of event handling. Firefox uses a different way of event handling[1], which requires each event to be explicitly propagated as a function parameter in order to use it in a function. Although this does not affect the applicability of our approach, it requires changes in implementation.

[1]Firefox event handling: `https://developer.mozilla.org/En/DOM:element`

```
<div id="VIC" class="widget">
  <form id="vicForm" onsubmit="send('http://www.efw.com/post.php',␣this)">
  <input type="text" name="code">
  </form>
</div>

<div id="MAL" class="widget">
  <script type="text/javascript">
  var url = "http://www.efw.com/post.php";
  var form = getMaliciousFormContents();
  document.getElement("VIC.vicForm").onsubmit = send(url, form);
  </script>
</div>

AllowedList vic = {http://www.efw.com/post.php}
```

Figure 8.1: Example of an HTTP request violation not detected by our approach.

8.1.8 CSS Dispositioning

CSS dispositioning is a form of hacking in which elements are dispositioned in such a way that they appear in front of other elements. An example is a widget which creates a login form, and positions it using CSS on top of a form in another widget. Although the visible form appears exactly the same to the user, it is in fact the form created by the malicious widget. Our approach cannot detect such behaviour, as the DOM of the other widget does not change. A possible solution for this would be to extend the DOM change violation detection algorithm with a collision detection algorithm, to detect overlapping elements on the page. The detected elements can be used as input for the widget boundary detection algorithm, after which the boundaries can be used to validate the collision.

8.2 Security

In addition to completeness of a security approach, it is important that it cannot easily be bypassed. The first observation is that widgets execute mostly on the client-side of an application using JavaScript. As explained in Chapter 4, testing the client-side of AJAX applications is a difficult and new research area. 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. Because this requires a DOM change in the other widget, this action should be recognized by our DOM change violation detection algorithm.

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 for this could be to implement a verification mechanism in ATUSA, which verifies that a node is annotated before and after an event is fired. This could stop a malicious widget from removing the `requestforproxyid` attribute. The malicious widget may try to place the unique value of `requestforproxyid` in another element. In this case the XPath query which is used to locate the element based on the value in the HTTP request will return more than one element, which is a violation by default.

Another possible security issue could be that JavaScript on the page subverts AJAX again. An example of this is a subversion, in which the `requestforproxyid` of another element than the active one is added to the HTTP request. This is a difficult problem and should be addressed in further research.

8.3 Scalability

In Section 7.6 we have shown our approach works when more than twenty widgets are on the same page. In an industrial framework, many more widgets may require testing. Our approach requires all widgets 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, or on the number of users who have the widget on their personal homepage.

Another aspect of scalability is the number of testers which can use our approach at the same time. Because ATUSA and our plugins require client-side installation only, our approach does not limit the number of concurrent users. Therefore this number is limited by the number of allowed concurrent users in the framework, or by the web server. In addition, it is important to consider that two testers may share pages under test, which can lead to problems. If every tester uses his own framework account, and places the widgets under test on his own personal page, there are no scalability issues regarding the number of concurrent testers imposed by our approach.

8.4 Performance

In Chapter 7 we have measured the performance of our approach. Although the measured crawling times are indications only because of server delay, the results are promising. A problem inherent to crawler-based approaches is that they use runtime information, and therefore have to deal with slow servers. In addition, ATUSA can crawl content which can only be made visible through asynchronous calls. In order to do this, ATUSA waits a short period of time after firing an event, to see if the event has had an effect. During a crawling session many events are fired, which means ATUSA is idle during a considerable portion of the crawling time. This negatively affects the performance of our approach. In Section 7.6 we have shown that the overhead introduced by our approach is small, which indicates our performance can be closely tied to the performance of ATUSA.

8.5 Threats to Validity

To ensure the internal validity of our experiment we have tested our plugins with a JUnit[2] test suite. In addition, we have used the simplified widget framework described in Section 7.3 to validate the behaviour and functionality of our plugins.

The requirements of the widget pairs used in the case studies were designed based upon known exploits in web application security research, for example by analyzing the effects of cross-site scripting or phishing attacks. For example, 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, 'Changing the action URL of a form'.

To improve the external validity of our case studies, we have tested equal scenarios on different frameworks, including an industrial, widely used framework (Pageflakes).

A threat to the validity of our experiment is the complexity of the widget pairs implemented for the case studies. Although the pairs exhibit realistic behaviour, the event sequences required to reach a violated state are short. A future case study should be done either with more complex or real, published widgets.

A final validity threat is that we have explicitly excluded elements from the crawling process. This may have led to considerably shorter crawling times. In addition, excluding elements may cause false positives to be hidden from the reports generated by the plugins.

8.6 Different Applications

Although our approach is described with a bias towards AJAX widgets, its applicability is not limited to it. In fact, inter-element interactions between any type of elements can be tested by changing the configuration settings for widget boundary detection. In addition, our approach is not limited to AJAX applications, but can be used to test the security of any web application in which inter-element interactions are an issue.

In addition to the security testing field, our approach has proved to be useful in other areas. Many developers who start working the jQuery library make mistakes, and because of its powerfulness, these mistakes may affect the whole page instead of one element. Because our DOM change violation detection approach can detect such mistakes, our plugin can help developers ensuring their jQuery actions only affect the desired widget.

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 the detection of links to external domains. Our approach, when slightly altered, 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 all content. ATUSA can verify the validity of links and AJAX calls by analyzing the HTTP response code for each request on the proxy.

[2]JUnit: `http://www.junit.org/`

Chapter 9

Related Work

In this chapter an overview of research related to our work will be given. First the difference between static and dynamic approaches will be explained. After this an overview of research on traditional web application security and security in AJAX applications will be given. Finally SOP-related research will be discussed.

9.1 Static versus Dynamic Analysis

A classification of security tools can be made based upon the analysis approach they use. A security tool can use dynamic analysis, static analysis or a combination of those. Static analyzers parse and analyze the source code of an application [15]. Because this process may take a long time, a model of the application is often used instead. A static analysis method which is often used is type checking. Because static analysis tools cannot understand the semantics of source code they often report a high number of false positives [30].

Dynamic analyzers execute the application and observe its runtime behaviour [13]. The advantage of this is that they are suitable for more applications because they have no requirement on the implementation language. Another advantage of dynamic analyzers is that they create execution traces of what went wrong. A disadvantage of using dynamic analysis for web applications is that a dynamic analyzer must be able to expose all content on a web page [28].

Static analyzers are often integrated early, while dynamic analyzers are mostly used later in the development cycle, simply because it may be difficult to retrieve runtime information from incomplete applications. A security tool may also use both analysis methods to achieve a higher level of security testing [15].

9.2 Traditional Web Application Security

Over the years many research on the security of web applications has been done. Every year the Open Web Application Security Project (OWASP)[1] releases a top ten of the types of vulnerabilities which occurred most in the wild [32]. The most occurring vulnerabilities

[1]OWASP: `http://www.owasp.org`

by far are those which are caused by weak input validation, such as cross-site scripting (XSS) and SQL injection [32, 40]. These vulnerabilities are explained in Appendix D.

A key difference between our approach and existing security testing methods is that these are often not applicable to AJAX web sites. Existing methods use crawlers which are not capable of keeping track of the DOM during the crawling process. Therefore they usually cannot be used to test AJAX applications. However, these methods contain valuable ideas and processes, which is why they are discussed here.

9.2.1 Black-box Security Scanners

An important group of security tools are black-box security scanners. Black-box security scanners dynamically access the web application using public interfaces only, like a malicious user would [14]. They are used after the development and before the deployment cycle [4]. The reason for this is that they try to detect vulnerabilities by entering malicious input and analyzing the response of the application, a process known as fault injection [13, 14, 18]. This means they require a fully functional application and therefore using them before this stage would be less effective.

A black-box security scanner is SecuBat [18]. SecuBat detects SQL injection and XSS. It does this by extracting all forms from a web page. These forms are injected with faults and submitted. To detect SQL injection, the response page is analyzed and checked for the existence of SQL error indications such as 'sqlexception' or 'error'. SecuBat's crawler is not capable of sending JavaScript events. SecuBat has a plugin model which allows for extension of the application.

Huang et al [13, 14] propose another black-box security scanner called WAVES. WAVES is an open source automated security scanner for web applications, which aims at detecting SQL injection and XSS. It contains a crawler which is capable of crawling a web site by following links, filling out forms and executing JavaScript events. Filling out forms is an important aspect of a crawler as forms hide a large part of the content on the web [14]. WAVES uses the topic model to fill out forms. The goal of the topic model is to provide semantically and syntactically correct values, to make acceptance of input more likely. This is done by maintaining a list of input field names and correct values, for example {`'company', 'compName':'IBM', 'Microsoft', 'Exact'`}. When an input field is found in a form, a value which is likely to be correct for it will be selected from the list.

WAVES and SecuBat are both tools for the detection of SQL injection and XSS. Because they do not keep track of the DOM it is difficult to validate the security of inter-widget interactions with them. These interactions exist at the client-side only, which results in the fact that a black-box security scanner should be able to inspect the client-side. Our approach fulfills this requirement and therefore does not have the problem existing scanners have.

9.2.2 Proxy Approaches

Another group of dynamic security approaches are those which use a proxy. A proxy is placed between the client and server, with the goal of getting more information about what is being sent to the server. Other purposes of using a proxy are injecting data into a request or response, or validating HTTP requests.

Noxes [20] demonstrates the use of a proxy. Noxes is a client-side firewall, which has access rules for browser connections instead of applications. Every page has a list of allowed domains it may access. Because AJAX applications follow the single-page model, only one list can be used for each application. Our approach allows for the specification of such lists on widget level rather than application level, which makes our approach better applicable for AJAX applications, especially widget frameworks. In addition, our approach is a detection approach, whereas Noxes is a prevention approach.

BrowserShield [34] is a tool for protecting users against known vulnerabilities during the time between the publication of the vulnerability and the release of the security patch. BrowserShield protects against browser vulnerabilities rather than against application-level vulnerabilities. It uses a proxy to inject and rewrite JavaScript which is sent to the client.

Another approach which uses a proxy is proposed by Scott et al. [36]. Their approach allows for the definition of security policies to validate or transform the parameters of HTTP requests. The requests are validated, for example using type checking, or transformed, for example using strong input validation, on the proxy. This approach differs from ours in the way that security policies must be defined manually for each parameter, while our approach does not require the definition of such policies, but uses a generic detection mechanism instead.

A final approach which uses a proxy is UsaProxy [2], which is not a security approach but an approach for tracking the usability of AJAX applications. UsaProxy injects JavaScript to each page which tracks all user actions on the page. At certain intervals the data is sent to the server, where it is logged. UsaProxy does not couple HTTP requests and the origin element, which our approach does.

9.2.3 Static Approaches

Static analysis on applications which are a hybrid of client and server-side is difficult. Existing approaches focus on either the client or the server-side. Most static approaches for detecting XSS and SQL injection use information flow analysis [15]. In this type of analysis all paths from entry points to output points are extracted and verified to check whether input reaches an output point in its original, non-validated, form. If this is the case, a vulnerability was found.

An example of such a server-side static approach for PHP is Pixy [17]. Pixy is an open source tool for statically detecting XSS vulnerabilities. Because PHP is a server-side language, Pixy is only applicable to the server-side of AJAX applications.

WebSSARI [15] is another static analysis tool for PHP. Huang et al. [15] realize that static analysis often does not give a satisfactory approximation of runtime behaviour. Therefore they propose a hybrid approach, which uses both static and dynamic analysis. Using static analysis they point out code requiring runtime checks, and automatically insert those checks which allow for runtime protection. Because our approach has access to the DOM at all times during crawling, we do not have the need for insertion of runtime checks using static analysis, as we can analyze the application at all times during runtime.

Another static approach is proposed by Wasserman et al. [43]. The authors propose a method for analyzing the possible output of a validation function and verify whether this possible output would allow a user to access the browser's JavaScript interpreter.

Static analysis of AJAX is difficult because of the interaction between client and server-side in AJAX applications. To our knowledge no static analysis method for AJAX exists yet.

9.3 AJAX and Widget Security

Since AJAX is a relatively new technology, not much research has been done on it yet. The community has only recently began to realize AJAX imposes new issues on web application security. Attempts have been made to give an overview of security risks introduced or enhanced by AJAX [10, 39], but only few approaches known to the authors exist, which actively try to prevent or detect these specific problems.

This is even more the case for widget and mashup security. As explained in Section 3.2, most frameworks either disallow all communication between widgets, or allow all nonsecure communication. Google has made an attempt to allow widgets to communicate using the pubsub protocol [9]. This protocol allows widgets to subscribe to communication channels. Publisher widgets may communicate with subscriber widgets by sending messages over such a channel. However, using this protocol is not allowed for user-published widgets.

9.3.1 Sandbox Approaches

The approach taken most in current research is the sandbox approach. By placing the widget in a sandbox, functionality is being limited. Examples of such approaches are SubSpace [16] and MashupOS [42], which both introduce new HTML elements which give a widget more freedom than an `IFrame`, and less than a `DIV`. Our approach places a virtual sandbox on the widget by disallowing communication outside its boundary, without the requirement for changes to the HTML standard or browsers.

An approach which limits functionality by sandboxing JavaScript objects is Google Caja [7]. Google Caja is a subset of JavaScript, in which all objects can communicate with other objects of which they have a reference only. A JavaScript file is converted to a Caja module, which contains only secure JavaScript. Because this approach rewrites JavaScript, it is important that the JavaScript is in a standard format. Because our approach does not use the JavaScript source code, it does not have this requirement. In addition, our approach is designed for detection rather than prevention.

Chapter 10

Conclusion

AJAX has led to a new type of web application component, called web widget. A widget is a mini-application, which can be placed on a homepage next to other widgets using a widget framework. A consequence of this is that widgets may access and change each other's properties. From a security point of view, this is often not allowed, and therefore a violation. In the beginning of this report, we stated the following research questions:

RQ1: What are the problems and difficulties of detecting these violations?

RQ2: Can we propose an automated approach for detecting these violations?

We have identified two types of inter-widget interaction violations, and we have proposed an automated approach for detecting them. The first type of violation is the DOM change violation, in which a widget changes the DOM of another widget. The second type of violation is the HTTP request violation, in which a widget makes an HTTP request it is not allowed to.

The implementation of our approach uses ATUSA, a tool for automated testing of AJAX applications. For this purpose we have implemented several ATUSA plugins and we have integrated an HTTP proxy into ATUSA. We have evaluated this implementation using two case studies. The first case study was on a simplified widget framework. The second case study was on the EWF, a widget framework researched by the Research and Innovation team of Exact Software. From the evaluation results we can conclude our approach is successful in detecting DOM change violations and HTTP request violations, with a small number of false positives and within reasonable time.

10.1 Contributions

In this report, we have contributed the following:

C1: The first dynamic approach for automatically testing AJAX-based web widgets,

C2: The definition of a widget boundary in the DOM and a method for deciding to which widget a DOM element belongs,

C3: A method for coupling an HTTP request to the HTML element from which the initiating event was triggered,

C4: A case study of extension of ATUSA using plugins, and proof that it can be used for automated security testing,

C5: A method for automatically detecting DOM change violations and HTTP request violations.

10.2 Future Work

Our approach can be improved on a number of aspects. The first is completeness, of which a number of examples are given in Section 8.1. An important future task is adapting ATUSA such that it is able to automatically drag and drop widgets onto the personal homepage. This would greatly benefit the testing process and would add to the completeness of our approach. More research should be done on CSS dispositioning and its threats, as this is a type of vulnerability which has not been widely researched yet.

Future work also includes research on defining the best subsets of widgets for testing, as it is not possible to place all available widgets on the same page in larger widget frameworks.

Finally, more research should be done on threats to the validity of our approach. Any client-side based security approach is difficult to secure, which makes it interesting to see if an external tool, such as ATUSA, can contribute to this process. Therefore more case studies should be performed.

Bibliography

[1] The AspectJ Team. *The AspectJ Programming Guide*. Palo Alto Research Center, 2003. Version 1.2.

[2] Richard Atterer and Albrecht Schmidt. Tracking the Interaction of Users with AJAX Applications for Usability Testing. In *CHI '07: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 1347–1350, New York, NY, USA, 2007. ACM.

[3] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[4] Mark Curphey and Rudolph Araujo. Web Application Security Assessment Tools. *IEEE Security and Privacy*, 4(4):32–41, 2006.

[5] Robert J. Ennals and Minos N. Garofalakis. MashMaker: Mashups for the Masses. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 1116–1118, New York, NY, USA, 2007. ACM.

[6] Jesse James Garrett. AJAX: A New Approach to Web Applications. `http://www.adaptivepath.com/ideas/essays/archives/000385.php`, 2005.

[7] Google. Google Caja. `http://code.google.com/p/google-caja/`.

[8] Google. Google Mashup Editor Getting Started Guide. `http://code.google.com/gme/docs/gettingstarted.html`, 2008.

[9] Google. Gadget-to-Gadget Communication. `http://code.google.com/apis/gadgets/docs/pubsub.html`, 2008.

[10] Billy Hoffman and Bryan Sullivan. *AJAX Security*. Pearson Education, 2008.

[11] Gregor Hohpe. Google Mashup Editor and Yahoo! Pipes: Friend, not Foe. `http://code.google.com/support/bin/answer.py?answer=72765&topic=12044`, 2007.

[12] Michael Howard and David E. Leblanc. *Writing Secure Code*. Microsoft Press, Redmond, WA, USA, 2002.

[13] Yao-Wen Huang, Shih-Kun Huang, Tsung-Po Lin, and Chung-Hung Tsai. Web Application Security Assessment by Fault Injection and Behavior Monitoring. In *WWW '03: Proceedings of the 12th international conference on World Wide Web*, pages 148–159, New York, NY, USA, 2003. ACM.

[14] Yao-Wen Huang, Chung-Hung Tsai, D. T. Lee, and Sy-Yen Kuo. Non-Detrimental Web Application Security Scanning. In *ISSRE '04: Proceedings of the 15th International Symposium on Software Reliability Engineering*, pages 219–230, Washington, DC, USA, 2004. IEEE Computer Society.

[15] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Securing Web Application Code by Static Analysis and Runtime Protection. In *WWW '04: Proceedings of the 13th international conference on World Wide Web*, pages 40–52, New York, NY, USA, 2004. ACM.

[16] Collin Jackson and Helen 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.

[17] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (short paper). In *SP '06: Proceedings of the 2006 IEEE Symposium on Security and Privacy*, pages 258–263, Washington, DC, USA, 2006. IEEE Computer Society.

[18] Stefan Kals, Engin Kirda, Christopher Kruegel, and Nenad Jovanovic. SecuBat: A Web Vulnerability Scanner. In *WWW '06: Proceedings of the 15th international conference on World Wide Web*, pages 247–256, New York, NY, USA, 2006. ACM.

[19] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *11th Europeen Conf. Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242. Springer Verlag, 1997.

[20] Engin Kirda, Christopher Kruegel, Giovanni Vigna, and Nenad Jovanovic. Noxes: A Client-side Solution for Mitigating Cross-site Scripting Attacks. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 330–337, New York, NY, USA, 2006. ACM.

[21] Peter-Paul Koch. JavaScript - Event Compatibility Tables. `http://www.quirksmode.org/dom/events/index.html`, 2008.

[22] Peter-Paul Koch. W3C DOM Compatibility Tables. `http://www.quirksmode.org/dom/compatibility.html`, 2008.

[23] Jean-Claude Laprie and Brian Randell. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Trans. Dependable Secur. Comput.*, 1(1):11–33, 2004. Fellow-Algirdas Avizienis and Senior Member-Carl Landwehr.

[24] Duane Merrill. Mashups: The New Breed of Web App. http://www.ibm.com/developerworks/library/x-mashups.html, 2006.

[25] Ali Mesbah, Engin Bozdag, and Arie van Deursen. Crawling AJAX by Inferring User Interface State Changes. In D. Schwabe and F. Curbera, editors, *Proceedings of the 8th International Conference on Web Engineering (ICWE'08)*. IEEE Computer Society, July 2008.

[26] Ali Mesbah and Arie van Deursen. An Architectural Style for AJAX. In D. Paulish, I. Gorton, J. Tyree, and D. Soni, editors, *Proceedings of the 6th Working IEEE/IFIP Conference on Software Architecture (WICSA'07)*, pages 44–53. IEEE Computer Society, 2007.

[27] Ali Mesbah and Arie van Deursen. Migrating Multi-page Web Applications to Single-page AJAX Interfaces. In R. L. Krikhaar, C. Verhoef, and G. A. Di Lucca, editors, *Proceedings of the 11th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 181–190. IEEE Computer Society, March 2007.

[28] Ali Mesbah and Arie van Deursen. Exposing the Hidden Web Induced by AJAX. Technical Report SERG-2008-001, Delft University of Technology, The Netherlands, 2008.

[29] Ali Mesbah and Arie van Deursen. Invariant-Based Automatic Testing of AJAX User Interfaces. In *Proceedings of the 31st International Conference on Software Engineering (ICSE'09)*. IEEE Computer Society, 2009.

[30] Matteo Meucci. *OWASP Testing Guide v2*. OWASP Foundation, 2007.

[31] Microsoft. Microsoft Popfly Documentation. http://www.popflywiki.com/, 2008.

[32] OWASP. OWASP Top Ten Project. http://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project, 2007.

[33] Stefano Di Paola and Giorgio Fedon. Subverting AJAX. In *23rd CCC Conference*, 2006.

[34] Charles Reis, John Dunagan, Helen J. Wang, Opher Dubrovsky, and Saher Esmeir. BrowserShield: Vulnerability-driven Filtering of Dynamic HTML. *ACM Trans. Web*, 1(3):11, 2007.

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

[36] David Scott and Richard Sharp. Abstracting Application-level Web Security. In *WWW '02: Proceedings of the 11th international conference on World Wide Web*, pages 396–407, New York, NY, USA, 2002. ACM.

[37] Chris Shiflett. Security Corner: Cross-Site Request Forgeries. http://shiflett.org/articles/cross-site-request-forgeries, 2004.

[38] Clinton W. Smullen and Stephanie A. Smullen. An Experimental Study of AJAX Application Performance. *JOURNAL OF SOFTWARE*, 3:30–37, 2008.

[39] Michael Sonntag. AJAX Security in Groupware. In *EUROMICRO '06: Proceedings of the 32nd EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 472–479, Washington, DC, USA, 2006. IEEE Computer Society.

[40] Andrew van der Stock and Adrian Wiesmann, editors. *A Guide to Building Secure Web Applications and Web Services*. OWASP Foundation, 2006.

[41] W3C. Frames in HTML Documents. `http://www.w3.org/TR/html4/present/frames.html`.

[42] Helen J. Wang, Xiaofeng Fan, Jon Howell, and Collin Jackson. Protection and Communication Abstractions for Web Browsers in MashupOS. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 1–16, New York, NY, USA, 2007. ACM.

[43] Gary Wassermann and Zhendong Su. Static Detection of Cross-site Scripting Vulnerabilities. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 171–180, New York, NY, USA, 2008. ACM.

[44] Yahoo! Yahoo! Pipes Documentation. `http://pipes.yahoo.com/pipes/docs`, 2008.

Appendix A

Glossary

In this appendix we give an overview of frequently used terms and abbreviations.

AJAX: Asynchronous JavaScript and XML - a technique used to make asynchronous requests in web applications.

AOP: Aspect-Oriented Programming - a method of programming which allows for the definition of cross-cutting concerns.

API: Application Programming Interface - an interface for accessing a service or data source.

ATUSA: Automated test framework for AJAX applications. Contains CRAWLJAX and validation plugins.

CSRF: Cross-Site Request Forgery - an attack in which unauthorized commands are sent by a web page the user trusts.

CSS: Cascading Style Sheets - a method for specifying the layout of a web page.

DOM: Document Object Model - the representation of a web page as seen by JavaScript.

EWF: Exact Widget Framework - the framework researched by Exact which allows users to manage widgets.

GME: Google Mashup Editor - tool for creating mashups.

GUI: Graphical User Interface - the graphical user interface of an application.

HTML: HyperText Markup Language - the language which is used to create web pages.

JPF: Java Plugin Framework - plugin framework used by ATUSA.

JSON: JavaScript Object Notation - a data interchange format.

SOP: Same Origin Policy - a policy implemented by browsers that prohibits two documents from different origins from communicating with each other.

SPIAR: Single Page Interface Architecture - a software architecture for AJAX applications.

Traditional web application: A web application which does not use AJAX.

UI: User Interface - the user interface of an application.

URL: Uniform Resource Locator - a method for uniquely identifying a web page.

UWA: Universal Widget API - a universal API for describing widgets.

(Web) Widget: A mini-application which runs in an environment provided by a widget framework.

XML: eXtensible Markup Language - a data interchange format.

XSS: Cross-Site Scripting - an attack in which unauthorized code is injected in a web page.

Appendix B

ATUSA Adjustments Overview

In this chapter the changes which were made to ATUSA are explained. This chapter discusses only the changes which were made to the core source code of ATUSA. The goal of the approach used for implementing these changes was to offer plugins the possibility to extend ATUSA while restraining as much as possible from changing the source code. The implementation of the plugins is discussed in Chapters 5 and 6 of this report.

B.1 Active Element

As defined in Section 5.1.2, the active element is the element on which the last event is fired by ATUSA. This element must be accessed during the validation phase because our approaches rely on it. The active element was made available by storing it in the `IEBrowser` object, right before an event is fired. Events can be fired by the methods `input()` and `fireEventWait()`.

B.2 Accessing the Internal Browser DOM

As explained in Section 5.2.1, CRAWLJAX has access to three DOM objects. The DOM of the internal browser must be accessible to annotate data on its elements. CRAWLJAX uses Watij[1], an open source library for web application testing in Java, to access the internal browser. Unfortunately Watij does not allow direct access to the internal browser's DOM. However, this can be achieved by a workaround. By using reflection, the internal browser object can be retrieved from the Watij package and its DOM can be modified. Figure B.1 shows the code used to do this.

B.3 Plugin Configuration

CRAWLJAX is configured using a properties file as specified by the Commons Java Configuration API[2]. Originally CRAWLJAX and its plugins were being configured through one

[1]Watij: `http://www.watij.com`

[2]Commons Configuration: `http://commons.apache.org/configuration`

```
/**
 * Return the corresponding internal browser DOM element
 * for element.
 */
public static HTMLElement
        getInternalBrowserElement(watij.elements.HtmlElement element)
        throws Exception
{
        IEHtmlElement ieHtml = (IEHtmlElement) element;
        Method ieMethod =
                ieHtml.getClass().getDeclaredMethod("htmlElement", null);
        ieMethod.setAccessible(true);
        HTMLElement HTMLe = (HTMLElement) ieMethod.invoke(ieHtml, null);
        return HTMLe;
}
```

Figure B.1: Accessing the internal browser's DOM using reflection.

file (crawljax.properties). The properties file is parsed by the `PropertyHelper` class which makes the configuration available to CRAWLJAX throughout execution. When configuration options are added to the properties file, the source code of the `PropertyHelper` class must be adapted as well.

A method for avoiding this is to extend the `PropertyHelper` using Aspect Oriented Programming (see Appendix C). An abstract aspect `PluginPropertyHelper` was added to CRAWLJAX which can be extended by plugins which require their own configuration.

The aspect places a pointcut on calls to the `checkProperties()` method declared in `PropertyHelper`. This method is called by the `PropertyHelper` after all properties are initialized to validate the loaded configuration. By placing an advice right before this call additional properties can be added to the `PropertyHelper`. By placing an advice around the call the additional properties can also be validated without interfering with the original `checkProperties()` function. The advices in `PluginPropertyHelper` only call abstract methods which means that any extending aspect can add its own properties.

The `PropertyHelper` offers convenience methods for reading configuration values and returning them with the required type (the `getProperty*()` methods). In these methods the `Configuration` object is used, which is linked to one properties file throughout the session. In order to allow plugins to use their own properties file, the `PluginPropertyHelper` aspect implements wrappers for the methods of the `PropertyHelper`. These wrappers, the `getPluginProperty*()` methods, temporarily replace the `Configuration` object with the plugin's `Configuration` object and restore it after the requested property was retrieved.

The main advantage of this approach is that each plugin can have its own configuration file. Another advantage of this is that plugins can be easily turned on or off, by adding or removing them from the `atusa.plugins` setting, rather than requiring to turn on or off all their configuration options.

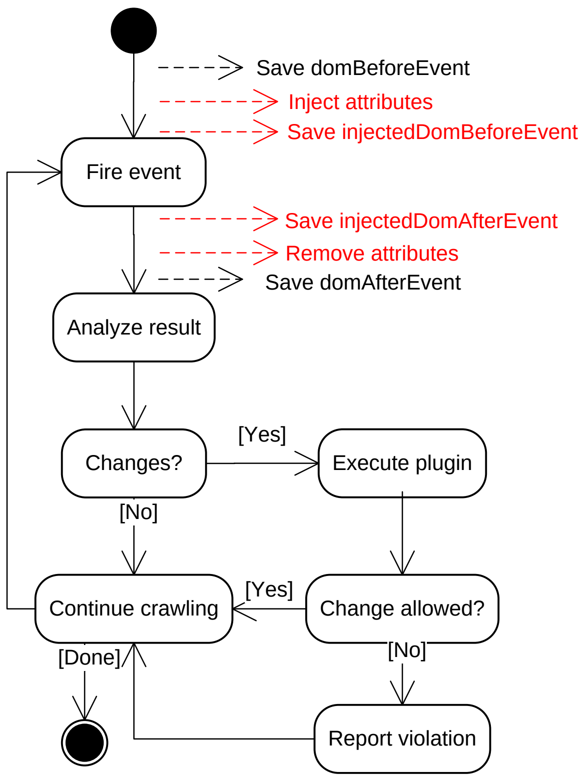


Figure B.2: Annotating variables during the crawling process.

B.4 Attribute Annotation

Some of the approaches discussed in this report require the annotation of an attribute on a DOM element in order to identify the element later on during the crawling process. Because annotating attributes changes the DOM, it is important to use and remove the attributes again before CRAWLJAX copies the DOM. When the attributes are not removed, CRAWLJAX will detect a new DOM state after each annotation and the crawling process may end up in an infinite loop.

The required annotation process is depicted in Figure B.2. This process is implemented in CRAWLJAX using an abstract aspect `AbstractInjectionAspect`. This aspect allows for the annotation of an attribute on a DOM element in the internal browser's DOM. It places a pointcut on the execution of the `fireEventWait()` method of the `IEBrowser`. By placing an advice right before and after the pointcut, an attribute can be annotated and removed without interfering with the execution of CRAWLJAX. To make the annotated DOMs available to the plugins for analysis they are being stored inside the `IEBrowser`. They can be requested using the `getInjectedDom*()` methods. The aspect uses the `AttributeInjector` class which offers functionality for performing and removing the annotation. This class also contains functionality for verifying whether a node already is annotated or whether a node should be excluded from the annotation process. Finally it is also possible to annotate a node and all its parents, which is a process that is used in widget boundary detection (Section 5.1.1).

B.5 DOM History

ATUSA executes its in-crawling plugins after a DOM change. With the introduction of our proxy for ATUSA (Section 6.2.1), HTTP requests have become another point of interest. It is possible that many events fired by ATUSA trigger HTTP requests without causing a DOM change. This means that at the time a plugin is started, requests which were not triggered by the last event may be present in the proxy buffer. ATUSA was extended to annotate a unique value into the DOM, which is appended to each request, everytime an event is fired. If requests in the proxy buffer were sent a number of events ago, their unique value cannot be found in the current DOM because it already has been removed.

To solve this, the DOM history was implemented. This object contains the last 25 DOM instances. When a unique value must be located in the DOM, the history is traced back until the value is found. The advantage of this approach is that despite the asynchronicity of AJAX, HTTP requests can be traced back exactly to the situation from which they were triggered.

Appendix C

Aspect-Oriented Programming

Some programming problems are difficult to solve using object-oriented programming without generating code clutter. An example of this is a logging system which writes a log entry at the entrance and exit of every public function. Such a system would require a call to `writeLog()` as first and last line of code of every public function, which results in cluttered code.

This problem can be solved using aspect-oriented programming (AOP) [19]. AOP allows for the generation of aspects, which describe crosscuts of system functionality. Solving the logging problem using aspects would result in an aspect, which describes that every function should be preceded and concluded with a call to `writeLog()`. This results in much cleaner code and abandons the need to add logging to every new component.

Another advantage of AOP is that existing code can be extended or changed without the need to physically change the source code. This allows for the extension of legacy or third party libraries.

A Java implementation of AOP is AspectJ [1]. AspectJ allows for the definition of pointcuts and advices. A pointcut describes a crosscut of the system, to which functionality must be added. A pointcut can match a number of execution points, called joinpoints. In AspectJ pointcuts can be defined on a number of actions, including function calls, function executions and variable accesses. An advice describes the action to be taken when a joinpoint is reached during execution. Advices can be applies before, after and around the joinpoint.

Figure C.1 shows the implementation of the logging problem using an aspect. The pointcut defines a crosscut on the execution of every public function. AspectJ is used to implement the approaches described in this thesis.

```
public aspect LoggingAspect{
        pointcut addLogging():
                execution(public * *(*));

        before(): addLogging(){
                writeLog("Before␣function:␣...");
        }

        after(): addLogging(){
                writeLog("After␣function:␣...");
        }
}
```

Figure C.1: Solution for the logging problem using AspectJ.

Appendix D

Vulnerabilities

Browser security measures cannot handle all security issues in web applications. Numerous types of vulnerabilities have been identified in web application security research [12, 30]. Some of the most common are discussed in this appendix. First cross-site scripting and injection, vulnerabilities which both exploit weak input validation, are explained. Finally Cross-site request forgery, an attack which exploits weak user validation, is discussed.

D.1 Cross-site Scripting

Cross-site scripting (XSS)[1] is an attack in which a malicious user exploits weak input validation by entering code that is eventually executed by a visitor [10, 12]. In 2007 the majority of attacks recorded were XSS attacks [32]. Characteristics of XSS vulnerabilities are that they are easy to fix, easy to exploit and can have a great impact when they are not fixed [30].

XSS is usually done by entering HTML code which includes JavaScript, which will be displayed somewhere later on. An example of this is a comment on a blog post. There are two types of XSS: transient and persistent. The difference between these is that persistent XSS remains on the page, for example because it is stored in a database, while transient XSS exists only in the current session. An example of transient XSS can be demonstrated by the following snippet of PHP code:

```
<p>An error occured: <?php echo $_GET['error']; ?></p>
```

The `$_GET` array indicates that the error message is sent via the URL, for example:

```
http://www.domain.com/page.php?error=112
```

It is displayed to the visitor without validation. A malicious user may exploit this by tricking a user into visiting the following page:

```
http://www.domain.com/page.php?error=<script>alert('XSS!')</script>
```

Because the value for `$_GET['error']` is not validated, it will be displayed on the page and the visitor will see a JavaScript alert box. Instead of displaying an alert box the malicious

[1]Because CSS is generally used to describe Cascading Style Sheets the acronym XSS is generally used to describe cross-site scripting to avoid confusion.

user could also have chosen to include more dangerous code like code that sends cookie details.

D.2 Injection

Injection attacks are based on exploiting weak validation of input which is used in a query of some sort later on. Examples of injection attacks are SQL injection, XPath injection and RSS injection [12, 30]. The idea of injection attacking is demonstrated by the following. On a website the following SQL query is used during the authentication process of a user:

```
SELECT * FROM login WHERE user = '$user' AND pwd ='$pwd';
```

If `$user` is input from a form and not validated, a malicious user may enter a value like `1' OR 1=1;--`, which results in the query:

```
SELECT * FROM login WHERE user = '1' OR 1=1;--' AND pwd ='$pwd';
```

The `WHERE` clause of this query is always true because of the `OR` clause and the semicolon which is the command delimiter in SQL. Because `--` indicates a comment in SQL, the part after the semicolon is not included in the query. The query will return the details for the first user in the table and since this is often the administrator, the malicious user is able to login as administrator. Similar approaches can be taken to inject code into XPath and RSS queries.

D.3 Cross-site Request Forgery

Cross-site request forgery (CSRF) [37] is an attack that exploits weak user credential validation. It is becoming more popular and entered the latest OWASP TOP 10 [32] at the fifth place. Many websites which require a user to login store the session ID in a cookie during the authentication process. When a logged in user visits a protected page, the server looks for the session ID in its session table and decides whether the user is allowed to view the page.

CSRF tries to exploit this behaviour by tricking the user into requesting vulnerable pages. An example is the following. A bank `www.insecurebank.com` allows logged in users to transfer money by visiting `transfer.aspx?to=113&amount=100.00`. When this page is requested, the session ID is validated and if the user is logged in, the money is transferred to the user with account number 113.

If the cookie containing the session ID is not removed after visiting the bank and a malicious user can trick someone into visiting a specially crafted URL that transfers money to the account of the malicious user, he may trick that person into transferring money without knowing it. This type of attack, called CSRF, is often done with a tag such as the `<img src="..">` tag, which requests the page without requiring a click. CSRF attacks are often combined with XSS attacks, preferably on websites with a high volume of traffic to maximize exposure.

Appendix E

JavaScript Code for AJAX Subversion

The JavaScript code of Figures E.1 and E.2 adds the `requestforproxyid` attribute of the `srcElement` on which an event was fired to AJAX calls. If the `srcElement` does not have the property, 12345 is added to the call.

```
/* jQuery framework AJAX subversion code */
if(typeof(jQuery) != "undefined")
{
 oldAjax = $.ajax;

 jQuery.extend({
  ajax     : function(s){
     var id = window.event && window.event.srcElement ?
                window.event.srcElement.getAttribute('requestforproxyid') : 12345;
     // attach the requestforproxyid value to the variables to send
     if(typeof(s) === "object")
       s.data.requestforproxyid = id;
     else if(empty(s.data))
       s.data = "";
     else
       s.data += "&";

     s.data += "requestforproxyid=" + id;
     // send the request
     oldAjax(s);
  });
}
// end of jQuery framework AJAX subversion code
```

Figure E.1: JavaScript code used for subverting AJAX calls in the jQuery framework.

```
/* Microsoft AJAX.NET framework subversion code */
if(typeof(Sys) != "undefined"){
  // for EFW use the following line
  Sys.Net.WebRequest.prototype.oldGetBody = Sys$Net$WebRequest$get_body;
  // for Pageflakes use the following line
  // Sys.Net.WebRequest.prototype.oldGetBody = Sys.Net.WebRequest.prototype.get_body;
  Sys.Net.WebRequest.prototype.get_body = newGetBody;
}

function newGetBody(){
  var body = this.oldGetBody();
  var id = window.event && window.event.srcElement ?
            window.event.srcElement.getAttribute('requestforproxyid') : 12345;
  if(empty(body))
    body = "";
  else
    addAmpsnd = true;

  if(typeof(body) == "object")
    body.data.requestforproxyid = id;
  else
  {
    // json request is not supported yet
    var regex = new RegExp("{.*:.*}");
    var m = regex.exec(body);
    if(m == null)
    {
      if(addAmpsnd)
        body += "&";
      if(!body.match("requestforproxyid"))
        body.requestforproxyid = id;
    }
  }
  return body;
}
//end of Microsoft AJAX.NET framework subversion code
```

Figure E.2: JavaScript code used for subverting Microsoft AJAX.NET calls.