NTNU
Norwegian University of
Science and Technology

Web Applications Security

A security model for client-side web applications

Deepak Prabhakara

Master of Science in Communication Technology

Submission date: May 2009

Supervisor: Svein Johan Knapskog, ITEM
Co-supervisor: Svein J. Knapskog, ITEM
Peter Sjödin, KTH
Lars Erik Bolstad, Opera Software

Norwegian University of Science and Technology
Department of Telematics

Problem Description

The Web has evolved to support sophisticated web applications. These web applications are exposed to a number of attacks and vulnerabilities. The existing security model is unable to cope with these increasing attacks and there is a need for a new security model that not only provides the required security but also supports recent advances like AJAX and mashups.

The attacks on client-side Web Applications can be attributed to four main reasons – 1) lack of a security context for Web Browsers to take decisions on the legitimacy of requests, 2) inadequate JavaScript security, 3) lack of a Network Access Control and 4) lack of security in Cross-Domain Web Applications.

This work will explore these four reasons and propose a new security model that will attempt to improve overall security for Web Applications. The proposed security model should allow developers of Web Applications to define fine-grained security policies and Web Browsers to enforce these rules; analogous to a configurable firewall for each Web Application. The Browser should disallow all unauthorized requests, thus preventing most common attacks like Cross-Site Script Injections, Cross-Frame Scripting and Cross-Site Tracing. In addition the security model will explore a framework for secure Cross-Domain Communication.

Assignment given: 08. December 2008
Supervisor: Svein Johan Knapskog, ITEM

Abstract

The Web has evolved to support sophisticated web applications. These web applications are exposed to a number of attacks and vulnerabilities. The existing security model is unable to cope with these increasing attacks and there is a need for a new security model that not only provides the required security but also supports recent advances like AJAX and mashups.

The attacks on client-side Web Applications can be attributed to four main reasons – 1) lack of a security context for Web Browsers to take decisions on the legitimacy of requests, 2) inadequate JavaScript security, 3) lack of a Network Access Control and 4) lack of security in Cross-Domain Web Applications.

This work explores these four reasons and proposes a new security model that attempts to improve overall security for Web Applications. The proposed security model allows developers of Web Applications to define fine-grained security policies and Web Browsers enforce these rules; analogous to a configurable firewall for each Web Application. The Browser has disallows all unauthorized requests, thus preventing most common attacks like Cross-Site Script Injections, Cross-Frame Scripting and Cross-Site Tracing. In addition the security model defines a framework for secure Cross-Domain Communication, thus allowing secure mashups of Web Services. The security model is backward compatible, does not affect the current usability of the Web Applications and has cross-platform applicability. The proposed security model was proven to protect against most common attacks, by a proof-of-concept implementation that was tested against a comprehensive list of known attacks.

Preface

This master thesis is part of the Master of Science degree program in Security and Mobile Computing (NordSecMob) at the Department of Telematics of the Norwegian University of Science and Technology (NTNU, Trondheim, Norway) and the Department of Information and Communication Technology of the Royal Institute of Technology (KTH, Stockholm, Sweden). It was carried out at Opera Software ASA (Oslo, Norway) during spring 2009.

I would like to thank my industry supervisor Lars Erik Bolstad for the sincere help and guidance and my professor Dr Svein Knapskog (NTNU) for his valuable advice. In addition I would like to thank Dr Peter Sjödin (KTH) for his support. Finally I would also like to thank my colleagues at Opera Software, especially Arve Bersvendsen and Wolfgang Mähr for the exchange of ideas, assistance during implementation and many useful suggestions to improve the thesis.

I certify that the thesis has been written by me and all the help received in my research work has been acknowledged. I certify that all information sources and reference literature used in my report has been cited correctly to the best of my knowledge. In addition I certify that this thesis meets the requirements for theses as set by NTNU and KTH.

Oslo, 11th of May, 2009

Deepak Prabhakara

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Abbreviations

WebApps	-	Web Applications
XML	-	Extensible Markup Language
AJAX	-	Asynchronous JavaScript and XML
API	-	Application Programming Interface
URL	-	Uniform Resource Locator
CSS	-	Cascading Style Sheets
HTML	-	Hyper Text Markup Language
HTTP	-	Hyper Text Transfer Protocol
POST	-	A type of HTTP request which sends data to a web server
TRACE	-	A type of HTTP request which sends data to a web server
GET	-	A type of HTTP request which sends data to a web server
DELETE	-	A type of HTTP request which sends data to a web server
DOM	-	Document Object Model
HCI-SEC	-	Human Computer Interaction - Security
W3C	-	World Wide Consortium
OWASP	-	Open Web Application Security Project
XST	-	Cross-Site Tracing
XSS	-	Cross-Site Script Injections
XFS	-	Cross Frame Scripting
CSRF, XSRF	-	Cross-Site Request Forgery
JS	-	JavaScript
JSON	-	JavaScript Object Notation
RFC	-	Request for Comment
RIA	-	Rich Internet Application
UI	-	User Interface
UTF	-	Unicode Transformation Format
JAR	-	Java Archive
DCOM	-	Distributed Component Object Model
CORBA	-	Common Object Request Broker Architecture
DoS	-	Denial of Service
SSL	-	Secure Sockets Layer
BEEP	-	Browser Enforced Embedded Policies
MET	-	Mutation Event Transforms
ZIP	-	Archive File Format
AIR	-	Adobe Integrated Runtime
ECMA	-	European Computer Manufacturers Association
ASCII	-	American Standard Code for Information Exchange

Chapter 1 Introduction

This chapter provides a brief introduction to the terminologies that will be used throughout this work. In addition it provides background information on the current state of security in Web Browsers.

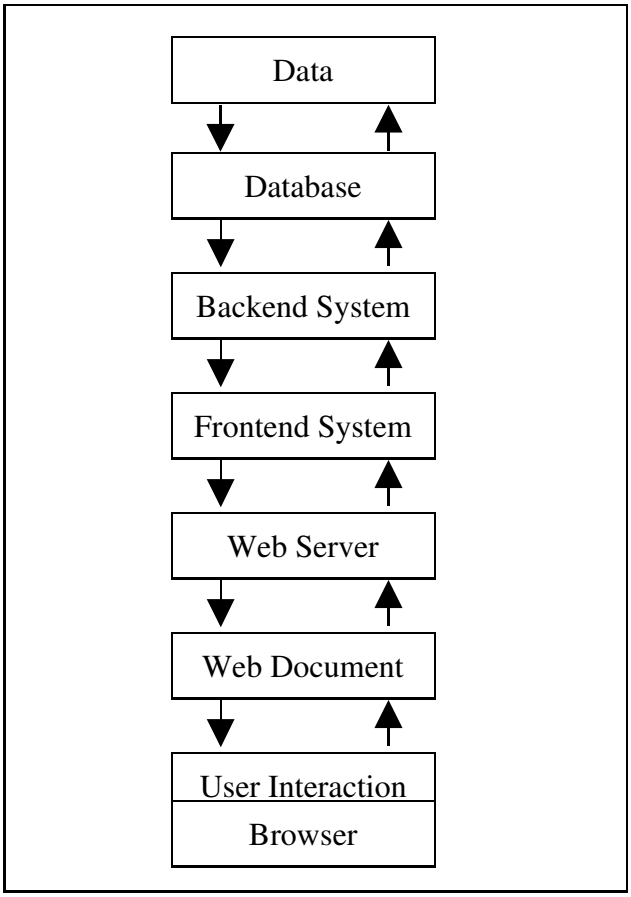
1.1 Terminology

All references to the word "Internet" is within the context of the interconnected network computers and all services provided by this network. All references to the word "Web" is in the context of the World Wide Web which is a system of interlinked hypertext documents accessible via the Internet. HTML is the markup language used to create hypertext documents. A "Web Browser" is any user agent that allows a user to view and interact with hypertext documents. JavaScript is a client-side scripting technique that is commonly used in combination with hypertext documents to create interactive sites. In addition to these terms just defined, the following terms will be used widely throughout the course of this work.

1.1.1 Web Applications

A Web Application (WebApp) is defined as an application which is accessed over any Internet communication protocol. It is built using web technologies and is conceptually a series of dynamic web documents in a standard format recognized by common browsers. A clear distinction that sets web applications apart from web documents is client-side scripting which enables a dynamic and rich user interface. However the concept has been evolving over the past few years with the advent of the Internet and web applications can now be seen as a broad spectrum ranging from simple documents to more complex concepts like Web 2.0. The basic components of a Web application are shown in Figure 1-1 **[48]**. The scope of a Web application is indeed vast and this thesis focuses only on the client side of Web applications which mainly consists of the Web Document, User Interaction and the Browser. The Web Server component is important and relevant here since the Browser directly interacts with the Web Server to fetch the Web application. However server-side security is not within the scope of this thesis.

Figure 1-1: Components of a Web Application



1.1.2 Asynchronous JavaScript and XML

Asynchronous JavaScript + XML (popularly known as AJAX) allows the creation of dynamic Web pages and has the usability of a Desktop application since the communication with the server is done is parallel with user interaction. It also allows multiple content and services to be integrated into a single Web application, popularly known as mashups **[1]**, **[2]**

AJAX is typically built on the following technologies:

- JavaScript **[3]**: a scripting language whose syntax is based on the Java language and is commonly used Web applications to implement client-side features.
- Document Object Model (DOM) **[4]**: a standard object model representation of HTML or XML documents. The DOM is accessible to the JavaScript code and provides a powerful way to dynamically modify the documents.
- Cascading Style Sheets (CSS) **[5]**: "a style sheet language used to describe the presentation of HTML documents. JavaScript can modify a style sheet at run time, allowing the presentation of the Web page to update dynamically."

Thus using AJAX, the presentation of a Web page can be modified dynamically. In addition, using asynchronous communication, enabled by the following technologies, the

data on the page can be updated dynamically avoiding the need to reload the entire Web page:

- XMLHttpRequest **[6]**: "is an API that allows client-side JavaScript to make HTTP connections to web servers and facilitates the exchange of data such as plain text, XML, or JavaScript Serialized Object Notation (JSON)."
- JSON **[7]**: "is a lightweight, text-based, language-independent, data-interchange format. It defines a small set of formatting rules to create a portable representation of structured data. It is generally considered a safe subset of JavaScript."

AJAX has several advantages because it is a cross-platform technique usable on many different operating systems, computer architectures, and web browsers as it is based on open standards. It has powered the Web 2.0 revolution and a number of popular applications are based on AJAX (fine examples would be Google mail, maps, docs) and allowed information flow through mashups. There have also been a number of free and open source implementations of AJAX frameworks and libraries which ease the development of Web 2.0 applications.

1.1.3 Mashups

A mashup **[1]** is a website or application that allows different content and services to be integrated into a single user experience. The mashup components typically use the data from each other and combine them to offer a new service. A very good example of a mashup would be Facebook which allows various applications to be added to user's profile (Zombies, iRead and Movies applications to name a few).

As a technology, mashup components can be conceptualized as separate AJAX applications which are integrated into an AJAX container. This container provides a framework for the components to interact with each other. When necessary, the container also provides a framework for cross-site communication. **[2]**

1.1.4 Security

Security is about trust and context, particularly so for Web applications. Trust is achieved by enforcing control over a resource and context helps in establishing the rules for enforcement. For Web applications the resource is typically a combination of the user and the data displayed to the user. Thus to achieve a good level of security in Web application it is necessary to: 1) "establish trust between the Web applications and the user" and 2) "establish trust between the Web applications and data or resources being served back to the user" (Steve Pettit 2001) **[48]**. In addition context plays a crucial role since security policies cannot be determined without a proper context. The Web browser typically enforces security policies since it has the required context.

1.2 Background

The web application environment has been very successful in providing rich interfaces and desktop application like behavior and performance. However the Internet and Web Browsers have been traditionally built for web documents and their security model is also built around this assumption. There is clearly a need to extend the capabilities of web applications and provide at least the following needs:

- Extended capabilities and permissions than web documents
- Consistent User Interfaces
- Platform Independence
- Extended API's – File Access, Network Access, Private Store, etc.
- Communication with other Web Applications to create mash-ups

With AJAX, Widgets and other new web technologies, a lot of extended API's have been provided by browser vendors, allowing mash-ups but in the process bypassing the existing security model.
Thus a new security model will aim to deliver the following benefits:

- Mechanisms to request for extra capabilities and permissions
- Match the needs of technologies like AJAX and Widgets, while at the same time ensuring security
- Mechanism for cross-site requests, so that mash-ups is possible
- Mechanisms to securely distribute Web Applications
- Make security more usable by hiding the hard parts of the security model in specifications and implementation, exposing only the essential parts for user interaction

A few key observations made by the author about the Web architecture that will help shape the security model:

- The current browser model is vulnerable to many attacks, due to its outdated security assumptions
- Context is very important in security. A set of actions might be perfectly safe under a given context and totally unsafe in another context
- The server does not know the context most of the times. However the Web Browser has full knowledge of the context
- The server should be allowed to dictate the security policies. These policies however should be easy to define and deploy. It should not radically change the way the Web works today
- The Web Browser should be used to enforce these policies. In this sense the web browser acts as a gatekeeper
- Presentation is more important than security, hence the new security model should not change the user experience drastically

1.3 Research Objective

The main objective of the thesis is to build upon the existing security model for Web Applications (See Chapter 2) and work out a new security model which will accommodate recent advances like Web 2.0 and mashups. This work explores the existing security model for Web Applications, implementation of the security model in current Web Browsers (IE, Mozilla and Opera), identify shortcomings in the current security model, study the needs of Web 2.0/mashups and analyze the various attacks on Web Applications. A new security model is proposed which attempts to make Web Applications more secure. A proof-of-concept implementation of the security model is to be tested against a list of known attacks to check the feasibility of the proposal.
To summarize, this thesis aims to achieve the following the objectives:

- Proposal for a New Security Model for Web applications, overcoming shortcomings like same origin model, cross-site scripting; and allowing secure cross-domain communication
- Proposal to extend the security model to other platforms, especially mobile devices
- Development of a proof-of-concept prototype for the security model
- Apply sections of the security proposal to revise the security model for Opera Widgets
- Possible standardization of the Security Model

1.4 Scope

The thesis mainly focuses on client-side web application security. The reference browser software that will be used is Opera 9.23 and all implementations and validations are performed on this browser. The thesis will strive to contribute a standardized, open and cross-platform security model, so that the whole Internet community is benefited from the findings.

1.5 Outline

Chapter 2 is divided into three parts. The first presents the current state of client side security model for Web applications and how AJAX technology has utilized a combination of existing web technologies to allow creation of interactive and powerful web applications. The second part describes the various vulnerabilities and attacks that WebApps are prone to and the third part describes commonly used mitigation techniques.

Chapter 3 explores related work in the field of WebApp security (especially in mitigation of cross-site scripting attacks) and presents the clear need to have a new revised security model. The discussion also covers a special class of WebApps; namely Widgets.

Chapter 4 describes the proposal for the new security model. The proposal address missing components in the current security model and attempts to fill in this void by using results from other related efforts, resources within Opera Software, the author's own observations and latest trends in Web technology. The proposal attempts to specify a new security model whilst ensuring backward compatibility.

Chapter 5 presents the evaluation of the new security model.

Chapter 6 discusses the evaluation of the work and shows how it is an improvement over the current security model.

Chapter 7 proposes ideas to extend this work further and highlights areas of potential research.

Chapter 8 discusses the achieved objectives and concludes the report.

Appendix contains documents and reports created during the course of this work.

Chapter 2 Current state of Web Applications security

The browser security model dates back to Netscape Navigator 2.0 released in 1996. Since then there have been a lot of advances in the capabilities of browsers including scripting, AJAX, etc. but the underlying assumption that "the content within a server is mutually trustworthy" has not changed. In this chapter the current browser security model is outlined.

2.1 Browser security policy

All browsers follow the same-origin policy which dates back to Netscape Navigator version 2.0. The same origin policy simply allows only documents and scripts loaded from the same origin as the loaded page to set and get properties of the document. Documents and scripts from any other origin cannot modify the loaded page.

The philosophy behind the same origin policy is simple: it is not safe to trust content from other sites, and hence access should be restricted to resources from the same website. Content and script loaded from other websites could be malicious since it is maintained by an untrusted entity.

The rule to determine whether two URL's belong to the same origin is outlined as "two pages to have the same origin if the protocol, port (if given), and host are the same for both pages". Table 2-1 gives examples of URL comparisons. The loaded URL in this case is http://www.myorigin.com/resource/index.html. **[8]**

Table 2-1: Same origin policy

URL	Result	Reason
http://www.myorigin.com/resource2/login.html	Allowed	Same protocol and host
http://www.myorigin.com/resource/internal/login.html	Allowed	Same protocol and host
http://www.myorigin.com:81/resource2/login.html	Blocked	Same protocol and host. Different port
https://www.myorigin.com/resource2/login.html	Blocked	Different protocol
http://m.myorigin.com/resource2/login.html	Blocked	Different host
http://myorigin.com/resource2/login.html	Blocked	Different host

However this policy can be overridden as has been seen in various AJAX techniques.

2.1.1 Vulnerabilities and Attacks

2.1.1.1 Same-origin policy violations

The same-origin policy is very important to protect user-data. If the same-origin policy was not implemented in browsers then documents and scripts from from one site could have access to the cookies, form entries, history of URL's visited, etc. from other sites. Thus the same-origin policy is considered one of the cornerstones of Web security.

Because the same-origin policy is so old and the Web has evolved, there are many shortcomings to the same-origin policy. Developers have been intentionally making

workarounds to this policy in order to develop more powerful Web applications. This has opened up Web applications to a lot of security attacks.

In **[12]**, the author describes a few examples of intentional workarounds to the same-origin policy. The most popular one is in Adobe's Flash (which is widely used in Web applications for videos and games) where the hosting site can set a configuration to disable the same-origin policy and leave the Web site vulnerable to CSRF attacks.

In **[13]**, the author describes a few examples of unintentional workarounds to the same-origin policy. The IE XmlHttpRequest JavaScript object was found vulnerable to header forgery due to a vulnerability in its same-origin policy implementation. The Adobe Flash plug-in was also discovered to have such a vulnerability.

When HTML files are run locally from the file system, the same origin policy is not enforced. Thus they are allowed to make HTTP requests to any server. This leads to potential problems where locally run HTML files can allow hackers to steal local file information.

2.1.1.2 External scripts

Web applications can load and execute external scripts. While this gives flexibility and allows creation of powerful Web applications, there are security implications of applying the same-origin policy on external scripts. Since the external script loaded runs in the same security context as the loaded page it has access to the loaded origin and can make requests to that origin. This means that an attacker could attack a web site by compromising its external scripts. **[13]**

2.1.1.3 Third Party Plug-in Vulnerabilities

Plug-ins complicate browser security because they are given unchecked access to browser internals, making it difficult for the browser to enforce security policies on plug-ins. Plug-ins are supplied to the browser in a binary format and usually loaded as a dynamically loaded library. Though plug-ins are provided with an API to interact with the browser, plug-ins run in the same address space as the browser, so they are free to modify browser structures as needed. Thus, a successful attack on a single plug-in leads to a full browser compromise.

Currently plug-in providers implement their own ad-hoc security mechanisms and policies for each different plug-in, which causes security problems even for un-compromised plug-ins. Security policy goals for the browser are not necessarily reflected by the plug-in security policy resulting in inconsistent accesses between the browser and the plug-in. Also, there can be differences in plug-in policy between plug-in implementations for the same content type. For example, different Flash players could allow different cross domain accesses based on their developers interpretation of Flash security policy. **[14]**

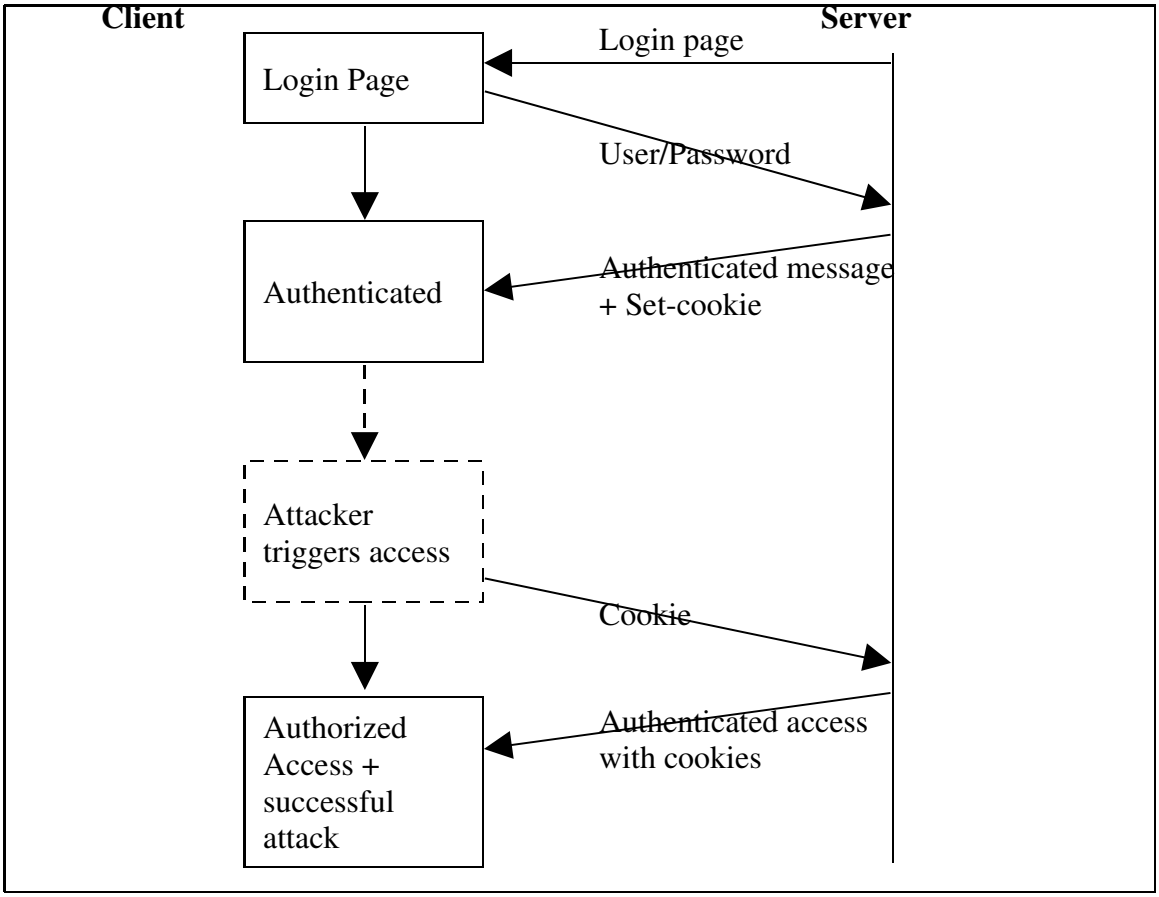
2.1.1.4 Cross-Site Request Forgery

Cross-Site Request Forgery (CSRF, XSRF, or cross-site reference forgery) is an attack that exploits the trust that a web site has established with the user's browser. A user can

unknowingly be tricked to submit a request to an already authenticated site. The following sequence of steps explain a basic cross-site request forgery attack: **[23]**

1. The attacker prepares a malicious link and sends the link to the victim (the link could be posted on a trusted site or sent by email or other means)
2. The victim clicks on the malicious link
3. The link takes the user to the malicious website where he is fooled into submitting a form whose action actually points to the targeted website
4. If the victim's browser has already authenticated the website, then the form submission will succeed
5. The form submission is designed to modify private user data like financial transactions or passwords

Figure 2-2: CSRF attack



Real-World Examples

- A CSRF vulnerability was discovered in Gmail where the attacker could trick a user to add a mail filter to their account. This allowed the attacker to have all mails from the user's account forwarded to his own address. **[24]**
- A CSRF vulnerability was discovered in Amazon's 1-click feature, which could cause victims to purchase items of the attacker's choosing without the victim's knowledge or consent. **[25]**

2.1.2 Mitigation Techniques

2.1.2.1 Countermeasures for same-origin policy violations

There is no protection against this in the current security model. There are numerous ways in which this policy can be overridden but no simple or straightforward way to mitigate this. This is definitely a major shortcoming of the current security model and needs to be revised.

2.1.2.2 Countermeasures for CSRF

A common countermeasure for CSRF is to check the Referrer header field in the incoming HTTP request. The legitimacy of the requesting page can then be checked. But the Referrer header is an optional field and hence this method is not reliable.

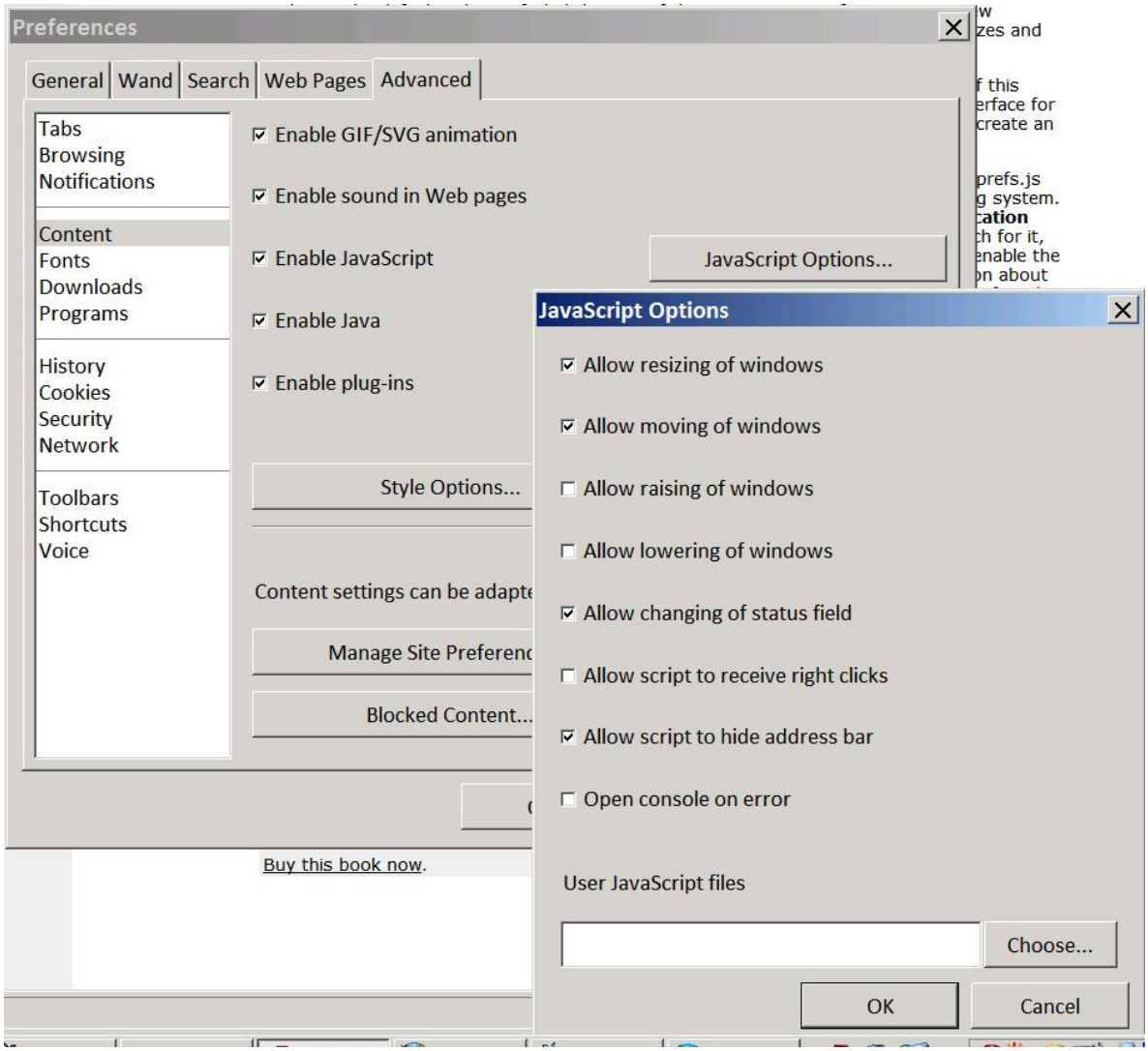
There is a second method which uses additional security tokens. This token is then inserted into all the client-side HTTP requests. As an example consider the original request URL to be http://csrf/attack. The server generates a security token with value abc and the Web application inserts this token value in all it requests – http://csrf/attack?secret_token=abc. Thus without knowing the security token it is not possible to launch a CSRF attack. It is necessary that the security token value must be unpredictable, otherwise security cannot be guaranteed. **[2]**

2.2 JavaScript Security

The JavaScript security model inherits from the security model for Java applications. All scripts run in a restricted sandbox, thus isolating them from the rest of the system and providing access to only the loaded document. Scripts do not have access to the local file system, memory space of other programs, other operating system specific internals and hence in theory malicious scripts cannot cause major damage to the system. However there are many vulnerabilities that can be exploited and scripts can escape the sandbox restrictions. **[9]**

Browsers offer Configurable Security Policies where the user can set fine-grain capabilities for each site. Opera provides a User JavaScript for this purpose **[10]**. However this is very technical and most casual users cannot make use of this feature. The author's proposal is to move these kind of policies to the server (the web server can send a policy list to the browser before each page or sets of pages is loaded). This will take away the complexity of defining the JavaScript policies and make the solution more usable for all users. However Opera does offer a UI to change the overall JavaScript policy easily as seen in the Figure 2-3 below.

Figure 2-3: JavaScript settings in Opera Browser



2.2.1 Vulnerabilities and Attacks

2.2.1.1 Cross-Site Script Injections

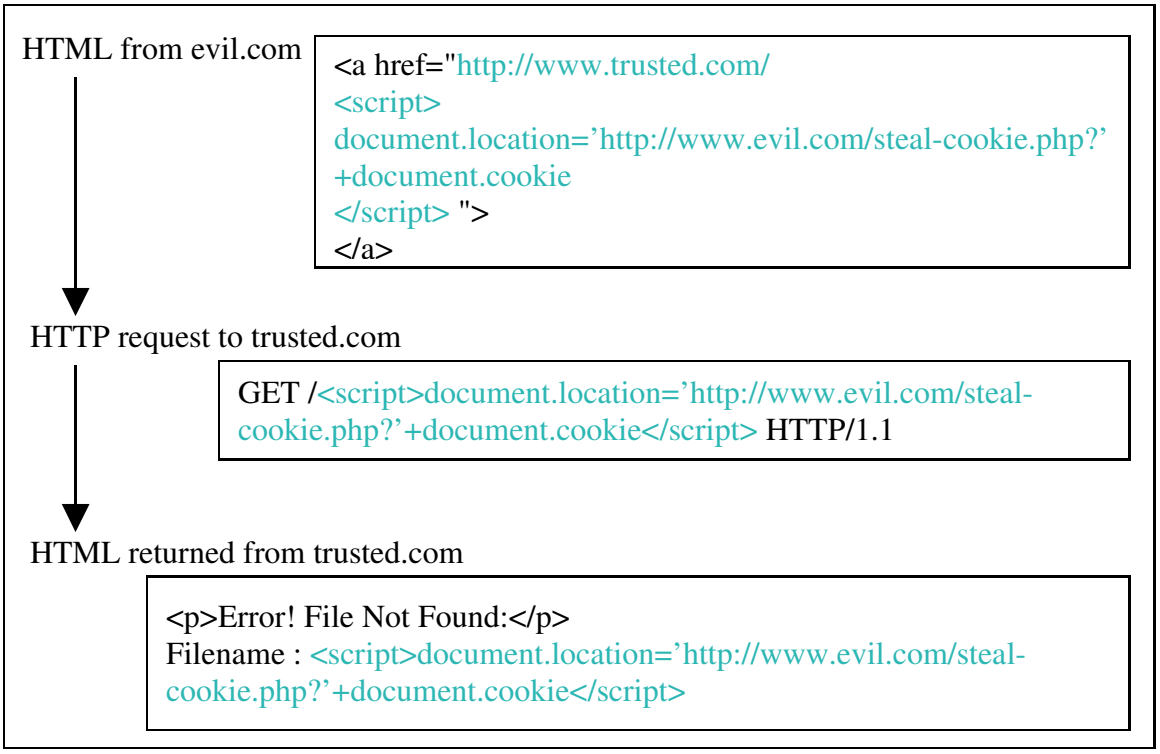
Cross-Site Scripting (XSS) is an attack technique that makes a browser execute injected scripts on a web site. The injected code is primarily JavaScript and HTML code but it could also be VBScript, ActiveX, Java, Flash, or any other browser-supported technology.

When the malicious code gets executed on the victim's browser, the attacker can potentially steal any sensitive and private data related to the web site from where its loaded. Cross-site scripting attacks exploit the trust the user places on the web site.

There are two main types of XSS attacks, non-persistent (reflected) and persistent (stored). Non-persistent attacks works by getting the user to click on a link that has been encoded with a malicious script. When the user clicks on such a link the malicious code is

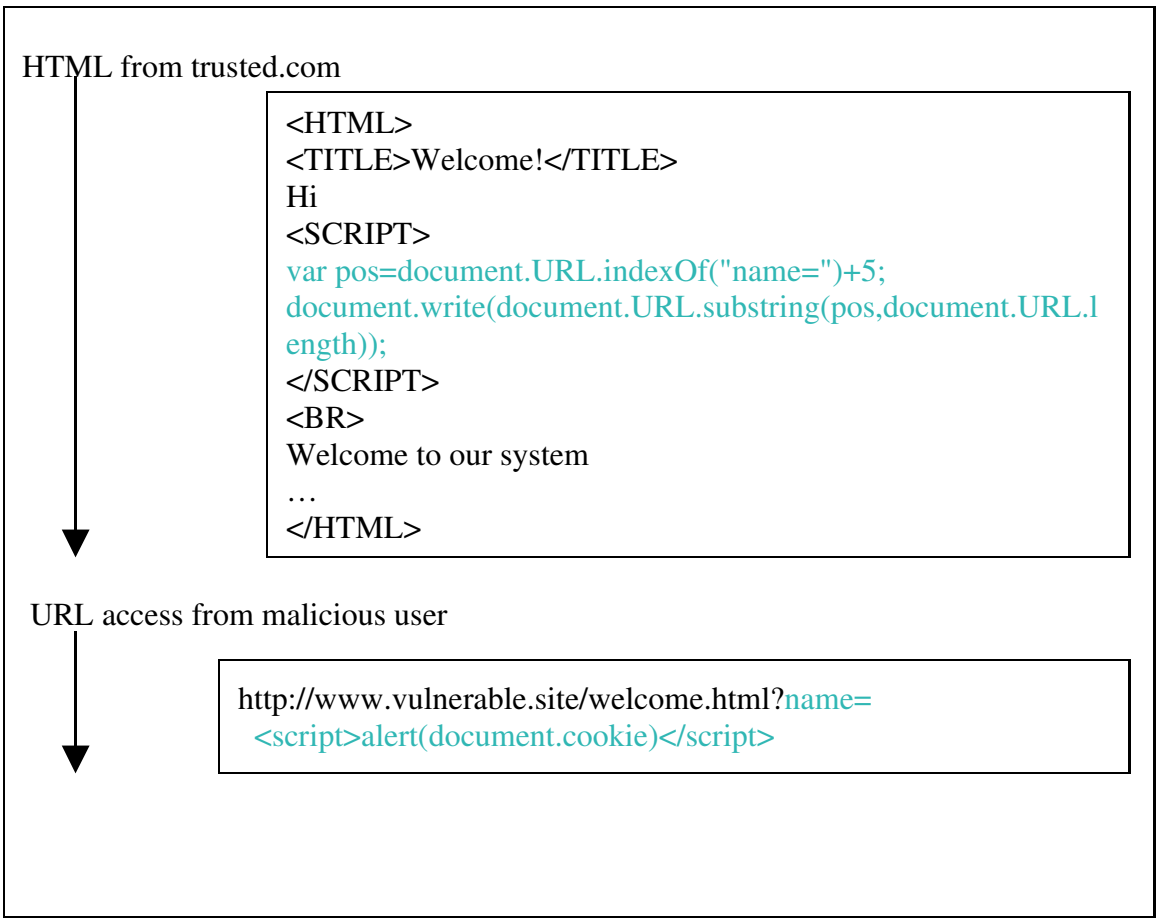
echoed from the server and gets executed by the browser. Persistent attacks works by submitting malicious code to the server where it gets stored persistently and is embedded into the web page. When the user visits such a page, the malicious code gets executed upon loading the web page in the browser **[15]**. Figure 2-4 shows how a typical non-persistent XSS attack works.

Figure 2-4: Non-persistent XSS example



Another kind of Cross-site Scripting attack, usually known as the third kind of XSS attack is DOM Based XSS. This is different from the other two kinds of XSS because it does not rely on any server side embedding but rather uses the DOM data in an insecure manner. **[16]**

Figure 2-5: DOM based XSS example



Cross-site scripting is one of the major web application attacks and a lot of information and documented research is available on this attack. **[13]**, **[17]**, **[18]**

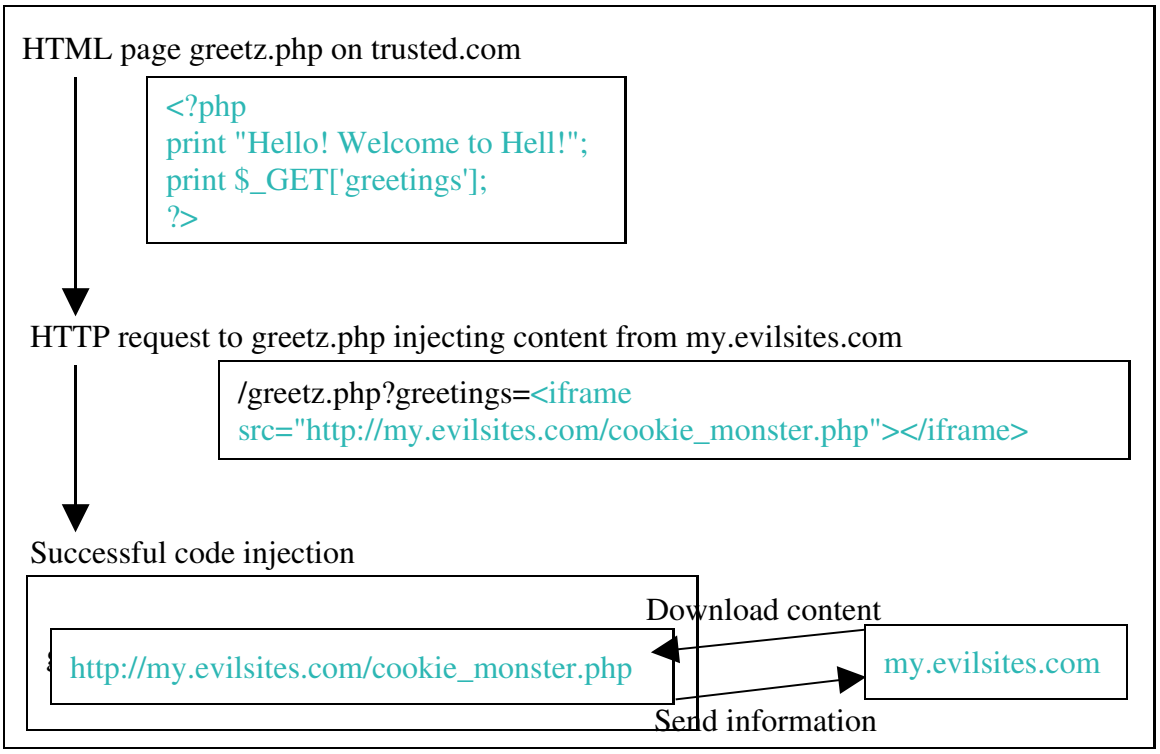
2.2.1.2 Cross-Site Tracing

Cross-site tracing (XST) is a cross-site scripting variant which exploits the HTTP TRACE method. The TRACE method is typically used by web servers for debugging purposes and it includes all header information. The attacker could use this header information and submit it to another site, thus completing the attack. **[19]**

2.2.1.3 Cross-Frame Scripting

Cross Frame Scripting (XFS) is a vulnerability that affects web applications that use frames in their web pages. Frames allow web pages to present the web content framed in different sections of the browser window. When the browser renders the framed web page it also tries to load the web pages that each frame references to. This can be done either via HTML tags or by using JavaScript either to load a web page within one frame or by calling a section of JavaScript code that resides in another frame. The security issue by calling JavaScript across the frames of a web page is that such scripts could be malicious if loaded through a user controlled variable. **[20]**

Figure 2-6: IFrame Insertion Attack



2.2.1.4 Web Cache Poisoning

Web cache poisoning is an attack which replaces the DNS address of a site with an address supplied by the attacker. When a user attempts to access the legitimate site, the browser makes a DNS call and the malicious address is returned to it. **[21]**

2.2.1.5 HTTP Response Splitting

HTTP Response Splitting is a variant of cross-site scripting where the attack is performed on the HTTP response header by exploiting the fact that in HTTP, the response header is separated from the body by two CRLF's (Carriage Return and Line Feed ASCII characters). By injecting additional CRLF characters the attacker is able to take control of the HTTP body or split the response. **[22]**

2.2.2 Mitigation Techniques

2.2.2.1 Signed Scripts

Scripts can be signed using a digital signature. Typically the entire web page and all the included scripts are signed and archived in a JAR file along with the valid digital signature. The security model for JavaScript is very similar to the security model for traditional Java applications. The digital signature is issued by a known certificate authority (for example VeriSign or Thawte) and signing guarantees that the script is in its original form and not tampered by a malicious entity. The signature also identifies the author of the script. Once the signature has been verified, the script has privileged access to browser resources. An access-control mechanism is defined for each script and this is

enforced by the browser while providing the extended privileges. The user has full control over granting the requested privileges. **[30]**

But signed scripts are hard to use in practice since most web developers cannot afford a widely recognized digital signature. Even though signed scripts can assure the origin and integrity of the script, there is no way to know what the script does or contains. Hence even signed scripts need user approval before they are allowed to run. **[9]**

2.2.2.2 XSS Mitigation

Avoiding XSS vulnerabilities is not a trivial task. This is because every situation is inherently different. The simplest countermeasure begins with encoding all HTML special characters. This prevents scripts from being executed but a side-effect is that it also prevents user-supplied HTML input. There has been a lot of research into various techniques to counter XSS. Content filtering and Input Validation are two common techniques to mitigate XSS.

Content Filtering and Input Validation

Jim et al **[28]** describes different techniques of content filtering and input validation. Figure 2-7 summarizes these various techniques described by the authors, that filter or transform potentially malicious content. The simplest kind of filter is to escape the special characters "<" and ">" and replace them with the HTML characters "&lt" and "&gt" so that the browser will display the characters instead of parsing them. When combined with technologies like tainting **[27]** that track potentially-malicious content, this is an excellent defense. But this approach filters all special characters and blocks not only scripts but also HTML content like bold, italics, bullets, etc. Therefore a lot of sites instead attempt to detect only scripts but this poses several difficulties:

- "Scripting techniques: Scripts can be embedded in a web page in many ways; Figure 2-7 shows some examples. Line 2 embeds a script contained in a separate file. Line 3 is an inline script. Line 4 is an event handler that will be attached to the img element on line 8, and which will be invoked when the user moves the mouse over the element. Line 5 is an inline CSS style declaration that says that the background of elements in class ".bar" should be obtained by executing a script. The script is invoked by the browser as it renders line 9. The script is contained in a javascript:URL; such URLs can appear in a document wherever any other URL can appear. Line 7 is an inline event handler that will execute when the body of the document has finished loading. Line 10 is an element that uses an inline CSS style to invoke a script. Line 11 embeds script in XML appearing in HTML; note that the script can be broken across multiple CDATA sections. Line 12 is a refresh directive that indicates that the page should be refreshed by loading a data:URL. The data:URL is the base64 encoding of a javascript:URL, and it is executed on page refresh. However this is only a partial list and browsers are constantly evolving to enable more scripting capabilities." (Jim et al, 2007) **[28]**
- "Encodings and quoting: Quotes that delimit content and encodings of special characters add further complications. There are multiple kinds of quoting and escaping (for URLs, HTML, and JavaScript), which must be stripped at multiple stages. There are multiple quote characters, plus cases in which quotes can be omitted. The base64 encoding of line 12 in Figure 2-7 is one example; others are line

13, which uses a javascript:URL that has been character encoded, and line 14, which uses HTML entity encoding to hide quote characters in a script (this can confuse filters that look for literal quote characters).” (Jim et al, 2007) **[28]**

- “Browser quirks. Script detection is also complicated by the fact that the process of rendering in the browser is ill defined. Different browsers can render pages in very different ways, so, what one browser sees as a script may not be a script to another browser. Furthermore, browsers make a best effort to render all pages, no matter how ill-formed: better to render something than show a blank page or an error message. This leads to some surprising script embeddings. For example, some browsers allow newlines or other non printing characters to appear in the “javascript:” portion of a javascript:URL, so that <img src=’java script:alert(1)’> will result in script execution. For another example, <img src=‘javascript:alert("Hello ’world’")‘> can execute in some browsers, even though backquote (‘) is not a standard quote character in HTML or JavaScript. Even something completely malformed such as <img """><script>alert("ack")</script>"> executes in some browsers.” (Jim et al, 2007) **[28]**

Figure 2-7: Different ways of embedding scripts in web pages (Jim et al, 2007)

```
1. <html><head>
2. <script src="a.js"></script>
3. <script> ... </script>
4. <script for=foo event=onmouseover> ... </script>
5. <style>.bar{background-image:url("javascript:alert(’JavaScript’)");}</style>
6. </head>
7. <body onload="alert(’JavaScript’)">
8. <img id=foo src="image.jpg">
9. <a class=bar></a>
10. <div style="background-image: url(javascript:alert(’JavaScript'))">...</div>
11. <XML ID=I><X><C><![CDATA[<IMG SRC="javas]]><![CDATA[cript:alert(’XSS’);">]]>
12. <meta http-equiv="refresh"
content="0;url=data:text/html;base64,PHNjcmlwdD5hbGVydCgnWFNTJyk8L3NjcmlwdD4K">
13. <img src=&#106;&#97;&#118;&#97;&#115;&#99;&#114;&#105;&#112;&#116;
&#58;&#97;&#108;&#101;&#114;&#116;&#40;&#39;&#88;&#83;&#83;&#39;&#41;>
14. <img src=javascript:alert(&quot;3&quot;)>
15. </body></html>
```

Robert Hansen **[38]** has compiled a list of XSS attacks specifically meant for filter evasion.

2.3 AJAX Security

AJAX has combined existing web technologies to provide a framework to build powerful web applications but at the same time it has modified the approach to web application development. The security weaknesses in AJAX is due to this change in approach making the traditional security model insufficient. AJAX supports a wide range of data exchange mechanisms ranging from XML, HTML, JSON, Web Services SOAP, custom objects, etc. While this has eased the data communication between the client and server, there are security implications since this information is dynamically used in the active JavaScript code making AJAX vulnerable to a variety of innovative attacks that are not possible in traditional web applications. **[11]**

2.3.1 Vulnerabilities and Attacks

AJAX application are vulnerable to attacks because developers avoid the same-origin policy opening up the possibility of malicious code insertion. Unfortunately, this leads to a few of the following mentioned attacks specifically targeted at AJAX applications. It is interesting to note that they are variations of cross site attacks already discussed earlier.

2.3.1.1 JavaScript or JSON hijacking

JavaScript Hijacking is a CSRF variant that exploits communication of confidential information using JavaScript. JSON is commonly used as a data transport mechanism in Web 2.0 applications, and JSON being a subset of JavaScript contains executable code. By overriding constructors and functions in JavaScript, an attacker is able to read the JSON information if a user is made to click on a malicious link. **[2]**, **[26]**

2.3.1.2 Denial of Service Attack

A denial-of-service attack (DoS attack) attempts to deny machine resources to the affected user. This is usually one by stressing the machine with requests to the point where the machine can no longer respond to legitimate requests. It is very easy to launch a DoS attack using JavaScript, especially when the Web application loads and executes 3rd party JavaScript. The 3rd party JavaScript can launch a DoS attack by repeatedly requesting resources from the desired Web sites. **[2]**

2.3.1.3 Mashups Attack Opportunities

Mashup applications allow multiple web applications to run within a single user experience. If one of the mashup components is malicious then it compromises the user and the web site. The malicious component can inject all sorts of malicious code into the application and attempt different kinds of attacks including XSS, CSRF, and DoS. In addition it might also be able to steal sensitive information. **[2]**

2.3.2 Mitigation Techniques

In addition to the XSS mitigation techniques like input validation, HTML encoding and content filtering; the following AJAX specific techniques are useful in securing AJAX applications.

2.3.2.1 JSON Security

Being a subset of JavaScript, JSON is vulnerable to exploits by injection of malicious code. However, JSON does not allow assignments and invocation and is generally considered a safer subset of JavaScript and the eval() function is typically used to convert JSON data directly into JavaScript objects. But this quirk is exploited by attackers to inject malicious code by sending malformed JSON data to the eval() function.

One approach to secure the use of JSON is defined in RFC 4627 **[29]**. This method uses regular expressions to verify that the JSON data does not contain active code. Figure 2-8 demonstrates such a regular expression:

Figure 2-8: Checking a JSON string with a regular expression

```
var my_JSON_object = !(/[^,:{}\[\]0-9.\-+Eaeflnr-u \n\r\t]/.test( text.replace(/"(\.|
[^"\])*"/g, ' ''))) &&
 eval('(' + text + ')');
```

But these regular expression parsers still have open security issues. For example they allow JavaScript variables to be referenced, thus allowing attackers to directly use sensitive variables like passwords without actually needing to know the value.
Due to the fact that JSON grammar is simple, these parsers have low performance penalty **[2]**.

2.3.2.2 Preventing JSON Hijacking

Since JSON hijacking is a variant of CSRF, the countermeasure is to apply CSRF protection as described in section 2.1.2.2.

There is a supplementary countermeasure that can be used with the CSRF protection techniques to provide additional security. This involves encoding of the JSON data so that it cannot be executed by the browser. A JavaScript comment (“/*”, “*/”) is commonly used to wrap the JSON data. **[2]**

2.4 Security in Opera Browser: A general note

Opera has been built with primary focus on security. The browser security features are more restrictive than other browsers and this generally makes Opera less vulnerable to attacks. According to the article by SecurityTeacher **[31]**, Opera offers a lot from a privacy and security point of view along with great usability. Opera is very secure and easy to use.

2.4.1 Security Features

External Web Content and Third-Party Plug-ins

Opera does not support ActiveX technology from Microsoft. ActiveX has been identified to be a major security vulnerability and is prone to malware and spyware attacks. Opera also doesn’t support Visual Basic, which is again considered a big security risk by many experts. Opera only supports Java and JavaScript, which are considered to be fairly secure. However as this thesis will show later in Chapter 4, the JavaScript security model has a lot of scope for improvement.

Opera does support third-party plug-ins such as RealPlayer, Adobe PDF, Flash, etc. However, as explained in Section 2.1.1.3, third-party plug-ins are themselves vulnerable to attacks and they affect the security of browsers in turn. Two interesting features of Opera are “Quick Preferences” which allows users to customize multiple content displaying capabilities and the “Site Preferences” which allows the user to customize site specific properties.

Figure 2-9: Opera Browser Quick Preferences

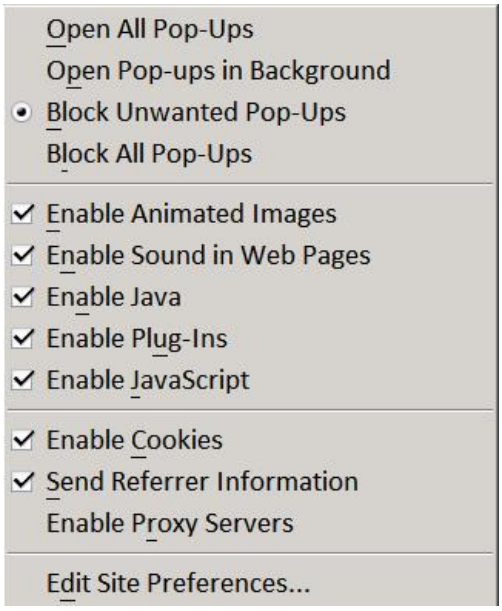
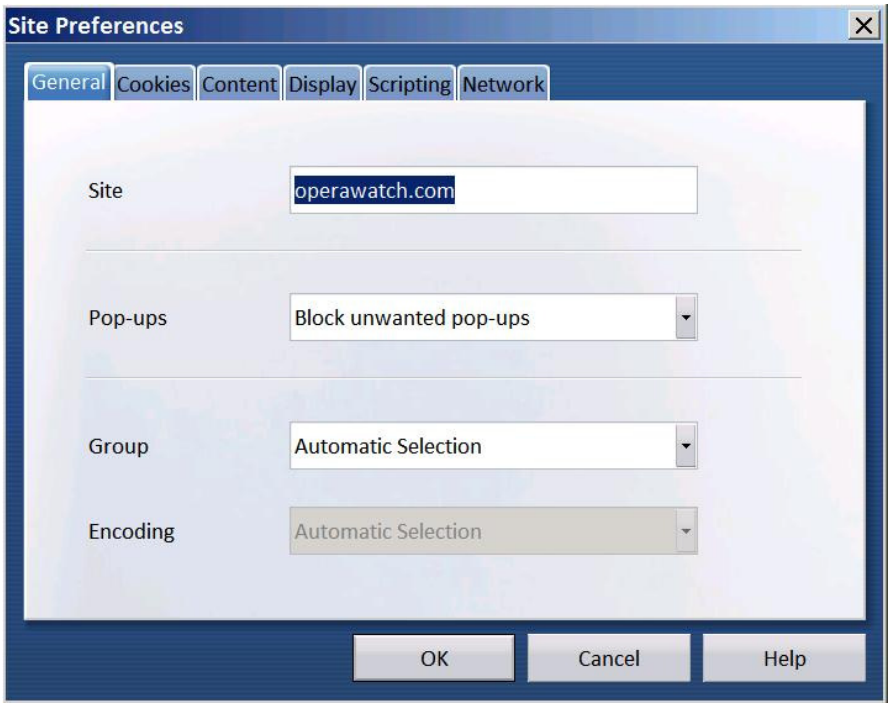


Figure 2-10: Opera Browser Site Preferences



Encryption Certificates and Security Protocols

Opera has visual clues built into the address bar which provide users information about the Security level of a page. SSL pages are identified by a closed padlock icon displayed inside a yellow security bar. By clicking on this bar the user can access more information about the certificate and its validity. “Opera supports Secure Socket Layer (SSL) versions

2 and 3, and TLS and offers automatic 128-bit encryption, the highest available security of any web browser." **[31]**

Automatic Updates and Security Patches

Opera has an automatic update check, which notifies the user of any updates to the browser and security advisories. This allows the users to get the fixes as soon as they are released by Opera. Opera is very quick to respond to security advisories.

Fraud Protection

Opera has an internal protection against Phishing (an online financial fraud where legitimate sites are impersonated with the intention of stealing financial or personal information). Opera alerts the user whenever it suspects that a spoof site has been visited. The tool is undergoing improvements and full-scale phishing protection should be available in the new Opera that will ship out later this year.

Figure 2-11: Opera Browser Fraud Protection

2.4.2 Privacy Features

Cookies and Referrer Logging

Opera allows users to customize cookie and referrer logging properties for each site. This is available through the "site preferences" option in the "quick preferences" menu as shown in Figure's 2-9 and 2-10. Opera also has a built-in cookie manager that provides advanced customization options.

Browsing History Cleanup

Opera has a "Delete Private Data" option that can be used to clear private data like cookies, web history, passwords, browser cache, file transfer history, etc. According to Simon Garfinkel **[41]**, this is still not fully secure as traces of the data are still left on the

hard disk. It is still possible to retrieve this information as it doesn't get completely erases from the hard disk. However this is more related to Operating System principles and beyond the scope of this thesis.

2.4.3 Security Bugs Patch Response

According to Secunia, Opera browser has a 100% patch record **[45]** – the highest among all browser vendors. This makes Opera very secure browser and it has been built up with a strong focus on security.

Chapter 3 Related Work

This chapter explores related work in the area of Web Application Security. Along with providing a brief summary of each work, inferences are drawn on how effective the methods are towards securing Web Applications and observations made on their weaknesses which can be addressed adequately in this work.

3.1 Alternate XSS Mitigation Techniques

This section explains a few interesting methods at mitigating XSS attacks. These methods are complementary to common techniques of content filtering and input validation. They mainly work to catch XSS attacks that slip through content and input filters.

3.1.1 Browser-Enforced Embedded Policies

Jim et al **[28]** discuss BEEP, a simple technique for preventing cross-site scripting attacks. Cross-script injections have dominated Web application vulnerabilities as more and more web sites have started to republish user content. Filtering of user content is a very common method used to counter XSS attacks. However, these filters can be evaded due to incompatibilities in the various browsers (as covered in Section 2.2.2.2), various scripting techniques and different ways of encoding data.

BEEP relies on two important facts.

- The web browser only executes scripts when it detects it while rendering the page.
- The web application developer has complete knowledge on which scripts are required to run the application (ie. The developer knows the security context).

Using these two facts BEEP allows web developers to write security policies which define the expected behavior of scripts. The browser then enforces these policies while detecting scripts on the page.

BEEP provides a very flexible mechanism for defining security policies. A JavaScript hook function is defined and implemented in the browsers, this hook function defines what scripts are allowed and what scripts are blocked. When a browser detects a script, it checks the hook function before it executes it. Thus by allowing the policies to be defined using JavaScript, BEEP gives the developers a method to define security policies just like coding a JavaScript function. This makes it easy to use BEEP. BEEP has implemented white-list and sandbox policies, but it allows for a lot of variants and extensions.

3.1.2 Noxes: A client-side solution for mitigation of cross-site scripting attacks

David Endler **[32]** discusses an automated XSS attack and predicts that XSS techniques will evolve to launch automated attacks without the need for human intervention. There are several methods to mitigate XSS attacks but all of these methods apply security at the application-level. This means that application providers have take action again XSS problems and many times the provider is unable to take timely action to prevent such attacks. Kirda et al **[33]** discuss Noxes; a personal web firewall which is effective against XSS attacks. Noxes claims to be the “first client-side solution” which provides protection without needed any action necessary from the application providers. Noxes protects

against XSS by providing users with a connection alert prompt when it suspects that sensitive information is being sent out.

3.1.3 Mutation-Event Transforms

Úlfar et al **[34]** discuss Mutation-Event Transforms (METs), a method to define security policies which are enforced on the client-side by Web browsers. METs are just like BEEP **[28]** in that they both define security policies which are written to the web application from the server-side and enforced on the client-side. Both define the policies in JavaScript. METs biggest advantage over BEEP and other similar techniques is the flexibility it offers in defining the policies. METs require browsers to implement a mutation-event callback which in addition exposes already present functions and variables.

3.1.4 Observation

BEEP is a very effective and novel method to mitigate XSS attacks. However the Web developer is expected to write JavaScript code to enforce the security rules. While most experienced developers can manage this easily, the solution could be quite complex for most hobby Web developers.
Noxes is fairly complex for most general users, because it requires the users to set up rules for what requests are allowed and what are not allowed. The rules may vary for different Web Applications making the creation of one set of global rules a hard task. Noxes does not adequately address JavaScript security.
MET again provides programmable policies that might deter Web developers from using this method.
BEEP and MET do not address Network Access Control.

3.2 Sandbox Security

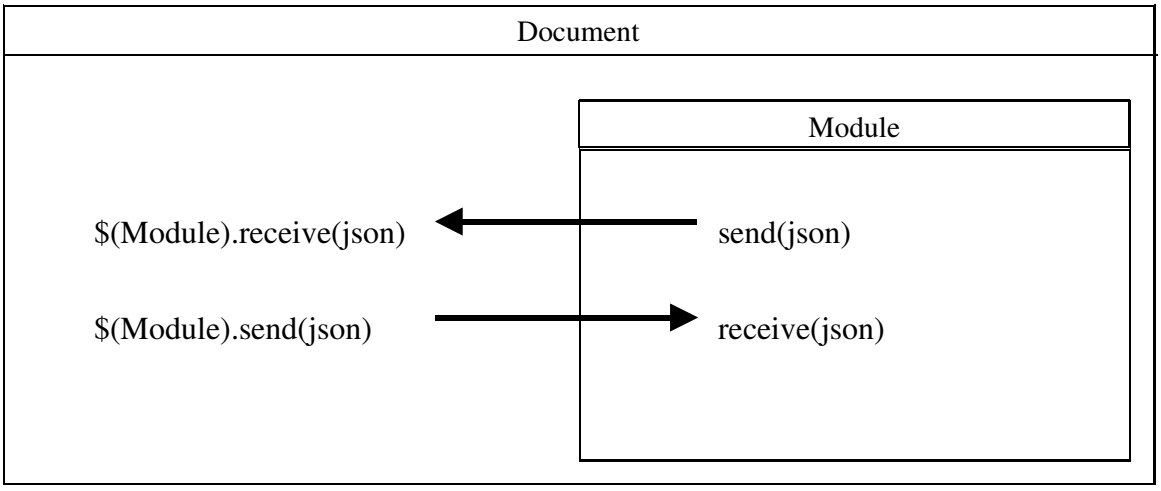
Douglas Crockford **[35]** discusses using the <module> tag to define a sandbox model in HTML. This change to HTML will support secure mashups. JSON data is exchanged between the document and the module it contains.

Figure 3-12: <module> HTML tag

```
<module id="NAME" src="URL" style="STYLE" />
```

A module is defined by three attributes and also contains two nodes for communication. The three attributes are id, src and style. The id is an identifier which can be used by scripts to access the module node, the src is the URL of the script or HTML file and style defined the size and location of the module. The two nodes can only communicate using a send/receive mechanism.

Figure 3-13: <module> architecture



3.2.1 Observation

The technique allows only one on one communication between parent and child, supporting only one port thus requiring the need to multiplex for more than one connection. This requires more work for broadcasting and communication between two or more modules.

3.3 HTML5 Security

The Working Draft of HTML 5 **[36]** describes a Cross-document messaging model that allows documents to communicate with each other regardless of their source domain, while not enabling any cross-site scripting attacks.

3.3.1 Observation

The technique does not allow the user to define fine-grain channels between components. Each component is required to perform its own access control mechanism on messages.

3.4 Special class of Web Applications: Widgets

Widgets are a special class of client-side web applications that uses the web technology as normal web pages but is allowed to run a separate applications. From the technology point of view, Widgets are typically AJAX applications written in HTML and JavaScript but can include any other web technology. Widgets range from simple applications that provide access to specific information like stocks, scores, weather etc. to complex mashups that combine information from multiple sources. Widgets differ from web applications in that they are downloaded and run locally on the machine.

3.4.1 W3C Widgets Security

Conceptually widgets are not web pages and hence they need a security model different from web pages. The author assisted Opera colleague Arve Bersvendsen in revising the Opera Widget Security Specifications. Opera is working closely with the W3C to standardize the widgets model and this revision was focused on improving security. The

new security model specification is outlined below. Before divulging into the details of the specifications, it is necessary to understand the components of a widget, especially the way it is packaged and installed. A widget typically consists of HTML code, JavaScript code, icons, resources and other files. These files are packaged into a ZIP format (identified by mime type: application/widget and file extension: .wgt) along with a widget configuration file "config.xml". The configuration file contains metadata about the widget - the name, description, icon, content, security elements, etc. In the current specification security is limited to just network access and plug-in. The new security model extends this security description to make the policy opt-in (which means the widget author has to specify the resources needed explicitly) and allow fine-grained policies to be defined. In the new model if no security policies are defined it is assumed that the widget doesn't need any access to the network or other resources.

Figure 3-14: Sample Widget Configuration File "config.xml"

```
<widget xmlns="http://www.w3.org/ns/widgets"
      id="http://datadriven.com.au/exampleWidget"
      version="2.0 Beta"
      height="200"
      width="200">
  <name>The example widget!</name>
  <description>
    A sample widget to demonstrate some of the possibilities.
  </description>
  <author url="http://foo-bar.example.org/"
    email="foo-bar@example.org">Foo Bar Corporation</author>
  <icon src="icons/example.png" />
  <content src="index.html"/>
  <access network="true"/>
  <license>
    Example license (based on MIT License)
    Copyright (c) 2008 The Foo Bar Corp.
    THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY
KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE
WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR
PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS
OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR
OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR
OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
  </license>
</widget>
```

New Widgets Security Model Specification
Introduction
Conceptually speaking a widget is "loaded" onto a computer before it is "run", and so its home domain is the computer on which it is running. In addition, many useful widgets accomplish their task by synthesizing information from various sources on the web. Thus the widget needs greater rights than normal web pages in order to access network resources. Yet the security concerns of unrestricted access cannot be ignored: The main concern is that a widget that accesses pages on an intranet can steal information, but there is also a general concern about leaking information from one web site to another.

This security model specifies what widgets are normally allowed to do, how they can be granted greater privileges, and how their privileges can be curtailed. Actions are controlled by reasonable defaults, with overrides from the user or system administrator, and checked against the behavior the widget declares itself to have.

A widget's config.xml file may contain a "security specification": a declaration of the protocols, hosts, ports, and pages that the widget will normally attempt to access. It may also contain declarations of other things that the widget will do.

The security specification is represented as the "security" element in the config.xml file. Additional local settings for widgets are stored in the configuration file widgets.xml. As for ini files, there can be fixed, global, and user versions of widgets.xml. A good user interface will be crucial to balance security and flexibility. This is covered in a separate companion specification called widget-ui (not in scope for this thesis).

This specification does not concern itself directly with signing, but an implementation of signatures could use different policy documents based on the identity provided by a signature, thus granting different privileges based on said signature.

The detailed specification is listed in Appendix A1.

3.4.2 Rich Internet Applications

Rich Internet Application’s (RIA's) are a special class of web applications that offer a rich, engaging experience that improves user satisfaction and increases productivity. Using the broad reach of the Internet, RIA's can be deployed across browsers and desktops. Adobe AIR is one such RIA platform. The security model of the Adobe AIR platform is discussed below and observations are drawn from this model.

AIR applications are designed to behave like native applications and hence have the same user privileges. Thus these applications usually have access to privileged operating system capabilities such as file system access, drawing directly to the screen, network communication, etc. A major advantage of AIR is that applications are generally safe from vulnerabilities such as buffer overflow and memory corruption since the runtime engine provides memory management. **[47]**

The Adobe AIR security features are outlined below. **[47]**

Code Signing
AIR applications need to be digitally signed to ensure integrity and to identify the origin of the application. Applications are signed with a certificate issued by a recognized certificate authority (for example Thawte or Verisign) provide assurance to the users that the application is legitimate and not been maliciously altered.

Security Sandboxes
AIR provides a security architecture which defines permissions for each file in the application, whether installed and loaded by the application. Each file is assigned a security sandbox and granted permissions based on their origin. A sandbox is a logical security grouping. Files that came installed with the application are put in an "application sandbox" and has full access to the entire AIR API. Loaded files are put in a separate restricted sandbox since it is risky to provide them full access to the entire API. Each sandbox is isolated from the rest to maintain local security.

Privileges of content in the application sandbox
As mentioned earlier the files in the application security sandbox have access to the entire AIR API. The other sandboxes have restricted access to AIR API's. There exist a number of JavaScript techniques to dynamically transform strings into executable code. Using such technique within the application sandbox poses a security risk since it is possible to execute a string which may contain malicious code. Such malicious code may come from remote sandboxes which are generally considered unsafe since they are loaded from external sources.

Privileges of content in the non-application sandboxes
Files loaded from external sources are assigned to a restricted "non-application sandbox". They behave like normal HTML content loaded in a browser and are only allowed to execute JavaScript. Malicious scripts cannot cause much harm since these sandboxes do not have access to privileged AIR API's.

Working securely with un-trusted content
AIR provides a method called "Sandbox Bridge" which bridges application sandboxes and non-application sandboxes. Use thing bridge a developer can expose certain services to the non-application content without allowing access to any privileged API's. In such cases the security level depends on how well the developer can utilize the sandbox bridge feature. The benefit of this method can be illustrated through an example. Consider a music store application that wants to allow external advertisers to connect to their application and retrieve only artists and CD information while hiding all other privileged information. A sandbox bridge can be useful in this case to provide only the artist and CD information to the non-application content.

HTML Security
HTML content in AIR application has different security considerations since it has the ability to use JavaScript. The application sandbox would pose a security risk if it allowed dynamically generated code (via. the *eval()* function) since malicious code can be injected and cause harm. Thus the various methods to generate dynamic code are only allowed

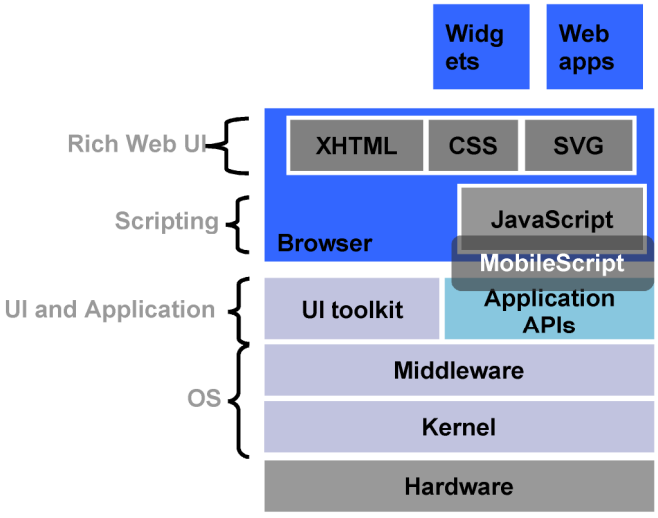
when for content that came installed with the application. However content from non-application sandboxes can generate dynamic code using all the possible methods since they do not have access to privileged APIs and cannot cause any harm. The sandbox bridge selectively opens up APIs to non-application content and thus developers have to careful while implementation of this feature.

Abode AIR has introduced new security features to enable development of rich Internet applications. However as it is clear from the above explanation of the security model, a lot of the responsibility lies in the hands of the developer to ensure good security practices are followed during the implementation of AIR applications. The security model still does not adequately address JavaScript security directly. But by limiting content within sandboxes, the harmful effects of malicious scripts can be contained within the sandboxes. A similar technique will be utilized in the thesis to improve the security model of JavaScript as will be seen in Chapter 4.

3.4.3 Mobile Widgets

Mobile Widgets are an extension of Desktop widgets, suited for small screens and provide additional device functionality by extending JavaScript to support MobileScript. Mobilescript is a common framework which allows quick and easy development of new services in mobile platforms, using an ECMA Script Open Source engine including new features which enable the developer to control new actions and events. MobileScript enables the horizontal development of services for any kind of mobile terminal supported. By providing a common standard, a wide range of mobile OS's (Windows Mobile, Symbian, Linux, etc.) can be supported **[46]**. The widgets are technically guaranteed to run on all supporting platforms without requiring any significant modifications. The following Figure describes the architecture of a typical Mobile Browser supporting MobileScript. **[50]**

Figure 3-15: Mobile Browser Architecture



But this raises serious security concerns through the same script injections and attacks that were discussed earlier. Chapter 4.4 discusses the security implications of MobileScript and proposes a security mechanism to safeguard devices against attacks.

3.5 Access Control for Cross-site Request (W3C working draft)

Anne van Kesteren **[37]** discusses a mechanism to enable client-side cross-site requests. It defines two algorithms for GET and non-GET methods that can be used to fetch cross-site resources. The server that contains the resource has to opt-in to the mechanism for this protocol to work.

The mechanism works in the following way:
Cross-site GET: The browser makes the GET request and the server returns a response that contains access-control information about allowed domains. If the domain is allowed then the browser makes the response accessible to the requesting script, otherwise it is blocked.

Cross-site POST/DELETE: POST and DELETE is done in multiple steps unlike the GET request. The browser first makes a GET request to the server along with the method that will follow. The server then returns a response containing the access-control information about allowed methods for sets of domains. The browser then checks the response to see if the domain and the method are allowed, if it is allowed then the browser makes the POST/DELETE request, otherwise blocks it.

3.5.1 Observation

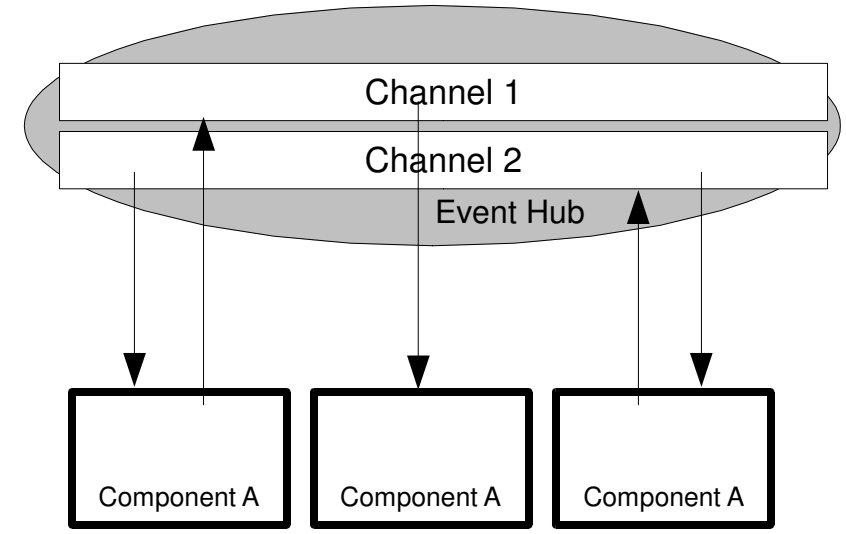
This technique does not adequately address cross-site attacks. The developers are still responsible to protect themselves against cross-site scripting attacks by not rendering or executing the retrieved content directly without validation. Another disadvantage is that this complicates things on the server for a complex web application since the access-control logic has to be repeated for every page on the server.

3.6 Secure Cross-Domain Communication – IBM SMash

Mashups provide a unified user experience of content and services from multiple sources and allow web applications to be as powerful as desktop applications. However as we have seen earlier, the current security model was not designed to support these kind of advances and developers have circumvented the security model to implement mashups. Fredrik et al from IBM **[51]** discuss Smash, a secure component model for mashups which ensures integrity of communication between the mashup components and prevents phishing of components. It provides specifications for a security policy that defines how the components communicate and interact with each other.

The coupling between mashup components is reduced by using iframe's, provided by HTML as containers for a document. These different iframe's communicate through channels which are contained in a centralized event hub. The communication is message based to avoid components from needing to know the internals of other components. Figure 3-16 shows a conceptual model of Smash.

Figure 3-16: Secure component model for mashups



3.6.1 Observation

SMash adequately addresses security related to mashups. It also protects against cross-site scripting, component phishing and other attacks. It also provides a mechanism to specify policies which is of importance since it can be combined with the policies defined by the new security model in this thesis.

3.7 Summary: The need for a new security model

XSS which is a major source of client-side security attacks on the Internet today is partly due to the flexibility in the JavaScript model which executes arbitrary scripts without any checks. Techniques like filtering and HTML encoding prevent scrip execution from arbitrary sources to a certain extent but they are not foolproof and there are several ways to bypass these techniques. An important observation about JavaScript is that it is the flexibility which makes it a powerful scripting language for Web applications, but it also makes it insecure. There are several ways in which a script can be included in a document and it is impossible for the filters to account for all these different ways. There is clearly scope for improvement in the JavaScript model which can make execution of scripts more manageable for the browser and developer. This will be discussed in detail in Chapter 4.

Access to the browser's network resources is currently free from any sort of policies. If a malicious script or attack has successfully managed to bypass the same origin policy then there is nothing that can be done to prevent the browser from completing the request. An

interesting observation is that the browser currently lacks any framework to define security policies per web application. This can be thought as analogous to having a configurable firewall for each web application that runs in the browser. The firewall would specify policies to enforce access control on network resources. So for example outgoing connections to only a particular set of hosts can be allowed or incoming connections can be allowed only from a dedicated source, etc. This will be discussed in detail in Chapter 4.

The same origin model is clearly not sufficient to accommodate Web 2.0 technologies. There is a need for a cross-domain communication model which works with current browser implementations as well as accommodates the advances in Web 2.0. There have been a lot of techniques and solutions to this problem as discussed a little earlier. Chapter 4 will discuss the details of an improved security model. The cross-domain communication technique will not be implemented as there are already stable and secure solutions for achieving this (notably IBM's SMash).

Chapter 4 Security Model Proposal

This chapter outlines the new model, which improves upon the existing model implemented in browsers by focusing on four areas – JavaScript, Network Access Control, Cross-domain Requests and Signing. The main contribution of the thesis is improvements to JavaScript security and a framework for defining policies which enforce access control on network resources accessed by the web application. Cross-domain communication model is derived from IBM's SMash framework which offers enterprise level mashup integration. Signing itself does not particularly make web applications secure and in practice it is not very practical. However for special classes of web applications like widgets and micro-widgets; signing offers trust, integrity and verification which enhance security.

4.1 JavaScript Access Control

As we saw in the earlier chapters JavaScript opens up scope for exploits because the scripts in a page are automatically trusted by the browser. Hence any evil script injected into the page would still be trusted since the browser has no idea how to distinguish between a valid and evil script. The thesis proposes a JavaScript Security Policy that allows the developers to provide such a context to the browser, so that scripts can be allowed or disallowed by the browser instead of just trusting all scripts. This approach is very similar to techniques proposed in BEEP **[28]** and MET **[34]**. However it improves upon these techniques by allowing the web developer to specify fine-grained policies by defining domains within which scripts are allowed to run. The sandbox model for scripts from **[28]** has been improved upon. The technique to whitelist scripts has not been included because this is a cumbersome process and requires the developer to patiently calculate the checksum for each script and include it in the policy. Instead the developer has flexible policies to define what scripts may run, for example only internal scripts could be allowed to run and all external scripts blocked or vice versa. Or only scripts in the <head> tag of the document may be allowed to run. In addition the developer can specify sources from where the external script may be allowed. Thus XSS can be easily avoided since the browser has a better context of the scripts and know which ones to allow and which ones to block. Further fine-graining is allowed by allowing the user to specify if access to cookies, frames, iframes, objects, etc may be allowed. This technique is complementary to other common security measures typically employed by developers, like client-side and server-side filtering and sanitization. The policy is opt-in and backward compatible, which means that the developer can opt to specify the policies to make it more secure. By not defining the security policy, the developer conveys his intention of rolling back to the old security model.

4.1.1 Defining Policies

The technique allows scripts to be restricted to domains – "all", "external", "internal", "head", "body" or "none". Multiple values of the domain may be used. For example "external|body" implies that only external scripts contained within the <body> tag of the document will be executed. The following table shows the allowed values for the Script Policy.

Table 4-2: Script Policy

Value	Definition
all	All internal, external, JavaScript URL's allowed. Default Value.
external	Only external scripts are allowed. The source of the script may be specified using the Script Host policy explained later in this section. All internal scripts are disallowed including JavaScript URL's.
internal	Only internal scripts are allowed. All external scripts are disallowed.
head	Only scripts defined within the <head> tag of the document are allowed. This value may be combined with "external" or "internal" value to define more specific policies.
body	Only scripts defined within the <body> tag of the document are allowed. This value may be combined with "external" or "internal" value to define more specific policies.
none	No scripts allowed. Equivalent to turning off JavaScript execution.

In addition a Script Source Policy may be defined. This contains a list of sources from which external script loading may be allowed. Default value is the host from which the document is served and is defined by the magic value "location". All other scripts from sources not defined in this list are blocked.

Table 4-3: Script Source Policy

Attribute	Value
Scheme	"javascript", "http", "https", "file", etc. Scheme name, as per Section 2.1 of RFC 1738 **[42]**
Host	network location and login information, port, as per Section 3.1 of RFC 1738 **[42]**
Path	URL path, as per Section 3.1 of RFC 1738 **[42]**

Further fine-graining (called Object Policy) is allowed by letting the developer decide if access to cookies, iframe, frame, object, etc. is allowed. And a list of sources can be defined (same format as Script Host Policy) such that the objects may make reference to only the list of defined hosts. This would prevent maliciously inserted scripts and objects from "leaking" information to a malicious site.

Table 4-4: Object Policy

Object	Value	Definition
cookies, history	"rw", "read", "write", "none"	"rw" is both read and write
iframe, frame, object, embed, applet, etc.	"all", "restricted", "none"	"restricted" means only access to sources included in the list of sources

4.1.2 Script Sandboxing

Sandboxing is a very useful technique which allows the web developer to include suspected strings within a sandbox for added security. Content inside the sandbox would only be allowed if it is normal text or html and any harmful scripts and malicious objects

would be blocked. Examples where this technique would be useful are blogs and wiki's where the site takes in user input and displays it back on the web page. Sandboxing is achieved by defining a tag with a class attribute "ScriptPolicy:sandbox" and including the suspected string within the element. This technique is very similar to the one defined in BEEP **[28]** but is more flexible because it doesn't require the developer to include the content within the element through the innerHTML method. Sandboxing is not allowed for scripts inside the <head> element.

Figure 4-17: Script Sandboxing Example

```
<div class="SecurityPolicy:sandbox"><A HREF=javascript:alert('This shouldn't
execute if sandboxing works correctly!')>me</A></div>
```

4.1.3 Delivering the Policies

An important part of the Script Policy is the actual delivery of the policies to the browser. The policies can be delivered in a JavaScript contained within the <head> element of the HTML document. The script is included before all other scripts. The browser would then execute this script first and the policies would get loaded. The policies are defined within the script either as an xml string which is then parsed or as comments in the script which can again be parsed to obtain the defined policies.

Figure 4-18: Example of using a script to deliver the security policy

```
<script>
// SecurityPolicy
// SecurityPolicy:scriptPolicy = "external|body"
// SecurityPolicy:scriptSource = "location"
// /SecurityPolicy
</script>
```

A second method is the elegant use of the <meta> tag to define a policy using the name and content attributes.

Figure 4-19: Example of using the <meta> tag to deliver the security policy

```
<meta name="SecurityPolicy:version" content="1.0" />
<meta name="SecurityPolicy:scriptPolicy" content="external|body" />
<meta name="SecurityPolicy:scriptSource" content="location" />
```

A third method would be to define a separate policy file which the browser would download and parse. The policy could be defined as in XML format or in <meta> tags. The advantage of this method over the two described above is that it is easy to describe just one policy file for an entire web application, instead of defining the policy on each document served.

4.1.4 Formal Syntax

```
SecurityPolicy      = <meta name=Policy  content=Value|ObjectValue /> ;
Policy              = Version | ScriptPolicy | ScriptSource | ObjectPolicy |
```

```
                    ObjectSource;
Version             = "SecurityPolicy:version"
ScriptPolicy        = "SecurityPolicy:scriptPolicy" ;
ScriptSource        = "SecurityPolicy:scriptSource" ;
ObjectPolicy        = "SecurityPolicy:objectPolicy" ;
ObjectSource        = "SecurityPolicy:objectSource" ;
Value               = VersionValue | "all" | "external" | "internal" | "head" | "body" |
                          "none" | "rw" | "read" | "write" | "restricted" | Host ;
VersionValue        = [1-9].[1-9]
ObjectValue         = "location" | URL defined by RFC 1738 ;
Host                = "location" | URL defined by RFC 1738 ;
```

4.2 Browser Network Access Control

A majority of the attacks involve leaking information to malicious sites. This can easily be prevented if the browser implemented an additional layer of security on its network resources. There are two interesting observations from the way we applications work today which will help in designing such an access control. One, the server has the best knowledge about which sites to allow access to and two, the browser has the right context to implement the access control. What this essentially means is that the server delivers the network access control policy (specified by the developer of the web application) and the browser enforces these policies. This ensures that the server (developer) is in control of deciding who should be given access to which resource. This is analogous to a highly configurable firewall for each web application. The rules of the firewall are flexible and allow the developer to either increase or decrease security as required by his web application. For example he can increase security by limiting the POST requests to his server to avoid CSRF and at the same time decrease security to allow mashup services. This is done through fine-grained policies as defined in the next section. This technique has also been explored by Justin Schuh. **[49]**

4.2.1 Defining Policies

The policies are very similar to those defined in a firewall. The following fields are available for defining the policies: Rule, Direction, Domain, Port, Protocol, Source Path, Destination Path, Method and Object. The fields and allowed values are explained in table below.

Table 4-5: Network Access Control

Field	Value	Definition
Rule	"accept", "reject"	Defines whether the policy should be accepted or rejected
Direction	"in", "out", "both"	Defines the direction of connection that is allowed: inbound, outbound or both
Domain	Network Location	network location and login information, port, as per Section 3.1 of RFC 1738 **[42]**
Port	Number or Range	Port Number
Protocol	Protocol or Scheme	Scheme name, as per Section 2.1 of RFC 1738 **[42]**
Source Path	URL path	URL path, as per Section 3.1 of RFC 1738 **[42]**
Destination Path	URL path	URL path, as per Section 3.1 of RFC 1738 **[42]**
Method	"GET", "POST", "HEAD", "PUT", "DELETE", "TRACE"	Defines the connection method allowed
Object	"user", "img", "script", "link", etc.	The Object that initiates the request

These set of nine fields allow fine-grained policies to be defined for a wide range of web applications including mashups (along with the cross-domain communication described in chapter 4.3). These network access policies along with the enhanced JavaScript Security covered in chapter 4.1 ensure that malicious scripts and code is prevented from applications and even in the case when new techniques and attacks are discovered the network access ensures that user data and information is not leaked to malicious sites. Such attempts would get blocked at the browser level and the only way to launch a successful attack will be to take over the domain which serves up the web application. And that is already a very hard thing to do due to continuous improvements to server side security. Thus these techniques together with server side security techniques ensure a complete end-to-end security for web applications, while accommodating Web 2.0 technologies and ensuring backward compatibility with old browsers. In the absence of the new security policies, the web application defaults to the same-origin model security. Several examples of defining the network access policies are contained in the Appendix. The next section covers an important aspect of the security policies, namely secure delivery of the policies to the browser.

4.2.2 Delivering the Policies

An important part of the Policy is the actual delivery of the policies to the browser. Each policy contains a version number so that the browser can refresh the policy file if the version number has changed. Each web application could have a single policy file containing the policies as XML data (networkpolicy.xml). The browser would fetch the policy file first and then fetch the web application. If the policy file is missing then the browser reverts to the default same-origin policy model. There could also be different versions of the policy file for different sections of the web application depending on whether the policies differ when the user has logged in. This has the benefit of one policy file for a single web application. Another alternative to delivering the policies is using the

<meta> tags. The browser would then parse the policies from these tags before loading the page. This also avoids attacks like web cache poisoning since the policies are delivered along with the web page. However this creates complications for a complex Web Application since the policies will have to be included in every page belonging to the WebApp. However this can easily be overcome by using a unique identifier for the policies delivered on the first page retrieved and the rest of the pages refer to this unique ID. For example the page *login.html* could deliver the policies for the Web Application and subsequent pages like *scrapbook.html, options.html* etc. can refer to the unique ID instead of repeating the policies again.

4.3 Secure Cross-Domain Communication

In Chapter 3.6, IBM SMash was discussed. SMash allows secure cross-domain communication to create mashups. A complete security model is achieved by combining the security techniques described in Chapter 4.1 and 4.2 with the SMash solution. This can be achieved in the following way. JavaScript policies are defined for each web application. When these applications are used as components of a mashup, the security policies can be extracted to construct a SMash security model which defines the communication model.

4.4 Cross Platform Security Policy

This section covers the cross-domain applicability of the Security policies defined above. JavaScript security for widgets can be extended from the Sandbox model. This is especially useful in micro-widgets where the widgets have access to the native device API's through DOM. For example access to the AddressBook and Messaging System can be provided through DOM API's. Just like how a Sandbox domain was defined, other domains can be defined which contain the capability of the script. For example a NetworkAccess domain would mean that scripts defined within that domain are allowed to access the network to send and receive information. Further fine-graining is possible (for example only receiving allowed while sending is blocked) depending on the needs of the platform. This ensures that the browser is aware of the widgets script capabilities before it is allowed to run and any policy violations are detected and prevented. As an example of such a policy consider a Windows Mobile micro-widget which has access to Network, Personal Information Manager (Address Book, Calendar, etc.), File System and Messaging System (MMS, SMS). Then the following capabilities can be defined – NetworkAccess, PIMAccess, FileSystem and Messaging. Further fine-graining is done by defining whether each of these resources can be read, write, read-write or no access. NetworkAccess and Messaging can be send, receive, both or none.

Chapter 5 Evaluation

Evaluation was performed on two levels. The first level was to attempt to approximate the overhead added by the JavaScript prototype. The second level was to write a custom html page and attempt various XSS attacks outlined by Robert Hansen **[38]**.

The test suite consisted of local copies of web pages from About.com, Adobe.com, Amazon.com, Apple.com, BBC.com, CNN.com, Craiglist.com, Ebay.com, maps.google.com, Microsoft.com, MSN.com, Opera.com, Techreview.com, Wikipedia.com, Yahoo.com and Youtube.com. Local pages were used to avoid network latency from affecting the results. Security policies were added manually to all the saved pages by including them in <meta> tags.

Figure 5-20: Security Policy used for evaluation

```
<meta name="SecurityPolicy:version" content="1.0" />
<meta name="SecurityPolicy:scriptPolicy" content="all" />
<meta name="SecurityPolicy:scriptSource"
content="location,http://*.google.com,http://*.apple.com,http://*.about.com,htpp://*.adobe.com,http://*.amazon.com,http://*.bbc.com,http://*.cnn.com,http://*.craiglist.com,http://*.ebay.com,http://*.microsoft.com,http://*.msn.com,http://*.opera.com,http://*.techreview.com,http://*.wikipedia.com,http://*.yahoo.com,http://*.youtube.com" />
```

Overhead Estimation

Three automated runs were performed and the following observations were recorded – total load time of the page, total time taken by the additional policy checks, total number of external scripts in the page and total number of internal scripts in the page. In addition the percentage overhead was calculated using the total load time and time taken by the policy checks. After the three runs, the average overhead percentage was calculated to be approximately 8.3%. The following graphs detail the results of the test.

Figure 5-21: Graph showing total load time and overhead time

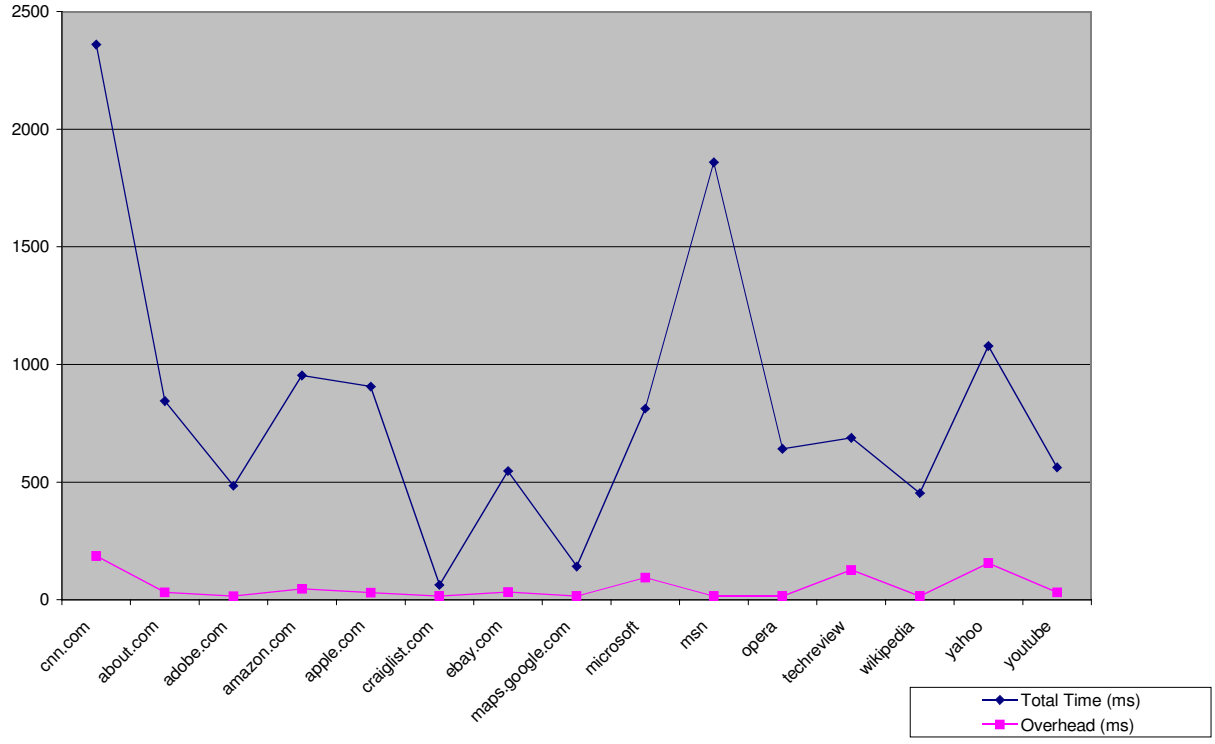


Figure 5-22: Graph showing percentage of overhead

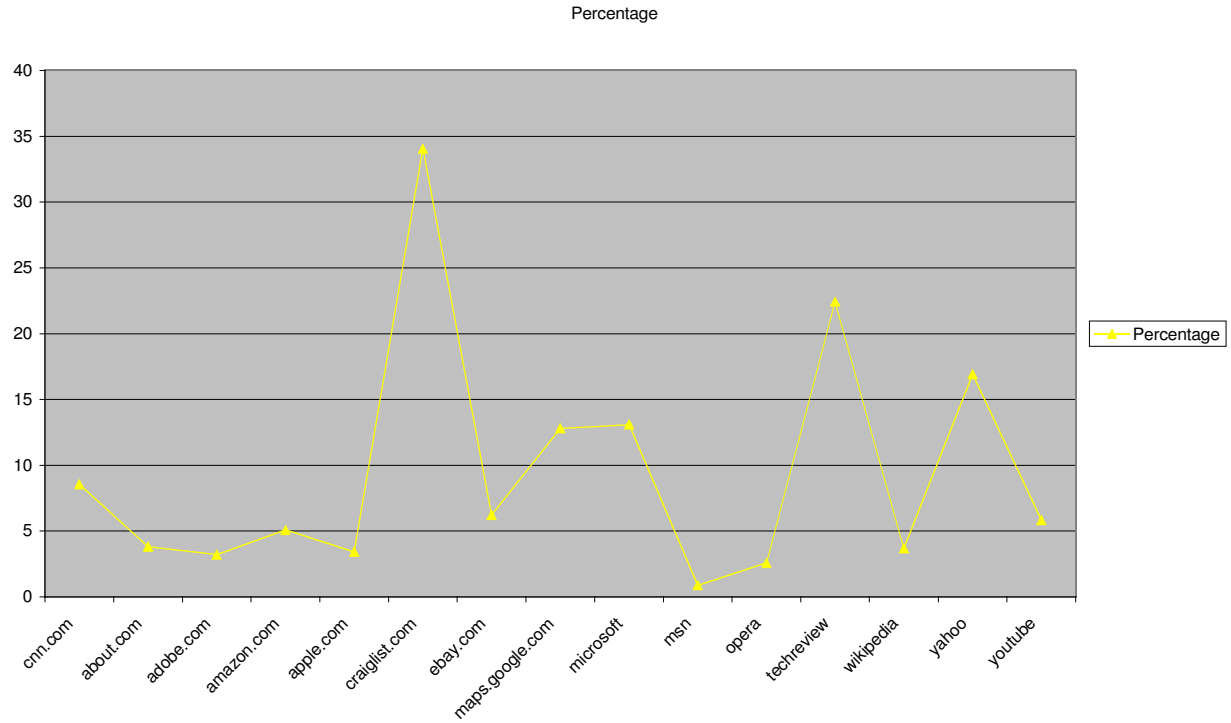
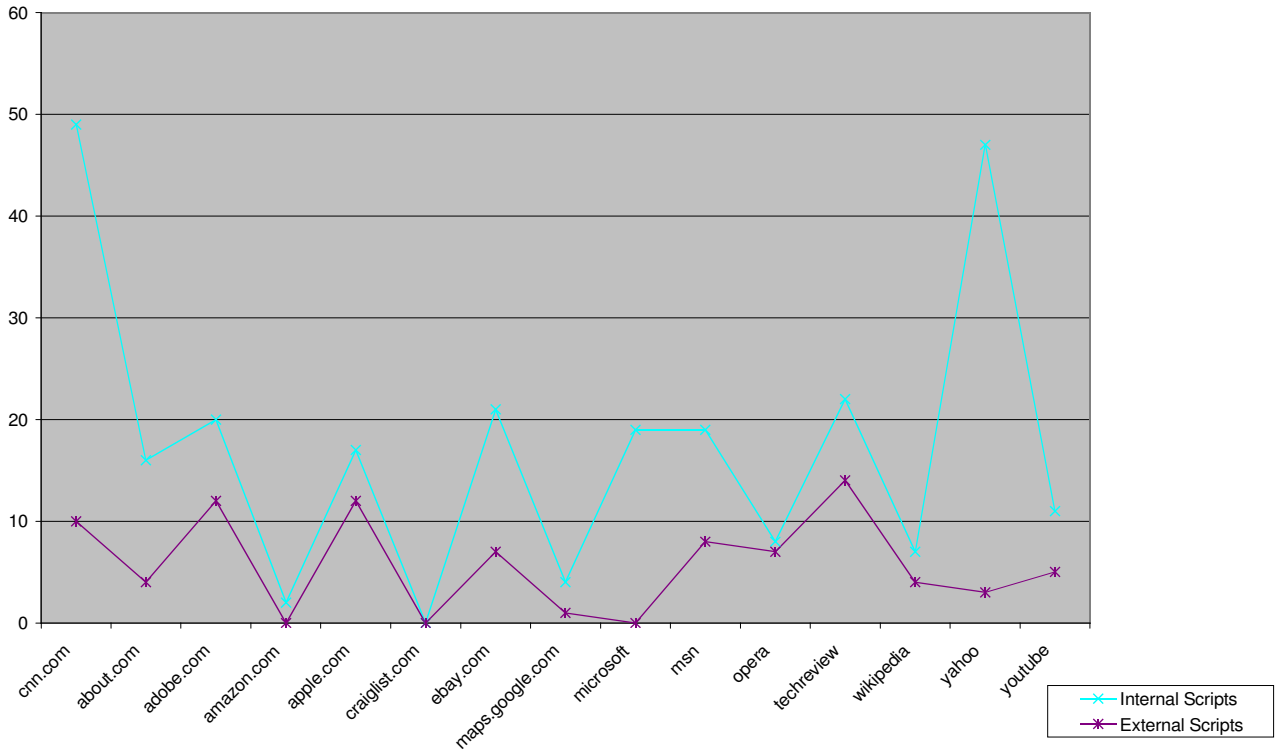


Figure 5-23: Graph showing total number of external and internal scripts



Mitigation of XSS attacks

An html page was written with XSS attacks outlined by Robert Hansen **[38]**. With the security policies correctly defined, the prototype successfully avoided all given XSS attacks. The sandbox policy successfully allowed valid html content to be displayed in the page whilst blocking all kinds of scripting attempts.

Recent work on Mobile Web Security

The BONDI **[52]** project launched by Open Mobile Terminal Platform (OMTP) **[53]** has released security specifications for the Mobile Web (in early 2009). The section on "JavaScript API Access" and "Security Framework for API Access" is very closely related to the work proposed in section 4.4 (Cross Platform Security Policy) and is a validation of the intention of this thesis work towards standardization of the proposed security model.

Chapter 6 Discussion

The Internet has become a powerful tool for communication, enabled creation of complex services and allowed businesses to build an online presence. However there have been long lists of attacks and vulnerabilities being discovered and exploited everyday. Among these XSS attacks hold a special place and are receiving a lot of attention. Rightly so because these kinds of attacks are very common in Web 2.0 application and mashups of web services which combine content generated from different sources. The current security model of browsers is not designed to handle these XSS attacks efficiently and hence there is a need to revise this model. And the common XSS prevention techniques are inadequate because (1) they are difficult to implement (there are several ways of encoding input) and (2) they limit creative ways of allowing user-generated input. This thesis has explored the various security techniques to mitigate attacks and vulnerabilities on web applications and proposed a new security model that takes into account the current weakness in JavaScript, Network Access Control and creation of mashups.

The overhead of the prototype implemented in JavaScript is minimal (8.3% on average). Since the security model is to be standardized and the final implementation native to the browser, this overhead will be lesser than the observed value. This is attributed to the fact that JavaScript is slower than native code.

The method to specify policies is simple (through the use of <meta> tags) but at the same time allows powerful fine-grained definition of policies. The policies encapsulate the capabilities and intentions of the Web Application and provide the browser a good context to enforce them. Leakage of confidential and unauthorized information to malicious sites is prevented by the Network Access Policy and the Script Policy ensures that malicious scripts are detected and prevented from causing harm.

The framework provided by IBM's SMash allows the creation of secure mashups. SMash gives Ajax apps a way to confirm the identity of each mashup component, and then decide whether to trust it. Exactly how this decision is made is left up to Ajax developers. SMash does not directly address cross-site scripting and other attacks. However by combining SMash with the security model described in this thesis it is possible to check the behavior of each mashup component against its security policy, thus detecting any malicious attacks and preventing them from causing harm.

Finally the proposed security model places the responsibility of security in the developer's hand and is transparent to the user. This is probably a right attempt because security is hard for a normal user and configuring options and rules is a very hard task. The developer on the other hand has full knowledge of the capabilities of his Web Application and hence creating of a security policy is an easy task.

Chapter 7 Future Work

This thesis proposes a few areas that can be interesting for Future Work. The new security model should undergo review by security experts (particularly the W3C) and be converted into a standard. For example the Access Control for JavaScript can be included in the ECMAScript standards which are constantly undergoing revisions by the W3C.

The security model should be implemented natively in browsers. The JavaScript prototype is sufficient to show the proof-of-concept but is not optimal since it slower and less secure than native implementation.

The policy distribution is one key area that could be of interest when extending this work. Policy distribution itself could be vulnerable to a wide range of attacks. A lot more effort and research needs to go into ensuring that the policy distribution can be made safe for deployment.

The proposed security policy is dependent on the creation of policy rules by the developer. If the proposal is to draw large scale adoption, there should be automated tools that could analyze Web Applications and build a set of security policies to ease the work for developers.

This work intentionally did not focus on usability since any specification that implies a particular user experience will be limiting. User experience itself is a broad topic and depends on various factors which all cannot be adequately addressed in a work on security. Thus another interesting extension of this work would be in the area of User Experience.

Chapter 8 Conclusion

In conclusion, the majority of attacks on web applications are due to the lack of proper context (a set of rules which the browser can understand and enforce) for the web browser. This thesis proposes a new security model for improved overall security, tighter control on JavaScript, Network Access and Cross-domain communication. This security model is backward compatible and allows the use of fine-grained security policies. The model works by allowing the developers of Web applications to define security policies which describe how the application is expected to function. These policies are understood by the Web Browser, which then enforces these rules when running the Web Application.

The formal specification of the security model was described and a prototype was developed for the Opera browser using the UserJS feature. The prototype provides safeguard against the most common XSS attacks and allows the web developers to define fine-grained security policies for their web applications. When combined with secure server-side web application security, a complete end-to-end security is achieved for web applications. The proof-of-concept implementation was found to be effective against most XSS attacks. **[38]**

In addition this thesis explores security specific to widgets (a special class of web applications) including cross-platform security of widgets. Sections of the security model proposal were applied to revise the security specifications for Opera Widgets (See Appendix A1). The results of BONDI project from OMTP **[52]** are very closely related to the security model proposed in this work and offers validation of the usefulness of having a standardized security model.

Ultimately no security model can guarantee full protection from vulnerabilities and attacks. There will always be hackers and enthusiastic programmers who will find security exploits and holes in any system. However, the art of software security is to make it harder for such attacks to be carried out and we hope that this proposed security model is another step on this path.

References

[1] Wikipedia.org (2008). *"Mashup (web application hybrid)"* [Electronic version]. Retrieved 21-04-2008, from http://en.wikipedia.org/wiki/Mashup_(web_application_hybrid)

[2] OpenAjax.org (2008). *"Ajax and Mashup Security"* [Electronic version]. Retrieved 21-04-2008, from http://www.openajax.org/whitepapers/Ajax%20and%20Mashup%20Security.php

[3] Wikipedia.org (2008). *"JavaScript"* [Electronic version]. Retrieved 21-04-2008, from http://en.wikipedia.org/wiki/JavaScript

[4] W3.org (2008). *"Document Object Model"* [Electronic version]. Retrieved 21-04-2008, from http://www.w3.org/DOM/

[5] W3.org (2008). *"Cascading Style Sheets"* [Electronic version]. Retrieved 21-04-2008, from http://www.w3.org/Style/CSS/

[6] W3.org (2008). *"The XMLHttpRequest Object"* [Electronic version]. Retrieved 21-04-2008, from http://www.w3.org/TR/XMLHttpRequest/

[7] JSON.org (2008). *"Introducing JSON"* [Electronic version]. Retrieved 21-04-2008, from http://www.json.org/

[8] Mozilla.org (2001). *"The Same Origin Policy"* [Electronic version]. Retrieved 21-04-2008, from http://www.mozilla.org/projects/security/components/same-origin.html

[9] JavaScript: The Complete Reference, second edition, by Thomas Powell and Fritz Schneider, McGraw-Hill/Osborne, ISBN: 0072253576)

[10] Opera.com (2008). *"Take Control with User JavaScript"* [Electronic version]. Retrieved 21-04-2008, from http://www.opera.com/support/tutorials/userjs/

[11] Net-Security.org (2006). *"Top 10 Ajax Security Holes and Driving Factors"* [Electronic version]. Retrieved 21-04-2008, from http://www.net-security.org/article.php?id=956

[12] Adobe.com (2005). *Adrian Ludwig -"Macromedia Flash Player 8 Security", Whitepaper, September 2005*. Retrieved 21-04-2008, from http://www.adobe.com/devnet/flashplayer/articles/flash_player_8_security.pdf

[13] Taossa.com (2008). *"Same-Origin Policy Part 1: Why we're stuck with things like XSS and XSRF/CSRF"* [Electronic version]. Retrieved 21-04-2008, from http://taossa.com/index.php/2007/02/08/same-origin-policy/

[14] Wikipedia.org (2008). *"Cross-site Scripting"* [Electronic version]. Retrieved 21-04-2008, from http://en.wikipedia.org/wiki/Cross-site_scripting

[15] WebAppSec.org (2008). *"Threat Classification"* [Electronic version]. Retrieved 21-04-2008, from http://www.webappsec.org/projects/threat/classes/cross-site_scripting.shtml

[16] WebAppSec.org (2008). *"DOM Based Cross Site Scripting or XSS of the Third Kind"* [Electronic version]. Retrieved 21-04-2008, from http://www.webappsec.org/projects/articles/071105.shtml

[17] CGISecurity.com (2003). *"The Cross Site Scripting (XSS) FAQ"* [Electronic version]. Retrieved 21-04-2008, from http://www.cgisecurity.com/articles/xss-faq.shtml

[18] OWASP.org (2008). *"Cross Site Scripting"* [Electronic version]. Retrieved 21-04-2008, from http://www.owasp.org/index.php/XSS

[19] CGISecurity.com (2003). *Jeremiah Grossman -"Cross-Site Tracing (XST)", Whitepaper, January 2003* [Electronic version]. Retrieved 21-04-2008, from http://www.cgisecurity.com/whitehat-mirror/WH-WhitePaper_XST_ebook.pdf

[20] OWASP.org (2007). *"Cross Frame Scripting"* [Electronic version]. Retrieved 21-04-2008, from http://www.owasp.org/index.php/Cross_Frame_Scripting

[21] PacketStormSecurity.org (2007). *Amit Klein – "Divide and Conquer - HTTP Response Splitting, Web Cache Poisoning Attacks, and Related Topics", Whitepaper, March 2004* [Electronic version]. Retrieved 21-04-2008, from http://www.packetstormsecurity.org/papers/general/whitepaper_httpresponse.pdf

[22] CGISecurity.com (2006). *Amit Klein – "HTTP Response Smuggling", 2006* [Electronic version]. Retrieved 21-04-2008, from http://www.cgisecurity.com/lib/http_response_smuggling.shtml

[23] CGISecurity.com (2008). *"The Cross-Site Request Forgery (CSRF/XSRF) FAQ"* [Electronic version]. Retrieved 21-04-2008, from http://www.cgisecurity.com/articles/csrf-faq.shtml

[24] Netcraft.com (2007). *"Google Fixes Gmail Cross-site Request Forgery Vulnerability"* [Electronic version]. Retrieved 21-04-2008, from http://news.netcraft.com/archives/2007/09/30/google_fixes_gmail_crosssite_request_forgery_vulnerability.html

[25] Shiflett.org (2007). *"My Amazon Anniversary"* [Electronic version]. Retrieved 21-04-2008, from http://shiflett.org/blog/2007/mar/my-amazon-anniversary

[26] Fortify.com (2007). *Brian Chess, Yekaterina Tsipenyuk O'Neil, Jacob West – "JavaScript Hijacking", Whitepaper, March 2007* [Electronic version]. Retrieved 21-04-2008, from http://www.fortify.com/servlet/downloads/public/JavaScript_Hijacking.pdf

[27] Anh Nguyen-Tuong, Salvatore Guarnieri, Doug Greene, Jeff Shirley, and David Evans. "Automatically hardening web applications using precise tainting". In Proceedings of the 20th IFIP International Information Security Conference, 2005.

[28] T. Jim, N. Swamy, and M. Hicks. "Defeating script injection attacks with browser-enforced embedded policies". In *WWW*, 2007.

[29] IETF.org (2006). *"RFC 4627 - The application/json Media Type for JavaScript Object Notation (JSON)"* [Electronic version]. Retrieved 21-04-2008, from http://www.ietf.org/rfc/rfc4627.txt

[30] Mozilla.org (2007). "*Signed Scripts in Mozilla"* [Electronic version]. Retrieved 21-04-2008, from http://www.mozilla.org/projects/security/components/signed-scripts.html

[31] SecurityTeacher.com (2009). *"Security in Opera Browser: A Brief Glance"* [Electronic version]. Retrieved 05-05-2009, from http://www.securityteacher.com/2006/02/18/security-in-opera-browser-a-brief-glance/

[32] David Endler. "The Evolution of Cross Site Scripting Attacks. Technical report", iDEFENSE Labs, 2002.

[33] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic. "Noxes: A client-side solution for mitigating cross-site scripting attacks". 2006.

[34] Úlfar Erlingsson, Benjamin Livshits, and Yinglian Xie. "End-to-end Web Application Security". In Proceedings of the 11th Workshop on Hot Topics in Operating Systems (HotOS'07), San Diego, CA, May 2007.

[35] JSON.org (2006). *Douglas Crockford -"The <module Tag> - A Proposed Solution to the Mashup Security Problem"* [Electronic version]. Retrieved 21-04-2008, from http://www.json.org/module.html

[36] W3.org (2008). "*HTML 5 – Cross-document Messaging"* [Electronic version]. Retrieved 21-04-2008, from http://www.w3.org/html/wg/html5/#crossDocumentMessages

[37] W3.org (2008). *"Access Control for Cross-site Requests"* [Electronic version]. Retrieved 21-04-2008, from http://www.w3.org/TR/access-control/

[38] Ha.ckers.org (2008). *XSS (Cross Site Scripting) Cheat Sheet* [Electronic version]. Retrieved 24-04-2008, from http://ha.ckers.org/xss.html

[39] Jerome Saltzer and Michael Schroeder. "The Protection of Information in Computer Systems". In Proceedings of the 4th Symposium on Operating System Principles, ACM Press, 1973.

[40] W3.org (2008). *" Web Security Experience, Indicators and Trust: Scope and Use Cases"* [Electronic version]. Retrieved 25-04-2008, from http://www.w3.org/TR/wsc-usecases/

[41] Simson L. Garfinkel, "Design Principles and Patterns for Computer Systems that are Simultaneously Secure and Usable". PhD Dissertation, 2005.

[42] FAQS.org. *"RFC 1738 - Uniform Resource Locators (URL)"* [Electronic version]. Retrieved 01-05-2008, from http://www.faqs.org/rfcs/rfc1738.html

[43] Opera.com (2008). *"Take Control with User JavaScript"* [Electronic version]. Retrieved 04-05-2008, from http://www.opera.com/support/tutorials/userjs/

[44] Wikipedia.org (2008). *"Bookmarklet"* [Electronic version]. Retrieved 04-05-2008, from http://en.wikipedia.org/wiki/Bookmarklet

[45] OperaWatch.com(2008). *"What percentage of known security bugs in browsers are actually fixed?"* [Electronic version]. Retrieved 27-05-2008, from http://operawatch.com/news/2007/01/what-percentage-of-known-security-bugs-in-browsers-are-actually-fixed.html

[46] LibreSoft.es(2008). *"MobileScript"* [Electronic version]. Retrieved 04-06-2008, from http://libresoft.es/Projects/Mobilescript/index_html

[47] Macromedia.com (2008). *Adobe Systems – "Adobe AIR Security", Whitepaper, December 2007* [Electronic version]. Retrieved 05-06-2008, from http://download.macromedia.com/pub/labs/air/air_security.pdf

[48] CGISecurity.com (2001). *Steve Pettit –"Anatomy of a Web Application: Security Considerations", Whitepaper, July 2001* [Electronic version]. Retrieved 21-04-2008, from http://www.cgisecurity.com/lib/Web_Server.pdf

[49] Taossa.com (2008). *"Same-Origin Policy Part 2: Server-Provided Policies?"* [Electronic version]. Retrieved 12-06-2008, from http://taossa.com/index.php/2007/02/17/same-origin-proposal/

[50] MobileMonday.org.uk (2008). *David Pollington - "Web Runtimes – evolving beyond the browser"* [Electronic version]. Retrieved 12-06-2008, from http://mobilemonday.org.uk/Vodafone%20MoMo%20Feb%204.ppt

[51] Frederik De Keukelaere, Sumeer Bhola, Michael Steiner, Suresh Chari, Sachiko Yoshihama. "SMash: Secure Cross-Domain Mashups on Unmodified Browsers". Research Report 2007.

[52] Bondi.omtp.org (2009). *BONDI Architecture and Security Document* [Electronic version]. Retrieved 05-05-2009, from http://bondi.omtp.org/Documents/CR10/BONDI_Architecture_Security_Task_CR10.pdf

[53] Omtp.org (2009). Open Mobile Terminal Platform. Retrieved 05-05-2009, from http://www.omtp.org

Appendix

A.1 Revised Opera Widgets Security Specification

Widget lifecycle and identity

Latent widgets are identified by URLs that reference zip files on hosts on the Internet. (They can also be identified by a path in the file system to a config.xml file.) A widget is loaded into a widget runner (Opera) by loading the URL of the zip or config.xml file.

Once the zip file is loaded, a running widget instance will normally be created. How this happens is specified in the section on provisioning

The widget runner may be suspended, and running widget instances are suspended with it. Only when the user explicitly kills the instance are it's data deleted.

The widget instance is unique in the universe. There is no notion at all of two widget instances being "the same". No information in the config.xml file or in the URL from which the zip file is loaded has any bearing on widget identity.

Browser state

Separate widget instances share no information at all. Specifically:
A cookie set by a widget instance, or by a URL loaded by a widget (eg through XmlHttpRequest) is visible only to that widget instance, never to any other instances or to documents loaded into the browser in any other way.
If a URL loaded by a widget requires HTTP authentication then authentication must be performed on behalf of that widget instance; the authentication is not shared with other widget instances or with URLs loaded into the browser in any other way.
A set of settings for a widget instance is shared with no other widget instances.
Other persistent storage mechanisms, such as those defined in HTML must not share data with other widget instances, or with the storage context in the web browser.
Cache files or cache indexes are not shared with the web browser, or with any other widget instance
Widget cache data should not be indexed in Opera's internal history search engine.

Preferences

There shall be a preference to allow widget support to be turned off completely.

For the security settings it is a little more complicated. Opera has three levels of preferences: system fixed, user settings, and global defaults. These are consulted in that order; if a preference is found in one then the others are not consulted.

All three preference levels may contain global security settings as defined below for widgets.xml. They are merged according to the rules defined in the section "Merging several widgets.xml files".

Network access rights model

History

The previous model described URL access as accessing a resource in terms of using the DOM3 Load and Save APIs or XMLHttpRequest.

The previous model was also described as a partial opt-out model.

Scope
The access rights model determines whether a widget should be allowed to access any particular URL, where "access" means the following:
Access through opening a URL in an external resource, such as the browser, by means of a link with an appropriate target, a call to widget.openURL or similar.
Access through loading an embedded document or object inside the widget, such as an image, iframe, object, svg:foreignObject and similar.
Access to resources happening from a child window in the widget. This includes, but is not limited to resources such as images, frames, objects, embeds and similar contained in a document that is embedded in the widget, and displayed with Opera as a viewer. This relates to the previous point in a "turtles all the way down"-manner in such that all URL access from a widget instance must be checked.
Access to the URL through API calls, such as DOM3 Load/Save, XMLHttpRequest, or any other current or future mechanism.

The rules for accessing the network are determined based on the following components:
The widget's declaration of network intent.
Opera's declaration of global allowable network policy, either through fixed configuration or through profile configuration.
(Opera's declaration of allowable network policy for an individual widget).

Network classes
This specification describes two classes of networks, a "private" network, and a public network.

A private network, or local network is by default defined as the user's local machine, including any IP address that resolves to the local machine. Further, the IP ranges as defined by [RFC 1918], are considered to be private. These addresses are primarily being used in systems set up behind a NAT translation device, and provides machines with unique addresses where there is only one public IP address for several machines. These addresses are:

10.0.0.0 - 10.255.255.255 (10/8 prefix)
172.16.0.0 - 172.31.255.255 (172.16/12 prefix)
192.168.0.0 - 192.168.255.255 (192.168/16 prefix)

In addition, when a user is within an ad-hoc network, networking equipment (including software components in operating systems), typically use the IPv4 Link-Local addresses as defined by [RFC 3927], which is also considered to be part of the local network.

169.254.0.0 - 169.254.255.255 (169.254/16 prefix)

Opera must provide a default configuration wherein the four network ranges described above are defined to fall within the private IP range. This policy should be editable by the

user, meaning it should be possible to add or remove items to the list of private networks and hosts.

Widget network access configuration through config.xml

When a widget wants to request access to resources, it does so by adding a security fragment to the config.xml element, as specified in the Figure below:

Figure A-24: <security> element of the config.xml file

```
element security optional
  element access multiple
    element "protocol" multiple
      cdata
    element "host" multiple
      cdata
    element "port" multiple
      cdata
    element "path" multiple
      cdata
  element "content"
    attribute "plugin" value = "yes|no"
```

When a widget does not specify the access element, it is assumed to be present, being an exact match to what is specified in the default access configuration, meaning the widget can access any resource permitted by configuration. The moment the security element is present, it exclusively allows access to the resources specified within the access elements provided, if such access is granted by the stated security policy.

Security policy file, widgets.xml

The network policy for a widget is governed by the file widgets.xml, and describes which hosts and IP ranges belong to the private network, which hosts are allowable, and a blacklist containing network rules where access should always be denied.

Note that while this XML policy file primarily governs network access, it also specifies access to extended APIs and plugin technologies (including Java).

The syntax for the policy file is described in the Figure below:

Figure A-25: The widgets.xml policy file

```
element "widgets"
  element "access" multiple
    element "protocol" multiple
      cdata
    element "host" multiple
      cdata
    element "port" multiple
      cdata
    element "path" multiple
      cdata
  element "content"
    attribute "file" value = "none|restricted|unrestricted"
    attribute "webserver" value = "yes|no"
    attribute "plugin" value = "yes|no"
  element "private-network"
    attribute allow value="none|restricted|unrestricted"
    element "protocol" multiple
      cdata
    element "host" multiple
      cdata
    element "port" multiple
      cdata
    element "path" multiple
      cdata
  element "blacklist"
    element exclude
      element "protocol" multiple
        cdata
      element "host" multiple
        cdata
      element "port" multiple
        cdata
      element "path" multiple
        cdata
    element include
      element "protocol" multiple
        cdata
      element "host" multiple
        cdata
      element "port" multiple
        cdata
      element "path" multiple
        cdata
```

Default widgets.xml

Below is a proposed default security configuration file

Figure A-26: <security> element of the config.xml file

```
<widgets>
  <security>
    <access>
      <protocol>http</protocol>
      <protocol>https</protocol>
    </access>
    <content file="restricted" webserver="yes" plugin="yes" />
    <private-network allow="unrestricted">
      <host type="localhost" />
      <host type="range">10.0.0.1-10.255.255.255</host>
      <host type="range">172.16.0.0-172.31.255.0.0</host>
      <host type="range">192.168.0.1-192.168.0.255</host>
      <host type="range">169.254.0.0-169.254.255.255</host>
    </private-network>
  </security>
</widgets>
```

The access element
The access element is used to specify the access permissions of the widget. The element. If the contents of the access element is empty, then a widget would not be granted access. With the exception of the 'protocol' element, the absence of an element means the same as "match all", granting access to all resources not limiting access.

The content element
The content element is used to specify access to additional APIs and technologies. When the element, or one of its specified attributes is absent from the widgets.xml file, this is interpreted as "deny". Valid attributes for the content element are:
file
The file attribute is used to determine whether a widget should have access to APIs that work with any direct File I/O operations. Possible values for the attribute is "none", "restricted" or "unrestricted". When the value is "none", no direct File I/O is permitted. When the value of the attribute is set to "restricted", only operations that are a result of user interaction, such as browsing to a folder and accepting it in a folder chooser dialog is permitted, while "unrestricted" would allow access to file i/o operations without user interaction. If the attribute is not present, it should be interpreted as being present, with the value "none"
plugins
The plugins attribute is used to determine whether widgets should have access to embed objects viewed with external viewers and plugin engines, including java.

The private-network element
The private-network element is used to define which protocols, hosts, ports and IP ranges define the user's private network. As children, this element defines the same elements as for access.

The element also defines an attribute allow, used to specify which kinds of network access the widget is allowed. The attribute accepts one of three values:
none
Disallow access to all resources defined as being in the private network
restricted
Allow a widget to access either the private network, or the public network, but do not allow a widget to mix these modes.
unrestricted
Allow access to both the private network and the public network without restrictions.

The blacklist element
The blacklist element specifies two child elements "exclude", used to define combinations of host, port, path and protocols that cannot, under any circumstance be accessed, and an "include" element used to individually override restrictions set in place by "exclude".
The 'host' element
The host element is a syntactical construct used to determine which hosts a widget can contact. The element is defined as such

```
element host
  attribute type optional value="localhost|string|range"
  cdata
```

The type attribute determines which kind of matching is done for the host element:
localhost
When the value of the type attribute is specified as 'localhost', this refers to accessing the user's local machine, regardless of which method of access is chosen -- hostname, or one of the IP addresses that resolves to the user's local machine. In the case that localhost is specified, the cdata in the element is disregarded for comparison purposes
string
In the case the value of the type attribute is 'string', the cdata within the element is regarded as a string match for the hostname, including an optional leading wildcard.
range
When the value of the type attribute is 'range', the contents inside the element is expected to be an IP address, or a range of IP addresses separated by a single '-'.

If the 'type' attribute is omitted from any declaration, it is assumed to be present, with the value 'string'

The semantics of the 'host' element means slightly different things in different contexts:
In the 'access' element of widgets.xml, or in config.xml
When a 'host' element is missing from an access element, regardless of in which file this element occurs, the element is assumed to be present, as a single wildcard value:

<host>*</host>. When present, the default wildcard element is replaced with the exact value of any matching host elements.
In the 'private-network', 'exclude' and 'include' elements
In these elements, the host attribute is not assumed to be present if it is missing from the list, and so, any hosts not explicitly included in these element will not go into the definitions of the private network, the blacklist or its overrides.

The 'protocol' element
The protocol element is used to determine which protocols a widget should have access to. When missing from config.xml, the element is assumed to be present three times, with the values 'widget', 'http' and 'https'. When missing from any other context, the element gets no value, and for instance if missing from 'access', a widget would not be allowed to get network access.

The 'port' element
The port element is used to define which port numbers a widget should have access to when initiating network traffic. The port element accepts as its value a single port number, port numbers separated by a comma, or a range of ports separated using '-'.

When missing from an 'access' element, the port element is assumed to be present, matching all ports, and when a widget requests access to a resource using http, it can do so on every port. It should however be noted that there are certain port numbers for which Opera has disabled access for security reasons, and a configuration override cannot change these.

When a different port number is specified for a protocol in an access element, the port numbers specified are the only by which a widget can communicate.

The 'path' element
The path element is used to restrict widget access to particular paths within the rule sets. When missing from anywhere within widget configuration, or policy file, the element is assumed to be present, allowing access to any path, thus implicitly bearing the value "/".

The path is treated as a wildcard path, allowing access to any matching subpath of the element. Example:

In this access fragment, placed within widgets.xml or config.xml, access to any resource contained within the "/cats" subpath is allowed:
<path>/cats</path>

thus, the widget can access all of
/cats/siamese.html
/cats/
/catsoup

In the case that the author of either the widget.xml fragment, or config.xml fragment meant to limit access to '/cats/', the specification of the path would have to include the forward slash:
<path>/cats/</path>

Enabling network access
A widget must declare its intention with regard to API and network access through the config.xml file, using the syntax described in the below config.xml fragment:
element widget
attribute "network" optional value = "public|&private"

To acquire access to the network, the widget must thus declare this by adding the network attribute to the widget element in the widget's config.xml file. The value accepted for this attribute is a list of DOMStrings that can specify which network class it intends to access. The values are case sensitive. Valid values are:
private
Governs any network the user has classified as a private network in configuration.
public
Governs any network NOT classified as private

When a widget starts, Opera must read this configuration, and compare it to Opera's own configuration of network access in the widgets.xml file.

Examples:

Only request access to a user's intranet or private/trusted network
<widget network="private">

Only request access to what the user considers to a public/non-trusted network
<widget network="public">

Request unrestricted access:
<widget network="private public">

Do not request network access:
<widget>

Whether the widget is granted network access based on these criteria, is described in the Network access rights section.
URL access rights
Scope

The term 'URL access' specifies any access to a networked resource a widget may attempt. This includes:
The widget directly requesting a resource through inclusion (such as iframes, objects and other elements)

The widget including images, stylesheets, and scripts. Including indirectly, when a stylesheet or (SVG) image requests additional resources to be loaded.
The widget attempting to open a resource in an external application (in effect, using links or widget.openURL)
Resources contacted by API access: DOM3 Load/Save, XMLHttpRequest and similar
The destination of form elements
When Opera includes resources in iframes and similar, any access made by documents contained in these included documents.

Whenever Opera attempts to contact any resource, a number of checks have to be made to see if access to the URL is permitted.

Failure modes
If a widget, by configuration, excplicitly requires access to a resource forbidden by other configuration, installation of the widget should fail.

If a widget attempts to contact a resource that is in conflict with the widget's own stated security policy, or the widget attempts to contact a resource which it has been restricted from accessing (such as accessing a private network resource when it has only requested access to public network resources), it should fail in one of two ways:
If the access happens through direct access, such as opening URLs using link constructs, form submissions, or similar methods, the action should fail silently, but Opera may post a notice in the Error console.
If the access happens through API access, such as XMLHttpRequest, the action should generally fail with a security violation error in the same manner a web page that tries to do cross-domain requests would.

Private network class
The algorithm for allowing access to the private network is as such:
If the widget has added 'private' to the network attribute, continue. Otherwise, deny access.
If a system fixed configuration allows access, continue. Otherwise, deny access.
If the profile configuration, allows access, continue. Otherwise, deny access.
If a per-widget override configuration allows access, continue. Otherwise, deny access.

Public network class
The algorithm for accessing the public network mirror the algorithm for the private network.

Multiple network classes
In the case that a widget requests access to both network classes, the algorithm for determining private networks and public networks are checked individually. In addition, the following algorithm is used.
If the widget requires access to both network classes, continue.
If a system fixed configuration allows simultaneous access to both network classes, continue. Otherwise deny access.

If a profile configuration directive allows simultaneous access to both network classes, continue. Otherwise, deny access
If a per-widget override configuration allows access, continue. Otherwise, deny access.

In the case that any step in the above processes has resulted in "Deny access", deny access. Otherwise allow access.

URL-based access check
When a widget attempts to contact a resource, and this resource has been allowed by the network class above, apply the following algorithm:
Take the request URL string, make any comparison based on the matching algorithm defined below
Compare the URL components to the access element of the widget's config.xml. If no such directive exists, assume one is present with the same values as in the corresponding section in widgets.xml. If the request URL is permitted by this step (meaning, the URL matches the directive), continue to the next step. If the request URL does not match the directive, deny access.
Compare the URL components to the blacklist. If the request URL does not match a blacklist exclude directive, allow access. If the request URL matches a blacklist exclude directive, continue to the next step.
If there is no blacklist include directive, deny access. If there is such a directive present, check if the request URL matches the directive. If there is a match, allow access to the URL, otherwise deny.

URL comparison
When comparing a URL for match with any given directive, follow the following algorithm.
Componentize the request URL into the following components: protocol, host, port and path.
Compare the request protocol on a character by character basis with the corresponding protocol element. If the directive is in an implicit access directive (meaning it's not physically present in any document), assume it has the values http and https, respectively. If there is a match, continue. Otherwise, mark the URL as not matching.
Reverse the two strings given for the request host and the host specified for the directive (directive host). Do a case-insensitive character by character comparison of the strings. If a mismatch is found before the end of the directive host string is reached, and the last two characters in the directive host string are not the character sequence '.*', consider the request host to not be a match. If there are characters left to parse in the request host, and the last characters of the directive host were the wildcard sequence '.*' consider the host a match.
Compare the request port and the directive port. If the port number is missing from the request URL, consider the port to be the default port for the given protocol. If there is no port number present in the directive, consider the directive port to match all port numbers. Make a numeric match between the request and the directive. If the request port is in the range defined by the directive, consider it a match.
For the request path, and the directive path perform a case sensitive character-by-character match. If there at any point is a mismatch, consider the path as "not matching".

If there are no characters left to parse in the request path, and there are characters left to parse in the directive path, consider the paths as "not matching". If there are no characters left to parse in the directive path, and there are more characters left to parse in the request path, consider the path a match. If there is no path specified in the directive, always consider the path a match.,

Form and Links Behaviour

History

With the current security model, form submission and links poses the following issues:
Form submission can circumvent the internet/intranet restrictions currently in place.
Form submission without a target is problematic.
Links are not checked for security violations.

New behaviour

The new security model defines the following behaviour for forms and links; thus ensuring forms and links do not violate security policy:
A form element should have a valid target.
If the form has a reserved target, and this target leads to intrisically replacing the topmost document in the widget, in effect replacing the widget, submitting the form should fail silently.
If the form uses the '_blank' target for a GET request, the form should be submitted to an external application (in effect, Opera's main browser instance.
POST requests that result in submission to an external viewer SHOULD fail.
The URL provided by the form's action attribute MUST be permitted by the computed security policy for the widget.
Links must have default "_blank" target, and open in an external viewer. The URL referenced MUST be permitted by the computed security policy for the widget.

Embedded Objects Security

History

The current model does not define clearly the behavior of embedded objects (iframe, img, object, frame, etc.) displayed in the widgets.

Embedded object security context

The new security model defines the security context and behaviour for embedded objects:
Objects must adhere to widget security policy.
Objects must not cross network boundaries when loaded.
Objects may share cookies and cache with widgets.
Objects must not share cookies and cache with browser.
Objects must not be aware of widget's existence.
There should be no reference to window.opener (the widget shouldnt be visible to the object)
window.top should point to the embedding frame only.
Widgets may have a one-way reference to inject scripts, read DOM, etc. The reference must be strictly one way and initiated from the widget only.

Extended API access
In general: All extended API access has to be declared, and can be overridden by configuration. Some known components are listed here, the list however is not exhaustive, and may be extended at any stage.

File I/O
To gain access to file stream operations on the file system, the a widget author can specify the 'file' attribute on the widget element of config.xml. Valid values for the file attribute are "yes" and "no", respectively. If the attribute is missing, it is assumed to be present with the implied value 'no'.

config.xml fragment
  element widget
    attribute webserver optional value = "yes|no"
    attribute file optional value = "yes|no"

The widget: URL protocol
A widget:// protocol URL is defined as below
  widget_protocol_url = "widget://" [widget_identifier] "/" [path] ["?" query] ["#" fragment];
  widget_identifier = [a-z0-9]+

The path component is optional, and follows the the definition given for the path component in section 3.3 of rfc3986. Likewise, query strings and fragments are permitted, as defined by sections 3.4 and 3.5 of same document. The path always refers to the root of the widget folder, and traversing beyond this is not allowed.

Example: If we have a widget with the identifier decafbad, and we want to reference a file 'caffeine.xml', placed in the widget's root folder, for inclusion, we would refer to this url: widget://decafbad/caffeine.xml

The widget_identifier SHOULD be unique, and semi-randomly generated. Two different widget installations on a system MUST NOT share the same widget identifier

Access rules
Any access to URL's in the widget:// url space is bound by a strict same-origin policy, meaning that a resource defined by a widget: URL cannot be accessed in another context than the widget wherein the widget is contained.

External resources loaded in the widget, through iframes, svg:foreignObject and similar mechanisms MUST NOT be allowed to access resources[1] inside the widget, or to utilize the widget:// url space

If a widget embeds a file from within the widget, by means of frames or similar mechanisms, the embedded documents MUST be running in the context of the widget, and will be subject to the same security restrictions as the widget.

It SHOULD NOT be possible to launch one widget from within another widget.

It MUST NOT be possible to access the widget: URI space from other browsing contexts than inside the widget, or a window that can be determined to be completely trusted and running with extended security privileges. This means that a strict same-origin policy exists between different widgets.

Objects included in the widget from a source outside the widget cannot access documents using the widget:// protocol. This means that if a widget includes an iframe to http://example.com/, the document on example.com is not able to access URL's using the widget:// protocol.

Configuration Settings and UI Interaction

Scope

This section covers remaining topics not covered above. This includes configuration settings and UI interaction.

Desired UI interaction

UI interaction is critical to the success of the security model. The user should be able to clearly recognize whether he grants priviliges to a widget or to all widgets, for a session or permanently, for a subset of URLs or all URLs. The following UI issues must be considered during implementation:

An editable global configuration must be defined.
Per-widget configuration may be defined if necessary.
Security information view may be displayed for the widget.
Widgets must be installed by explicit user confirmation.
Every widget upgrade must be confirmed.
The model must disallow installation of widgets that violate security policies.
The user must be warned when he downloads widgets from an untrusted URL and he MUST have an option of adding the URL to the whitelist.

Integrity of the configuration

The access model assumes that the configuration expresses the wishes of the user. But the configuration may be compromised in various ways. Two notable attacks involve user manipulation (making the user change the configuration) and direct manipulation of the configuration.

The main avenues to avoiding user manipulation are a good user interface (see separate section below) and good default rules that makes user interaction generally unnecessary.

The configuration (meaning, widgets.xml plus the security sections of all the widgets) may be directly manipulated by other software, by spyware, and by intruders. (These are basically all the same thing.)

In order to safeguard the configuration it must at a minimum be checksummed, and the checksum must be stored relatively safely.

Checksumming implies that all widgets are installed through UI actions in Opera (since this forces the user into the loop when recomputing the checksum).

The checksum is probably safe if it is stored in the same manner as wand passwords (since nobody is making a fuss about the secrecy of wand passwords).

Checksumming can be performed across all widgets or per widget (and individually for the relevant parts of opera.ini). Somehow it seems more reasonable that it's done per widget: this allows compromised widgets to be flagged, reinstalled, or removed; allows new widgets to be installed quickly; and leaves correct information alone when a new widget is installed.

A compromised widget is refused access to any URL, and a compromised settings file is assumed to refuse any widget access to any URL, until the user manually rebuilds the checksum database.

A.2 Security Model Prototype Implementation

This appendix covers the implementation of prototypes for the security model proposed in Chapter 4. The prototype is limited but is sufficient to validate the usefulness of each policy outlined in Chapter 4. The source code is written in JavaScript relying on the very useful UserJS feature provided by the Opera browser. This was done with the intention of making as much of the work public as possible.

As we saw in the earlier chapters JavaScript opens up scope for exploits because the scripts in a page are automatically trusted by the browser. Hence any evil script injected is blindly executed by the browser. JavaScript Access Control attempts to fix this by giving the browser more information about the scripts included in a web application so that the browser has a better context to decide whether to allow or block the script.

Opera User JavaScript
The User JavaScript feature in the Opera browser lets users write custom JavaScript files for specific pages which are then used by the browser at runtime. **[43]**

User JavaScript is useful in the following cases (not an exhaustive list, the usefulness is left to the creativity of the user):

- Fixing scripts that break Web pages on the Opera Browser
- Adding custom content or modifying content on Web pages
- Controlling scripts on a page and the capabilities of each script
- Simplifying bookmarklets **[44]**

Opera treats the User JavaScript's as a script on the page visited by the user. However the User JavaScript runs before the first script and before the DOM creation has finished, thus allowing user's to control page scripts and content on the page.

User JavaScript is a superset of normal JavaScript. It allows everything that is possible in normal scripts in addition to a few extra functions like intercepting events, overriding variables and functions, events related to script loading, etc. This makes it an ideal candidate for implementing the JavaScript Access Control policies.

Design

Using User JavaScript it is possible to write JavaScript code to implement some of the JavaScript Access Control policies discussed in Chapter 4.1. UserJS provides events "BeforeScript", "BeforeJavascriptURL" and "BeforeExternalScript" which allow us to examine scripts before they execute and also the possibility to block scripts if desired. For those unfamiliar with the different types of scripts, Javascript URLs are scripts defined by the "javascript:" protocol. Internal scripts are defined using the <script> tag and external scripts are defined using the <script> tag with a "src" attribute pointing to a valid JavaScript file located at a given URL. Opera ensures that UserJS code will run before the scripts in the document are allowed to run. Thus the policies can be successfully enforced. The policies are parsed from <meta> tag and loaded into a local custom policy object. When the above mentioned script events occur, the script is checked against the policies and either allowed to run or blocked according to whether they conform to the policies or violate them. Sandboxing is implemented by blocking scripts defined within tags with a class attribute "ScriptPolicy:sandbox" (ie. scripts whose parentNode is a tag with class attribute equal to "ScriptPolicy:sandbox"). The object access control is implemented by using the "DOMNodeInserted" event. Such an event occurs when the DOM tree of the document changes due to addition of an extra node. This typically happens when nodes are added dynamically to the page or through script injections. When such node additions occur, they are check against the Object Policy specified and only if the policy is adhered to, the node addition is allowed. Otherwise the node is prevented from being added to the DOM tree. The host pattern matching is done by the use of Regular Expressions. Access to cookie and history objects is restricted by overriding these objects in the UserJS. When an access is made to these objects, the policy is checked and based on that access is either granted or rejected.

Figure A-27: JavaScript Access Policy Implementation Flowchart

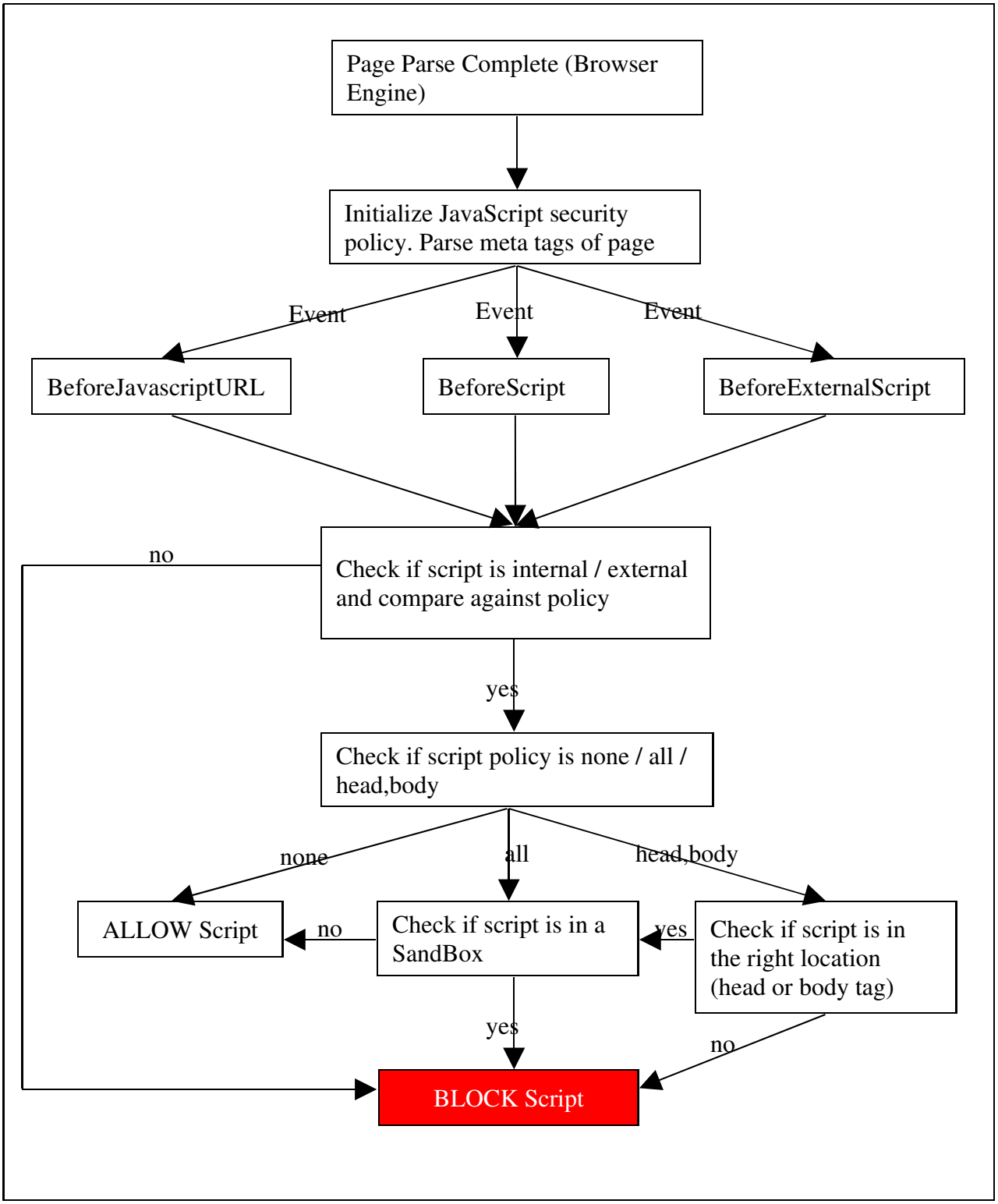


Figure A-28: JavaScript SandBox Implementation Flowchart