RUHR
UNIVERSITÄT
BOCHUM

RUB

RUHR-UNIVERSITÄT BOCHUM

autoCSP - CSP-injecting reverse HTTP proxy

Nicolas Golubovic

Bachelor Thesis. November 6, 2013.
Chair for Network and Data Security – Prof. Dr. Jörg Schwenk
Advisor: Dr.-Ing. Mario Heiderich
Nonacademic Advisor: Dipl-Ing. Felix Gröbert

Declaration

I hereby declare that this submission is my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by an- other person nor material which to a substantial extent has been accepted for the award of any other degree or diploma of the university or other institute of higher learning, except where due acknowledgment has been made in the text.

Erklärung

Ich erkläre, dass das Thema dieser Arbeit nicht identisch ist mit dem Thema einer von mir bereits für ein anderes Examen eingereichten Arbeit. Ich erkläre weiterhin, dass ich die Arbeit nicht bereits an einer anderen Hochschule zur Erlangung eines akademischen Grades eingereicht habe.

Ich versichere, dass ich die Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Die Stellen der Arbeit, die anderen Werken dem Wortlaut oder dem Sinn nach entnommen sind, habe ich unter Angabe der Quellen der Entlehnung kenntlich gemacht. Dies gilt sinngemäß auch für gelieferte Zeichnungen, Skizzen und bildliche Darstellungen und dergleichen.

Ort, Datum	Unterschrift

Acknowledgments

First and foremost I would like to thank Felix Gröbert of Google Inc. for proposing this thesis topic and giving invaluable feedback to my ideas. Furthermore, I thank Google, for allowing him to do so and thus supporting my thesis. Without his initiative and effort this thesis would not exist.

Special thanks to my advisor Mario Heiderich, who provided extremely helpful advice regarding both the topic and scientific writing.

Abstract

As the Internet gains wide adoption, the World Wide Web becomes one of the most important sources of information worldwide. Highly distributed networks serve millions of clients every day. In consequence of this popularity, user agents increased in complexity over the years, while still maintaining the technological burdens of their early days. Core security issues were left untouched by browser vendors and standards bodies. This led to a wide range of possible client-side attacks today. By default, there are only a few limitations to the capabilities of web documents. Accordingly, user agents shift the responsibility to protect from client-side attacks to web applications.

In an attempt to give web applications the possibility of limiting these capabilities, Sterne et al. proposed Content Security Policy in 2010. It enables fine grained policies, strictly controlling the allowed resources and other properties of web documents. As a result of that, it is able to reduce the attack surface of a web application in a substantial way. This comes at the cost of breaking compatibility with all websites using inline code and *eval*-like constructs. In order to use secure policies, these have to be rewritten. Therefore, deploying CSP may not always be feasible. Furthermore, policy generation introduces a lot of pitfalls because it leaves the decision, which resources (or resource types) might harm client-side security, to the web application. Thus, it is easily possible to create too broad policies, not increasing the security at all.

This thesis presents a reverse HTTP proxy, which is able to infer restrictive CSPs for arbitrary web applications. It externalizes inline code automatically in a secure way and attempts to replace calls to *eval*-like constructs. Overall, it allows web applications to adopt CSP without any changes to server-side code.

Contents

List of Figures

List of Tables

1. Introduction

Global Internet usage has continuously grown in the past years – a trend estimated to carry on in the near future[1]. Simultaneously, the World Wide Web became a popular medium for information exchange and communication. Hence, browsers, being the main mediator between users and web applications, greatly increased in complexity over the time. In the early days of the web, a browser was merely capable of displaying static content, using simple protocols and languages [1]. Modern browsers offer, for example, real time communication[2] and 3D graphics rendering [2]. While the web moved forwards in both usage and technology, the core security principles persisted. Reis et al. depict this as a fundamental problem of the web platform [3]. Security boundaries are blurred by exceptions and browser quirks. Furthermore, the simple protocols used in the web offer no possibility of separating structure from content. In the course of the last years, compatibility was chosen over security and hence these core issues still exist. Therefore, web security has become a patchwork of various mechanisms and techniques, implemented both on the client- and server-side. However, the ultimate responsibility to protect from attacks lies in the web application. If the web application is vulnerable to content injection attacks, such as Cross-site scripting (XSS) (cf. Section 3.9.2), the browser does little to prevent data leakage by default. This is especially troubling, considering the amount of private data stored in the web today. According to the Open Web Application Security Project (OWASP), XSS still is the third most frequent vulnerability in the web [4].

Defense in depth is an approach to prevent attacks through multiple layers of security mechanisms [5]. It assumes that each mechanism on its own could potentially be bypassed, leaving the web application unprotected. Thus, many different layers are employed at the same time to achieve a good overall security. Content Security Policy (CSP) aims to be one of these layers [6]. It allows web applications to limit the allowed resources and properties of all served documents – a practice known as capability control. A strict policy is a whitelist of allowed resources, none of which should be controllable by an attacker. Additionally, inline code needs to be disallowed because its origin cannot be reliably determined by browsers. Another risk can be mitigated by disabling *eval*-like constructs with CSP. This tightens the security of a web application tremendously, in contrast to the few limitations enforced on web documents by default. Attackers, not limited by CSP or comparable mechanisms, may extract arbitrary information with a client-side code injection. On the contrary, if such a policy is enforced, it first must be bypassed to leak data. Based on the strictness of the whitelist, this might prove difficult.

Deploying CSP may require huge effort, as secure policies disallow inline code and *eval*-like constructs. Occurrences of these code patterns must be rewritten, which may not always be possible. Some (very few)

[1] Wikipedia, *Global Internet usage*, `http://en.wikipedia.org/wiki/Global_Internet_usage`, Oct. 2013

[2] A. Bergkvist, D. C. Burnett, C. Jennings and A. Narayanan, *WebRTC 1.0: Real-time Communication Between Browsers - W3C Editor's Draft*, `http://dev.w3.org/2011/webrtc/editor/webrtc.html`, Oct. 2013

use cases actually require dynamic code building as offered by *eval*. Additionally, the amount of work which has to be carried out to rewrite the web application, might not always be feasible. Finally, even if a web application can be protected by a CSP, generating a policy introduces many pitfalls. Too broad policies might be easily bypassed, while too strict policies may break web application functionality or usability. For instance, a blog might be using a strict CSP to protect from data leakage. Each new post may not contain external images, unless it is hosted on a whitelisted location or the policy is extended. Compared to the little work required without CSP, this reduces the usability of the blog software substantially.

In order to make the widespread use of CSP possible, a solution to these problems must be found. This thesis proposes a reverse HTTP proxy, capable of inferring strict policies and externalizing inline code automatically. These policies can be enforced to obtain the protection offered by CSP, while preserving website functionality in many cases. The proxy is application agnostic and thus able to protect any web application without any changes to server-side code. In addition, this thesis evaluates the advantages and downsides of the chosen approach, highlighting the problems of making documents compatible with strict CSPs.

Outlook

This thesis is composed of seven chapters and one appendix. After the introduction, important research, which is related to this thesis, is presented in Chapter 2. Then, the fundamental knowledge, needed to understand this thesis, is briefly explained in Chapter 3. CSP is the core mechanism used by the proxy. Therefore, Chapter 4 describes it in detail. In Chapter 5, the proxy is introduced with details on its policy generation and code externalization. An evaluation of it is given in Chapter 6. Based on this, the conclusion elaborates on the advantages and downsides of this approach in Chapter 7.

2. Related Work

This thesis proposes a CSP-based protection mechanism, which offers XSS mitigation and general data leakage prevention. Many publications cover similar topics. Proposals range from server-side XSS detection [7] to client-side mitigation of the effects [8]. One of these mitigation techniques was proposed by Vogt et al. in 2007 [9]. Due to taint tracking of sensitive information, the mechanism allowed users to decide which data may be transferred to third parties.

Reis et al. argued in 2007 that the web is moving towards feature-rich web programs [3]. However, they pointed out major weaknesses of the web platform and proposed browser architecture refinements. One of these weaknesses is the poor separation between code and data in the web. This problem was tackled by multiple approaches, some of which were summarized by Louw et al. [10]. In 2007, Robertson et al. proposed a web application framework separating structure and content in a generalized way [9]. This allowed them to prevent both XSS and SQL injection vulnerabilities. Another approach separating untrusted from trusted content was proposed by Nadji et al. [11]. Untrusted content is marked by the browser and its capabilities can be limited. Until stated otherwise by a policy, the untrusted content is not able to use any HTML tags. Google Caja offers a similar capability control by sanitizing and rewriting HTML, CSS, and JavaScript [12]. Further research in the field of trusted and untrusted content separation was released 2009 by Louw et al. [13].

Three years before CSP, Jim et al. proposed and implemented a capability-controlling security policy for browsers [14]. In order to whitelist scripts, the content had to be hashed with the SHA-1 algorithm. A list of hashes was provided in the document, so that browsers could disallow all unintended and potentially malicious scripts. Currently, a similar mechanism is discussed for inclusion in the CSP 1.1 standard[1]. Furthermore, the proposal allowed parts of a document to be marked as untrusted, disallowing all kinds of active content.

A different approach was presented 2012 by M. Heiderich [15]. Instead of relying on the user agent to enforce a policy, a resilient client-side library was introduced. It can mediate access to critical DOM properties and therefore limit the capabilities of a document with a policy.

CSP was proposed by Stamm et al. [6] in 2010. Despite the fact that it is a relatively new proposal, it has gained much attention by the web security community. In 2011, Weinberger et al. evaluated a set of "HTML security policies" [16]. While pointing out the effectiveness of these policies, they also argue that these systems are still insufficient for the needs of web applications. CSP policy generation can be an error-prone and laborious task [17]. Therefore, a lot of work has been done to automate the generation of

[1] N. Green, *Proposal for script-hash directive in CSP 1.1*, `http://lists.w3.org/Archives/Public/public-webappsec/2013Feb/0052.html`, Oct. 2013

policies. Ideas like the CSP Bookmarklet[2] and CSP AiDer [18] try to address the issue on a site-to-site basis and automatically create a CSP policy based on the observed markup at the time of the execution. CSP AiDer suffers from the inability to observe dynamically added markup and both approaches do not consider multiple subpages or a dynamic environment. A more robust approach, leveraging the Report-Only functionality of CSP, has been implemented by P. Krawczyk[3]. UserCSP is a Firefox Add-on which enables users to define a CSP for websites that have no such policy, yet [17]. Additionally, the extension is able to infer a policy from the observed DOM.

[2]B. Sterne, *CSP Bookmarklet*, `https://github.com/bsterne/bsterne-tools/tree/master/csp-bookmarklet`, Oct. 2013

[3]P. Krawczyk, *Content Security Policy Builder*, `http://cspbuilder.info/`, Oct. 2013

3. Fundamentals

The following sections elaborate on the fundamentals needed to understand this thesis. First, an overview of the HyperText Transfer Protocol (HTTP) is given in Section 3.1. It serves as the basis for understanding the concept of a (reverse) HTTP proxy as described in Section 3.2. This concept is of key importance for the proxy proposed in Chapter 5. Then, languages directly relevant to the web and this thesis are introduced: The HyperText Markup Language (HTML) is outlined in Section 3.3, Cascading Style Sheets (CSS) in Section 3.4 and JavaScript in Section 3.5. A short summary of the JavaScript Object Notation (JSON) and its derivative JSON with padding (JSONP) is given in Section 3.6. Section 3.7 introduces the Document Object Model (DOM), a mediator between the elements and active content of a document. Before explaining attacks the proposed proxy aims to mitigate (cf. Section 3.9), a key component of browser security is explained – the Same-Origin-Policy (cf. Section 3.8).

3.1. HTTP

The Hypertext Transfer Protocol (HTTP) was invented 1989 by Tim Berners-Lee[1]. Today, it is one of the prevalent protocols in the Internet used to transmit resources. Version 1.1, as outlined in RFC 2616 [19], is the currently proposed standard, while version 1.0 [20] is still sparsely used. As most HTTP implementations use TCP for communication, they suffer from the overhead of all required handshakes, when requesting multiple resources. Most notably, HTTP 1.1 instructs clients to re-use connections, avoiding this overhead. Version 2.0 focusses on performance enhancements, too. It uses Google's SPDY protocol as the underlying basis and still is in a draft status[2]. Essentially, HTTP controls the communication between user agents (UAs), like browsers, and servers. It is a stateless protocol, meaning that two subsequent connections from the same client cannot be associated with each other on a protocol level.

Section 3.1.1 will explain the protocol in detail. Then, the HTTPS protocol is briefly outlined in Section 3.1.2. An exemplary HTTP communication is described in Section 3.1.3.

3.1.1. Protocol Details

HTTP 1.1 supports eight request methods which can be used to request resources and information.

- *GET* is the standard method to retrieve a resource without modifying it.

[1] T. Berners-Lee, *Biography*, `http://www.w3.org/People/Berners-Lee/`, Oct. 2013

[2] M. Belshe, R. Peon, M. Thomson and A. Melnikov, *Hypertext Transfer Protocol version 2.0*, `http://tools.ietf.org/html/draft-ietf-httpbis-http2-04`, Oct. 2013

- *POST* requests send data to the server which is eventually stored or used to modify other resources.
- A *HEAD* request only asks for the headers of a resource and is often used to check if cached resources have changed.
- *PUT* requests ask the server more specifically for storing the sent data.
- The *DELETE* request method instructs the server to delete the associated resource.
- A *TRACE* request can be used to ask the server to return the request.
- *OPTIONS* requests are used to get the supported request methods for a URL, but also for so-called "preflight requests". These requests are used to check if the server accepts Cross-origin resource sharing (CORS).
- At last, there is the *CONNECT* request method, converting the connection to a TCP/IP tunnel.

Status codes are used to indicate the condition of a request to a user agent. Each status code is a three digit number and there are five classes of status codes. There are informational (1xx), success (2xx), redirection (3xx), client error (4xx), and server error (5xx) status codes.

Every HTTP message consists of header and body, separated only by an empty line with a carriage return (CR) and line feed (LF). The header contains key-value pairs which have to end in a CR and LF, too. While HTTP headers mostly contain meta-information about the resource, the message body contains the actual payload.

3.1.2. HTTPS

The Transport Layer Security (TLS) protocol (or its predecessor Secure Sockets Layer) tremendously increases the security characteristics of HTTP. HTTP Secure (HTTPS) is the most commonly used protocol combining HTTP with TLS. Through the use of cryptography, HTTPS messages can be protected from eavesdropping. Most user agents have a list of Certificate Authority certificates which can be used to confirm the identity of a server. In theory, Certificate Authorities assure that this identity cannot be spoofed by adversaries, while the past has proved this trust relationship to be[3] difficult[4]. TLS provides the possibility for client authentication, too.

3.1.3. Example

A HTTP request is shown in Listing 3.1. It uses the GET method to request "file.html" from "example.com". Furthermore, the host name is transmitted in the "Host" header, making it possible for a single web server to distinguish and serve different websites. It is the only header an user agent is required to send in HTTP 1.1

[3] P. Bright, *Another fraudulent certificate raises the same old questions about certificate authorities*, `http://arstechnica.com/security/2011/08/earlier-this-year-an-iranian/`, Oct. 2013

[4] T. Espiner, *Trustwave sold root certificate for surveillance*, `http://www.zdnet.com/trustwave-sold-root-certificate-for-surveillance-3040095011/`, Oct. 2013

(apart from the method, file and version). The “User-Agent” header tells the server which user agent was used to request the resource. No message body is sent as part of the request.

```
GET /file.html HTTP/1.1
Host: example.com
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) ...
```

Listing 3.1: HTTP GET request

An exemplary response to the request of Listing 3.1 is presented in Listing 3.2. As the status code indicates, the resource was not found. The “Content-Type” header declares the error page to be HTML, encoded with a UTF-8 character set. Furthermore, the name of the web server software is transmitted in the “Server” header and the server’s current time in the “Date” header.

```
HTTP/1.1 404 Not Found
Content-Type: text/html; charset=UTF-8
Content-Length: 15
Date: Fri, 11 Oct 2013 20:02:31 GMT
Server: lighttpd

404 - Not Found
```

Listing 3.2: HTTP response

3.2. Proxy Servers

Proxy servers act as a mediator between clients and one or more servers. Initially, they were proposed as a way to structure large distributed systems [21]. Today, they are used for a broad range of applications: Since proxies conceal the real source address of a connection, they can be used to obtain anonymity in computer networks. Furthermore, proxies are able to monitor all network communications due to their role as a mediator between clients and servers. Naturally, these monitoring capabilities can also be utilized to manipulate and filter traffic. One example of this are HTTP caching proxies, which cache some or all observed resources [22]. If a client requests a cached resource, the proxy will directly respond with the resource without involving the original server. This can reduce both network and server loads [23].

Proxy software strongly differs in approach and implementation. Socket Secure (SOCKS) proxies, for example, are able to route arbitrary TCP and UDP traffic [24]. This thesis will focus on HTTP proxies, which are only capable of routing HTTP traffic [22]. Forward and reverse HTTP proxies are explained in the two subsequent sections.

3.2.1. Forward HTTP Proxies

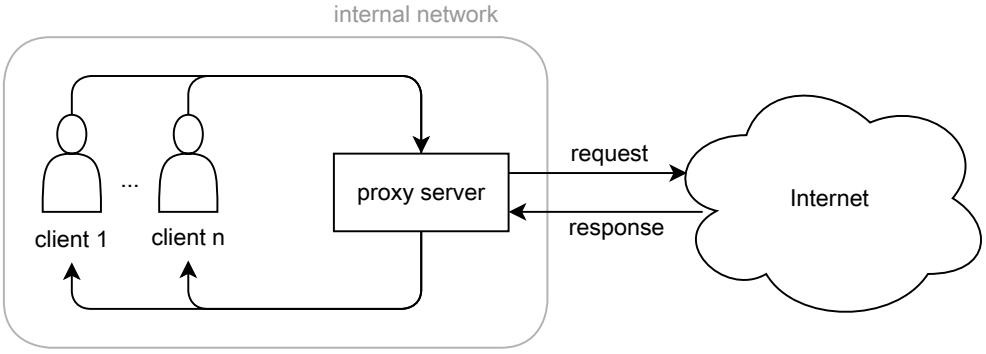

Figure 3.1.: A forward HTTP proxy acting as a mediator between one or more clients from an internal network and multiple web servers in the Internet

Forward proxies allow properly configured clients to connect to "origin servers"[5]. These "origin servers" may be part of another computer network, such as the Internet. Figure 3.1 shows a forward proxy which acts as a intermediate server between internal clients and the Internet. If the forward proxy is not part of an internal network and accessible by arbitrary clients from the Internet it is called an open proxy.

3.2.2. Reverse HTTP Proxies

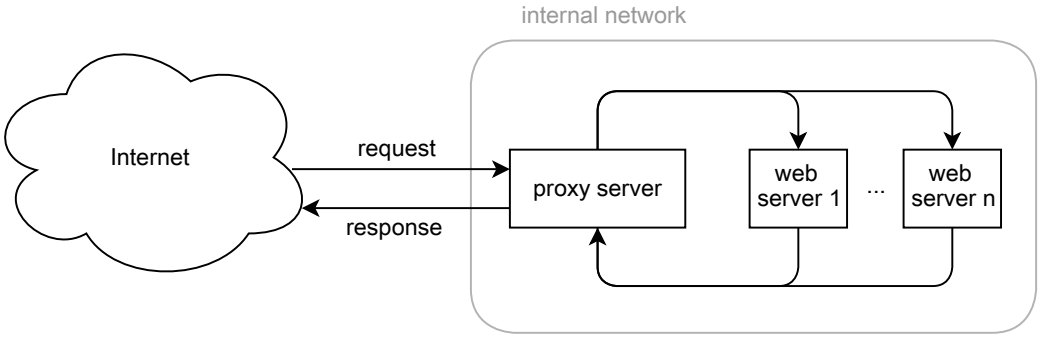

Figure 3.2.: A reverse HTTP proxy acting as a mediator between clients from the Internet and one or more web servers

In contrast to forward proxies, reverse proxies do not require any client-side configuration[5]. A reverse proxy acts as if it provides the requested service by itself. It routes requests to one or more servers and sends all responses back to the respective clients. Figure 3.2 presents a setup, hiding multiple web servers behind a reverse proxy. Only the reverse proxy is allowed to communicate with clients from the Internet.

[5]The Apache Software Foundation, *Apache Module mod_proxy*, `http://httpd.apache.org/docs/2.0/mod/mod_proxy.html#forwardreverse`, Oct. 2013

3.3. HTML

The HyperText Markup Language (HTML) was developed to structure documents in the web [25]. Nowadays, it is used widely, from native applications on mobile devices[6] to browser extensions[7]. HTML specifications are governed by the World Wide Web Consortium (W3C). This radically changed with the development of HTML5 [26], which was initiated by the Web Hypertext Application Technology Working Group (WHATWG). Browser vendors founded the WHATWG as a response to the W3C's plans to abandon efforts to evolve HTML 4.01 in favor of XML based standards like XHTML2[8]. Since then, the W3C compiled with the HTML5 specification and the WHATWG continues to work on HTML5, the "Living Standard" [27]. This means that the standard can never be considered complete and will evolve forever.

HTML syntax consists of tags, comments and text. Tags are enclosed by opening and closing angle brackets ("<" and ">") and contain an element name. HTML elements are expressed with a so-called "start tag" and an "end tag". Depending on the element, an end tag can be omitted. Start tags can contain attribute value pairs and elements can be nested, forming a tree structure (Section 3.7).

```
<!DOCTYPE html>
<html>
    <head>
        <title>Example</title>
    </head>
    <body>
        Content of the body.
        <a href="2.html">Hyperlink</a>
    </body>
</html>
```

Listing 3.3: HTML5 document

Listing 3.3 shows a HTML5 document. As seen in this example, elements can have multiple (or none) child elements. A `DOCTYPE` tells browsers about the used markup language and version. While technically not required by the HTML5 standard, the `html`, `head` and `body` elements outline the structure of a website. If they are missing, the browser automatically adds them to the document. Elements of the `head` typically contain meta-information about the document, most of which is not directly displayed in browsers, while the body contains the visible content. The anchor (`a`) element can point to other resources and trigger the navigation of the user agent when clicked.

[6] *Firefox OS*, `https://developer.mozilla.org/en/docs/Mozilla/Firefox_OS`, Oct. 2013

[7] *What are extensions?*, `http://developer.chrome.com/extensions/index.html`, Oct. 2013

[8] WHATWG, *HTML - 1.6 History*, `http://www.whatwg.org/specs/web-apps/current-work/multipage/introduction.html#history-1`, Oct. 2013

3.4. CSS

Cascading Style Sheets (CSS) were developed to separate the structure of a document from its presentation. While not limited to styling documents, CSS is primarliy used in combination with markup languages like HTML. The standardization process of CSS is categorized into levels. Level 1 is superseded by CSS2.1 [28], which is the main component of CSS Level 2. Level 3 is building on the same specification and extends it in various areas [29, 30, 31, 32].

CSS features a similar data structure like the DOM, called the CSSOM[9]. It enables access to the style sheets of a document with a well-defined API.

Section 3.4.1 elaborates on the syntax and precedence rules of CSS. Ways of embedding style sheets into markup and general examples are given in Section 3.4.2.

3.4.1. Syntax

Style sheets may contain multiple rules, which can be separated into *selectors* and *declaration blocks*. One or more selectors are used to identify elements, which should be styled by a declaration block. Selectors can select every element (universal), types, pseudo-classes, pseudo-elements, attributes and IDs [29]. Each line of Listing 3.4 shows one of the selector types. A declaration block is enclosed by curly braces and contains property-value pairs. These pairs contain the actual styling information. Some properties can be inherited from parent nodes.

```
* /* universal selector */
span /* type selector */
span:hover /* pseudo-class selector */
span::first-letter /* pseudo-element selector */
[attribute] /* attribute selector */
.classname /* class selector */
#idname /* ID selector */
```

Listing 3.4: CSS selector types

If two rules select the same elements and set the same properties, a precedence must be calculated. Selector specificity is used as the main factor for this calculation. Type and pseudo-element selectors have the lowest specificity. Class, pseudo-class and attribute selectors have a slightly higher precedence. The highest specificity is held by ID selectors, only surmounted by inline style attributes and the `!important` keyword. If two declarations have the same specificity, the latter is used.

3.4.2. Usage

Listing 3.5 shows a style sheet with two rules. First, the `body` element's background is set to green and its displayed font size to 20 pixels. Then, all elements with the ID "bold-title" or the class "someclass" are

[9] S. Pieters, G. Adams and A. Kesteren, *CSSOM - W3C Editor's Draft*, `http://dev.w3.org/csswg/cssom/`, Oct. 2013

selected. Text is displayed in a bold font in these elements. Furthermore, the rule is marked as important, giving it a high precedence.

```
body {
    background: green;
    font-size: 20px;
}
#bold-title, .someclass {
    font-weight: bold !important;
}
```

Listing 3.5: Two exemplary CSS rules

Style sheets can be embedded into HTML in three ways, as presented in Listing 3.6. When CSS code is mixed with markup, it is called *inline*, as shown in the first line of the listing. In the second line an external style sheet is loaded. An inline style attribute is demonstrated in the last line. Instead of using selectors, these attributes are directly defined on the elements which should be styled.

```
<style>body { color:red; }</style>
<b style="color:green;">green text</b>
<link rel="stylesheet" href="external.css">
```

Listing 3.6: Three ways of embedding CSS in HTML

3.5. JavaScript

JavaScript is an object-oriented programming language implemented in all major web browsers. It features a prototype-based inheritance model and dynamic typing. The language specification is governed by Ecma International and the current proposed version is ECMAScript 5.1 [33]. ECMAScript 6 is currently being developed under the code name "Harmony"[10]. While JavaScript was originally designed to serve as a client-side scripting language, it can be deployed as a server-side programming language, too [34].

There are four ways to embed JavaScript in HTML and each of them is shown in Listing 3.7. As with CSS, there is *inline* and *external* JavaScript. The first line shows an external script. It will be fetched from the URL and the markup parser will wait for the script to execute until it continues. This is also true for the inline script in line two. External script execution can be deferred with the `defer` attribute. The `async` attribute will continue markup parsing right away and execute the external script when it is ready. Line 3 demonstrates an inline event handler, which will execute the JavaScript code when the button is clicked. There are many different inline event handlers, but they can be generally identified by their prefix "on". Examples of this are "onmouseover", "oncut" and "onfocus". The last line of the Listing features the

[10] *ECMA-262. 6th Edition / Draft September 27, 2013. ECMAScript Language Specification*, `http://people.mozilla.org/~jorendorff/es6-draft.html`, Oct. 2013

JavaScript pseudo protocol. This protocol can be used in various locations which expect an URL. In the Listing, the JavaScript code will be executed after the link was clicked.

```
<script src="external.js"></script>
<script>alert('inline');</script>
<button onclick="alert('eventhandler');">click me</button>
<a href="javascript:alert('protocol')">click me</a>
```

Listing 3.7: Four ways of embedding JS in HTML

JavaScript code can read and manipulate the DOM. A simple DOM manipulation is shown in Listing 3.8. This enables JavaScript to manipulate the look and feel of a document. Furthermore, scripts are capable of sending data to a server with XMLHttpRequests [35] and even open a bidirectional communication channel to a server using the WebSocket API [36].

```
<div id="demo">
The Answer to the Ultimate Question of Life, the Universe, and
Everything?
</div>
<script>
var elt = document.getElementById('demo');
elt.innerHTML = '42.';
</script>
```

Listing 3.8: DOM manipulation in JavaScript

3.6. JSON

The JavaScript Object Notation (JSON) is derived from the ECMAScript programming language [37]. It was designed to be a text-based and language-independent data interchange format.

Six data types are defined in the JSON specification. *Strings* are enclosed by double quotes and can contain Unicode escape sequences, using JavaScript's syntax ("\uXXXX"). *Numbers* are allowed to be integers or in a floating point format. The precision is not determined by the specification. *Boolean* values can be represented using the lower case keywords `true` and `false`. An *Array* contains other data types separated by commas and is enclosed by square brackets. *Objects* map keys to values. Keys must be strings but values can resemble any data type. Keys are separated from values with colons and key value pairs with commas. This data type is enclosed by curly braces. The *null* keyword can be used to represent a value void of any other data type.

```
{"string": "value", "boolean": true, "null": null, "number": 42,
 "list of numbers": [1, 2, 3.14159]}
```

Listing 3.9: JSON data structure

Listing 3.9 presents an exemplary JSON data structure. It demonstrates every aforementioned data type, showing that those can be mixed and nested at will.

3.6.1. JSONP

JSON with padding (JSONP) is exploiting a property of the web to achieve browser independent cross-origin data sharing[11]. Reading data from other origins is normally prohibited by the SOP. JSONP is uses the ability to embed resources from any origin without restrictions. A cross-origin resource has to wrap the JavaScript data structure in a function call (the "padding"). Then, an embedding website can define this callback function and embed the resource as a script. It will execute and pass the data to the previously defined callback function. Reading the content of the embedded resource is not required anymore because it passes its data voluntarily to any document defining a callback function. As JSONP uses real JavaScript data types to pass data, the name is not fully accurate because JSON is not a subset of JavaScript but merely derived from it[12].

Listing 3.10 shows an example of a JSONP source, which is embedded by the code in Listing 3.11. In this example, the callback function on b.com will receive the JavaScript object as its first parameter. It is common practice to make the name of the callback function changeable through an URL parameter. An URL query string "?callback=invoke", for example, could set the callback function to "invoke" in such implementations.

```
callback({"key": "value", "key2": [1, 2, 3.14159]});
```

Listing 3.10: JSONP source on a.com

```
<script>function callback(data) { /* handle data */ }</script>
<script src="http://a.com/jsonp"></script>
```

Listing 3.11: JSONP-embedding document on b.com

3.7. DOM

When a browser parses the markup of a document, it gradually builds up a tree structure of all elements. This structure is called a Document Object Model (DOM) tree and all elements of the markup are called nodes, including text nodes. The root of all nodes in a browser is the document object. The DOM allows for dynamic changes to the structure and content of a document using a programmatic API. It exposes various other information, like the location (URL) of the document, the cookies and so forth.

Some nodes of the DOM must be unique and are not allowed to occur more than once. This includes the document object, which cannot be created by markup, but also the `html`, `body` and `head` elements. If any of these elements occur more than once, they will be merged into one node.

[11] *Defining Safer JSON-P*, `http://www.json-p.org/`, Oct. 2013

[12] M. Holm, *JSON: The JavaScript subset that isn't*, `http://timelessrepo.com/json-isnt-a-javascript-subset`, Oct. 2013

Currently, the latest specification of the DOM is the DOM3 standard [38], but DOM4 is actively being developed [39].

3.8. Same-Origin-Policy

Listing 3.12 shows an URL with all of its possible parts. Some parts of the URL can be omitted, such as the authentication information ("username:password@") and the port, which will default to 80. All combined parameters, which occur behind the first question mark are called the query string. This excludes the fragment identifier, separated by the number sign ("#"), which is special in the way that it will not be sent to the server by user agents.

```
protocol://username:password@host:port/path/file.ext?param=val#fragment
```

Listing 3.12: Full URL example

A web origin is consisting of the preeminent characteristics of a URL, namely the protocol, host name and port [40]. The concept of an origin is used by browsers to define a boundary which is essential to web security. The mechanism enforcing this boundary is the Same-Origin-Policy (SOP). Active content, such as JavaScript, is only allowed to cross this boundary on very rare occasions. These occasions often require the resources of such a cross-origin communication to comply with the request. Without the SOP, no secret could be safely transmitted or stored on a web application, undermining the security of any service using authentication and eliminating all privacy. Adversaries could steal data from any web application if they could manage to lure a victim on to a malicious page. As indicated above, the SOP contains many exceptions, which exist due to legacy peculiarities and features of newer web standards [41]. Embedding cross-origin images, for example, is possible without restrictions.

3.9. Attacks

This Section introduces attacks which are prevalent in today's web and relevant to the proposed proxy. First, data leakage is explained in Section 3.9.1. It is an important effect of various attacks, such as Cross-site scripting, which is described in Section 3.9.2.

3.9.1. Data Leakage

Data leakage is not an attack itself, but rather can be the effect of one. In general, the term describes an unauthorized transfer of information to a (potentially malicious) third party [42]. This very broad definition includes intentional as well as unintentional data leakage. While the information does not have to be sensitive, it mostly is if an attack was conducted to obtain it.

An E-Mail advertising service, for example, may be interested in knowing if a spam mail was read. In order to observe this, it could embed images into a HTML mail. When a victim opens the E-Mail, the mail client might retrieve the images. As this sends a request to the server of the advertising service, it is giving

a clear indication that the E-Mail was opened. Many mail clients protect from this data leakage by not retrieving images of unknown E-Mails. Only after the user has deemed the mail to be benign, the requests will be sent. In the web, the SOP prevents trivial data leakage.

3.9.2. Cross-Site Scripting

Cross-site scripting (XSS) belongs to the family of code injection attacks. Vulnerable web applications offer attackers the possibility to inject (potentially malicious) client-side code. Since it is run in the context of the web application, the SOP cannot prevent data leakage from happening. Consequently, an attacker is able to utilize all capabilities of client-side languages like JavaScript. Data leakage is one of the potential results of this vulnerability. Attackers might steal cookies, credentials and other private data. Otherwise, since the complete DOM is susceptible to manipulation, an adversary can change the look of the website (defacement). XSS vulnerabilities may arise from insufficient sanitization of user input or browser bugs. These attacks cannot be thwarted by user agents, since the basic technologies used in the web offer no possibility to separate structure from content [3]. Thus, a client cannot decide which part of the markup was intended by the web application.

Class	Description
Reflective	Payload sent by one client and is directly reflected back
Persistent (or stored)	Payload sent once and is served on subsequent visits
DOM-based	Payload triggered by client-side code
Mutation-based	Payload triggered by mutating DOM sinks

Table 3.1.: The four classes of XSS

Four classes of XSS are listed in Table 3.1. A XSS vulnerability is called *reflected*, if an user agent sends the attack payload to a web server and it is reflected back in the code of the web application. Parameters of the query string could, for example, be manipulated by an adversary. A vulnerable web application may improperly sanitize the input and return it as a part of the markup. Such a carefully prepared URL could be sent to a victim, enabling the attacker to leak private data. There are multiple ways of disguising the intention of the malicious URL, including link shortening, redirections, social engineering [43], and so forth. If the attack vector is saved and served on subsequent visits of the web application, it is called *persistent*. In contrast to *reflected* XSS flaws, it may not require any interaction between attacker and victim. *DOM-based* XSS vulnerabilities are caused by insecure client-side code [44]. In JavaScript, just a few sinks have been identified, which can lead to this class of XSS[13]. Many DOM-based XSS vulnerabilities are hard to detect on the server-side because the payload never leaves the client. A payload may, for example, be appended to the fragment identifier of a URL and thus never sent to the server. The fourth class of XSS attacks is called *mutation-based* XSS (mXSS) [45]. Properties like `innerHTML` enable developers to add markup to the DOM. Browsers have to ensure that the markup does not break the DOM tree and therefore it will

[13] S. Di Paola, *domxsswiki - Dom Xss Test Cases Wiki Cheatsheet Project*, `https://code.google.com/p/domxsswiki/`, Oct. 2013

be normalized. This mutation of the original markup string can lead to covert vulnerabilities, which affect all websites using `innerHTML` to output user content. Due to their nature, mXSS vulnerabilities must be fixed by browser vendors, leaving many users with older user agents unprotected. Instead of fixing the vulnerability, the effects can be mitigated with capability controlling mechanisms [45].

4. Content Security Policy

Content Security Policy (CSP) was proposed by Stamm et al. in 2010 and aims to be an additional defensive layer for existing web applications [6]. Based on this work, the W3C Web Application Security Working Group formed a specification [46]. CSP 1.0 is in the state of a W3C Candidate Recommendation since November 15th 2012. Currently, a new proposal is being developed[1], extending CSP in various ways, and addressing some of the problems of version 1.0. A policy can be transferred as an additional HTTP header of a resource or via a `meta` element in the markup of a document (experimental in CSP 1.1).

The core idea of CSP is to limit the capabilities of a document. resources are categorized into directives which control where they can be loaded from and connect to. Section 4.1 explains the core concept of CSP and Section 4.2 outlines all CSP 1.0 directives. A special mode of CSP, the "Report-Only" mode, is explained in Section 4.3. Then, CSP 1.1 is compared to CSP 1.0 in Section 4.4. Finally, Section 4.5 outlines the limitations of CSP and peculiarities of current implementations.

4.1. Concept

Using CSP, web applications can limit the capabilities of all served resources. Most notably, XSS attacks can be mitigated by this policy when three factors are given.

1. Script and object sources must be tightly controlled. It might prove to be hard for an attacker to serve a malicious payload from a whitelisted domain.

2. Inline code must be disabled, which is enforced by default. As inline code might be injected by an attacker, the browser cannot decide upon its origin. Conversely, if a browser could tell apart benign from malicious inline code, XSS could be mitigated automatically on the client side. Obviously, this is not the case.

3. *eval* and *eval*-like constructs have to be disabled because they could lead to potential DOM-based XSS vulnerabilities. Similar to inline code, these functions are disabled by default, too.

If these rules are enforced and no whitelisted URI can be compromised, an attacker may inject markup, but cannot execute JavaScript code. While this mitigates traditional XSS, other attacks trying to achieve data leakage might still work. An attacker is able to use CSS or other resource types for content extraction [47]. However, CSP allows to control each resource type employed in browsers. Additionally, the connection

[1] A. Barth, M. West and D. Veditz, *Content Security Policy 1.1 - W3C Editor's Draft*, `https://dvcs.w3.org/hg/content-security-policy/raw-file/tip/csp-specification.dev.html`, Oct. 2013

sinks of APIs like *XMLHttpRequest* can be limited to a small set of whitelisted URIs, too. This reduces the attack surface of the web application substantially and tremendously limits the possibilities for data leakages forced by an attacker.

4.2. Directives

```
Content-Security-Policy: default-src 'none'; script-src 'self';
                         style-src example.com foo.com;
                         object-src *.cdn.example.com;
                         img-src *; media-src data:;
                         frame-src https://example.com;
                         connect-src: *
```

Listing 4.1: Example of a Content Security Policy header

Listing 4.1 shows an exemplary CSP. Policies consist of directives and source expressions, separated by semicolons. Every directive can contain multiple source expressions which are separated by spaces. A source can be a keyword ("keyword-source"), a protocol handler ("scheme-source"), an origin ("host-source") or a path level URI ("ext-host-source"). Generally, source expressions whitelist URIs and resource types. Origins and URIs are allowed to contain wildcards ("*"), allowing a wide range of subdomains or even all domains (as seen at the `img-src` directive in Listing 4.1). Single quotes always denote keywords while all other values have to be parsed to get the type. There are two general keywords which can be used by every directive: `'none'` disables all resources of that kind and `'self'` only allows resources to be loaded if they have the same origin as the enforcing document.

Directive	Possible values
default-src	{scheme,host,ext-host}-source, `'self'`, `'none'`
connect-src	{scheme,host,ext-host}-source, `'self'`, `'none'`
font-src	{scheme,host,ext-host}-source, `'self'`, `'none'`
frame-src	{scheme,host,ext-host}-source, `'self'`, `'none'`
img-src	{scheme,host,ext-host}-source, `'self'`, `'none'`
media-src	{scheme,host,ext-host}-source, `'self'`, `'none'`
object-src	{scheme,host,ext-host}-source, `'self'`, `'none'`
script-src	{scheme,host,ext-host}-source, `'self'`, `'none'`, `'unsafe-inline'`, `'unsafe-eval'`
style-src	{scheme,host,ext-host}-source, `'self'`, `'unsafe-inline'`, `'none'`
report-uri	URI
sandbox	`allow-forms`, `allow-pointer-lock`, `allow-popups`, `allow-same-origin`, `allow-scripts`, `allow-top-navigation`

Table 4.1.: CSP 1.0 directives

Table 4.1 lists all CSP 1.0 directives and their possible values. `default-src` sets default values for all directives except `sandbox` and `report-uri`. This allows resources of any kind to be loaded from the whitelisted URIs. Instead, it might be desired to have a more granular control over single resource types. While not a resource type itself, the `connect-src` directive limits the communication capabilities of executed scripts. Only whitelisted resources can receive a request from the protected document, using APIs like *WebSocket* or *XMLHttpRequest*. Web fonts give web developers the ability to use nonstandard fonts in layouts and designs [48, 49]. Their sources can be controlled with `font-src`. `Frame` and `iframe` elements are able to embed other documents [26]. `frame-src` can limit the allowed document URIs. If a `img-src` directive is employed, images must be whitelisted in this directive to be requested and displayed in the document. This also includes images loaded by external resources like style sheets. Media elements, for playing back audio and video sources [26], can be controlled by the `media-src` directive. Object and embed elements can embed arbitrary resources [26]. Furthermore, they can be used to embed plug-in content (like Adobe Flash). Since some plug-ins have similar capabilities as scripts, they offer data leakage possibilities for attackers and hence can be controlled with the `object-src` directive. Scripts themselves can be limited by `script-src`. Aside from whitelisting URIs, the directive can be used to allow inline code and *eval*-like constructs. Allowing inline scripts with `'unsafe-inline'` subverts the XSS mitigation of CSP, the browser could not tell the attacker's inline code apart from the server's. Nevertheless, this can be used if the policy is deployed to serve other purposes. As *eval* and *eval*-like constructs can be used for DOM-based XSS attacks[2], the specification highly recommends to avoid whitelisting `'unsafe-eval'`, too. Style sheets can be controlled by the `style-src` directive. Similar to scripts, inline styles are disallowed by default. This can be relaxed with the `'unsafe-inline'` keyword. CSP contains the `sandbox` directive which mimics the sandbox attribute of `iframe` elements. This makes it possible to further reduce the capabilities of the served document. The `report-uri` directive does not control a security mechanism. Instead, it defines a single URI which will receive reports, containing detailed information of policy violations. Structure and content of these reports is explained in Section 4.3.

4.3. Report-Only

```
Content-Security-Policy-Report-Only: script-src example.com;
                                     report-uri /sink;
```

Listing 4.2: Example of a CSP Report-Only header

Besides the normal policy enforcement, CSP supports another mode of operation: In Report-Only mode no rule will be actively enforced, but all violations of the policy will be reported. The sink of all reports can be specified using the `report-uri` directive. If this directive is not specified, the browser neither reports the violations nor enforces the policy. Instead of defining the `report-uri` in Report-Only mode, it can also be employed in CSP's regular mode. This allows for detection of violating resources during policy enforcement.

[2] S. Di Paola, *domxsswiki - Browser JavaScript execution sinks*, `https://code.google.com/p/domxsswiki/wiki/ExecutionSinks`

Listing 4.2 shows an example of a Report-Only header. Scripts are only allowed from "example.com" and violation reports will be sent to "/sink". The slash at the beginning indicates that this URI is pointing to the same origin on which the protected document resides on. Reports are encoded in a JSON data structure and sent via HTTP POST whenever a violation of the policy occurs. Listing 4.4 shows a sample report. It can be observed on "/sink" when a browser navigates to a document with the markup shown in Listing 4.3, using the CSP Report-Only header of Listing 4.2.

```
<!DOCTYPE html>
<html>
    <head>
        <title>Example document</title>
    </head>
    <body>
        <script src="http://evil.com/hook.js"></script>
    </body>
</html>
```

Listing 4.3: HTML document with a potential injection

```
{"csp-report":{"document-uri":"http://ourdomain.com/file.html",
"referrer":"", "violated-directive":"script-src example.com",
"original-policy":"script-src example.com; report-uri /sink;",
"blocked-uri":"http://evil.com"}}
```

Listing 4.4: CSP violation report

Violation reports consist of a single JSON data structure. Five key-value pairs are mapped to the "csp-report" key of an enclosing JSON object. "document-uri" holds the full URL of the document in which the violation occurred. Furthermore, the document's "Referer" header is transmitted in "referrer". In "violated-directive", the CSP directive, which was violated by the resource URI in "blocked-uri", is stored. "original-policy" contains the complete policy which protected the document.

Blocked same-origin URIs are submitted in the report with a file-level URI. On the contrary, only the origin of cross-origin URIs is transmitted. This behavior is demonstrated by Listing 4.4 and is dictated by the specification to prevent information leakage [46].

4.4. CSP 1.1

Since CSP 1.1 is still a W3C Editor's Draft at the time of this writing, all features discussed here remain subject to changes[3].

[3] A. Barth, M. West and D. Veditz, *Content Security Policy 1.1 - W3C Editor's Draft*, https://dvcs.w3.org/hg/content-security-policy/raw-file/tip/csp-specification.dev.html, Oct. 2013

A noteworthy change of CSP 1.1 is the possibility to define whitelist URIs up to file-level. Selectively allowing only single scripts of a domain is not possible in CSP 1.0 [46]. This helps to further reduce the attack surface of a web application while keeping backwards compatibility due to the CSP 1.0 specification instructing implementations to ignore the path component of the URL.

Furthermore, CSP 1.1 allows CSP headers to be transmitted in a `meta` element in the markup. The security implications of this are briefly discussed in Section 4.5.2.

One of the more prominent changes of the new proposal is the ability to define nonces. If a valid nonce is found in the markup, the corresponding element will be allowed to load and execute. This can help to developers to avoid the externalization of all inline scripts and styles but comes at a great risk. If an attacker manages to guess or steal a valid nonce the CSP would be bypassed. This is risky considering that nonces are part of the markup and thus, the DOM. An exemplary attack is presented in Section 4.5.2.

Multiple attacks on CSP 1.0 used HTML forms and the `base` element to extract data from web applications [50]. CSP 1.1 includes experimental `form-action` and `base-uri` directives, which can be used to mitigate these attacks.

`plugin-types` is another new directive and restricts plug-ins to whitelisted MIME types, giving greater control over embedded objects (previously only controlled by `object-src`).

Browsers can be prevented from sending a HTTP "Referer" header with the `referrer` directive. Four keywords can be used to specify the browser's behavior with respect to the referrer: `never` will completely oppress the header while `origin` will only leak the document's origin. `default` will keep the complex default rules active and `always` instructs the browser to always send a referrer.

CSP 1.1 incorporates the non-standard HTTP header "X-XSS-Protection" in the `reflected-xss` directive. It controls the behavior of client-side XSS filters and can be used to disable, enable or put the filter into blocking mode. Blocking mode displays an empty page when the blacklist match occurs, while "normal" mode tries to sanitize the alleged evil code. Currently, only WebKit browsers (including the Blink fork of WebKit) [51] and Internet Explorer[4] natively implement such filters. In the Gecko family of browsers, the implementation of a client-side XSS filter is still an open issue[5]. However, it is unlikely to be tackled soon because of browser add-ons serving a similar purpose[6].

4.5. Limitations

CSP has a number of downsides and limitations. Conceptual limitations of both CSP 1.0 and 1.1 are outlined in Section 4.5.1. In theory, the specification dictates the security excellence of CSP. However, in practice, web applications have to rely on the actual implementation. Therefore, Section 4.5.2 deals with some of the limitations of current implementations in modern web browsers.

[4] D. Ross, *IE 8 XSS filter architecture/implementation*, `http://blogs.technet.com/srd/archive/2008/08/18/ie-8-xss-filter-architecture-implementation.aspx`, Oct. 2013

[5] J. Ruderman, *Bugzilla Bug 528661: (xssfilter) Heuristics to block reflected XSS (like in IE8)*, `https://bugzilla.mozilla.org/show_bug.cgi?id=528661`, Oct. 2013

[6] G. Maone, *NoScript Firefox extension*, `http://noscript.net/`, Oct. 2013

4.5.1. Conceptual Limitations

As a CSP can whitelist resources, it is expressing trust in these resources. Conversely, if this trust relationship is subverted by an attacker, the protection may be bypassed. In the worst case, a whitelisted script or plug-in file might be compromised by an attacker. This would lead to a total bypass of the protection because data leakage would be possible through redirection or similar methods. Depending on the policy, other whitelisted resource types might also be used for an attack. Naturally, the probability of such an attack is increased by whitelisting entire domains instead of file-level URIs. Consequently, the use of CSP 1.1 increases the security characteristics in regards to this attack because it allows for more granular policies.

The security implications of this are grave: JSONP, for example, is a critical resource type which may be used for an attack – without compromising the source. As it is embedded as a script, it needs to be whitelisted in CSP's `script-src` directive. Usually, JSONP implementations allow a callback function to be specified in the query string. This function name is prepended to the data the source emits. As query strings are not considered by CSP, an attacker is able to change the function at will. Furthermore, since the amount of scripts from the same whitelisted URI cannot be limited by CSP, it is possible to chain multiple `script` elements pointing to the same JSONP source. In consequence, an attacker might be able to leak data, using this chain and controlling the function calls. Furthermore, if the query string is output in an improperly sanitized way, arbitrary JavaScript code can be injected to simplify the exploitation. Similarly, file uploads to whitelisted origins can be used to bypass the protection. As many of such examples exist, the attack scenario can be generalized: If a web application allows users to control the first characters of any resource on a whitelisted URI, the CSP can be bypassed without compromising server-side files.

CSP 1.1 introduces a `meta` element for the definition of CSPs (cf. Section 4.4). Since it is part of the markup, the element is prone to injection attacks occurring in the prepending code. Listing 4.5 shows an exemplary attack. It uses a HTML comment to wrap the `meta` element and prevent it from being parsed. The specification tries to counter this with enforcing the `meta` element's position to be in the document's head, where injections are less likely to occur. However, the possibility of an injection still exists.

```
<!DOCTYPE html>
<html>
    <head>
        <script>xss()</script><!--
        <meta http-equiv="content-security-policy"
              content="default-src 'none'">
        ...
    </head>
    ...
</html>
```

Listing 4.5: Bypass of a CSP delivered in a `meta` element with an open comment

It should be noted that CSP relies on the Domain Name System (DNS), just like the SOP. In consequence, it is vulnerable to the same attacks. Multiple attacks against the DNS were published in the past [52, 53].

4.5.2. Limitations of Implementations

There is a class of attacks, introduced by implementation bugs, which cannot be foreseen by the specification. These may bypass the security of a CSP partially or completely. In the course of writing this thesis, one such bug was found[7]: An attacker can make use of the session restore mechanism in Mozilla Firefox up to version 27 to completely bypass the CSP of a web application. When navigating to a data-URI via a hyperlink, the context of the parent site is inherited in Firefox. This means that the document constructed from the data-URI is not restricted by the SOP when trying to access DOM properties of the parent page (like cookies). This is true for the CSP of the parent site, too. Another trait of Firefox is that when a crash or restart occurs, the last browsing session will be restored. This session restore mechanism forgets about the CSP, when restoring a data-URI, making it an exploitable bypass. An attacker can frame the victim site and lure a user into clicking the injected data-URI hyperlink containing the payload. This navigation is detectable by the parent frame which can then crash the browser with an unrelated exploit. Finally, the session restore mechanism will re-establish the browsing session, giving the malicious code full access to the victim site.

The nonce feature, introduced in CSP 1.1, suffers from the lax parsing of browsers. Unclosed HTML tags can be used to "steal" the nonce from a legitimate element in Chrome, up to the latest version (29 at the time of this writing). This can be used to execute arbitrary JavaScript code when an injection occurs before a whitelisted `script` element as shown in Listing 4.6. Other variations of this attack are imaginable. As the HTML5 specification requires such lax parsing rules [26], this problem will most likely persist and emerge in other browsers, too.

```
<script src=//injected.url/ <script nonce="value">[...]</script>
```

Listing 4.6: Injected `script` element "stealing" the nonce of a legitimate script

Implementation bugs are not always critical to security. Firefox completely dismisses file- and path-level URIs in policies[8]. According to the CSP 1.0 specification the origin of such URIs should be whitelisted, ignoring the path or file component. Nevertheless, this bug prevents strict policies, as defined in Section 5.1 of this thesis, from working in Firefox.

[7] N. Golubovic, *Bugzilla Bug 911547: data-URI + Firefox restart = CSP-bypass*, `https://bugzilla.mozilla.org/show_bug.cgi?id=911547`, Oct. 2013

[8] N. Golubovic, *Bugzilla Bug 916054: URLs with path are ignored by FF's CSP parser*, `https://bugzilla.mozilla.org/show_bug.cgi?id=916054`, Oct. 2013

5. Reverse HTTP Proxy

This Chapter describes a reverse HTTP proxy called “autoCSP”. It automatically generates policies and externalizes inline code for all served documents. Section 5.1 outlines the design goals and choices of the proxy. Implementation details are explained in the following sections, starting with the first mode of operation – the *learning mode* – in Section 5.2. Finally, Section 5.3 describes the *locked mode* in detail.

5.1. Design

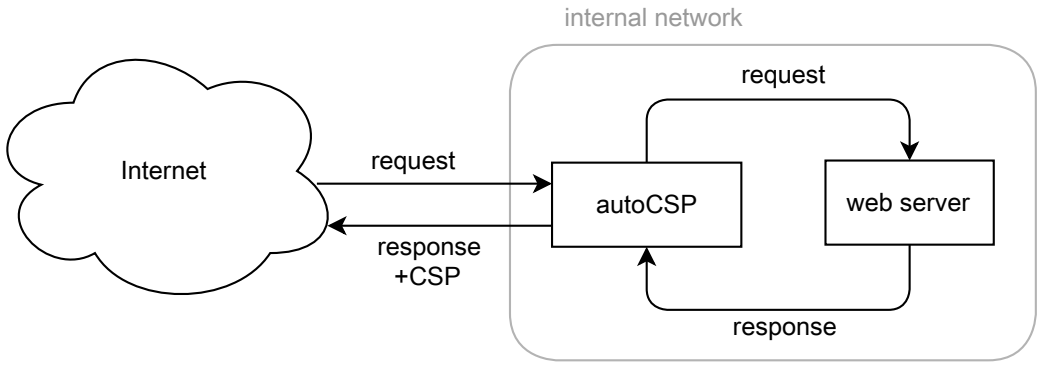

Figure 5.1.: Core idea of autoCSP

Figure 5.1 presents the core idea of the reverse HTTP proxy. An arbitrary web application can be locked behind the proxy and will be automatically protected by a strict CSP. A *strict policy* shall be defined to fulfill three requirements:

- It must include all resource-controlling CSP directives (explicit or implicit via `default-src`).
- Only resources needed by the respective document may be whitelisted. Thus, a complete domain should never be contained.
- It may not use any unsafe keyword, like, for example, `'unsafe-inline'`.

While the policy is required to be strong, it should not break website functionality. It may lower the acceptance of the protection mechanism, if web applications have to be changed to work with the proxy. Therefore, the proxy has to fix CSP-incompatible code patterns automatically. This leads to the following design goals:

1. The proxy should be transparent to clients and web applications.
2. All HTTP responses must be protected by a CSP.
3. Policies have to be automatically inferred and be strict.
4. autoCSP must ensure that enforced CSPs do not break the proxied web application.

Reverse proxies may have, but do not necessarily have to have knowledge of the proxied web application. Satisfying the first design goal, autoCSP treats the web server and all clients as black boxes. Hence, the reverse HTTP proxy can be deployed without modifying the web application's code. Despite being a major advantage, this property of the proxy also complicates the policy generation process tremendously. Instead of knowing which parts of the markup are intended by the web application, the proxy must infer this knowledge. This situation is comparable to a web browser trying to tell the web application's markup apart from injected code. It is simply not possible without additional information as evidenced by all the content injection attacks possible in the modern web (cf. Section 3.9). However, since the proxy is bound to protect only one web server, it is feasible to learn the structure of the web application before protecting it. While this might be seen as a violation of design goal number two, in fact, it is not. This is explained in Section 5.1.3.

Two modes of operation are introduced in the following sections. Section 5.2 covers the *learning mode*. In this mode, the proxy observes resource URIs and attempts to externalize all inline code. Section 5.3 describes the *locked mode*. It forms strict policies from the previously observed resource URIs and enforces them, while the externalized code is used to retain the functionality of the web application. Switching between the two modes is possible at any time but requires the proxy to be manually restarted. This raises the bar for adversaries because command execution on the proxy server is a prerequisite to switch to the (unsafe) *learning mode*.

Since CSPs employed by the web application itself might interfere with automatically generated policies, the proxy will discard all CSP headers of the original HTTP response.

Enforcing strict policies requires the use of CSP 1.1, since only this version allows for file-level URIs (cf. Section 4.4). While in theory the protection of the proxy is browser agnostic, only the Chromium browser family largely implements CSP 1.1[1].

5.1.1. Software Architecture

Since the proxy was implemented in the Python programming language[2] using the libmproxy library[3], it handles parallel connections with threads [54]. Furthermore, it is capable of serving resources via both HTTP and HTTPS. Data persistence is achieved with SQLite[4], requiring no manual database setup.

[1] F. Beaufort, `https://plus.google.com/100132233764003563318/posts/fLRLPHeiAAV`, Oct. 2013
[2] *Python Programming Language*, `http://www.python.org/`, Oct. 2013
[3] A. Cortesi, *libmproxy: mitmproxy as a library*, `http://mitmproxy.org/doc/library.html`, Oct. 2013
[4] *SQLite*, `http://www.sqlite.org/`, Oct. 2013

For the core components of the proxy, an event-driven architecture was chosen. Table 5.1 lists all observable internal events. So-called *interceptors* may subscribe to the events and will be called when such an event occurs. They will be executed both in *learning* and *locked mode*, but can be limited to one of them. Every event can have multiple *interceptors*, which are called in the order they are defined in the global settings file of the proxy. If an *interceptor* wishes to stop the following *interceptors* from executing, it can raise a special exception called `StopEventPropagation`. The next paragraph will elaborate on the observable events and describe use cases.

Event	Description
db_init	New database created
request	HTTP request from a client
response	HTTP response of the server

Table 5.1.: Internal events

A "db_init" event occurs when no database could be found on proxy startup. In that case a new database will be created and all *interceptors* subscribed to this event are given the possibility to recreate the needed database tables. "Request" events emerge when the client sends a HTTP request to the proxied web server. This request can be modified at will and is forwarded to the web server after all subscribed *interceptors* were executed. Instead of transmitting the request to the web server, an *interceptor* may decide to directly answer the client. This is, for example, used by the web interface presented in Section 5.2.3. A "response" event occurs when the proxied web server emits a HTTP response. This response can be modified by *interceptors* and will be forwarded to the respective client afterwards. Various content and header injections are conducted using this event, including the adding of the CSP headers and injecting scripts into the served document.

5.1.2. Database Layout

All database tables used by the proxy are listed in Table 5.2. As mentioned before, inline code has to be externalized to retain the functionality of the web application. Hence, the mechanism presented in Section 5.2.2 stores inline code in the "inline" database table. Table 5.3 presents the data structure used to save inline code. Two IDs are used to identify a database record: "id" is a unique ID which allows for global identification in "inline". Moreover, it may be desirable for the proxy to associate inline code snippets originating from the same request. For this reason a 32 byte long alphanumeric string is saved in "request_id". Externalized code must only be executed on the URI it was observed on, thus it is saved in the "document_uri" field. There are multiple types of inline code, for instance CSS style attributes and inline event handlers. "type" associates an abbreviated type to the code, which is stored in "source". This code is hashed with the SHA256 algorithm for reasons explained in Section 5.3.2 and the hash is saved in the "hash" field. Duplicate entries in the database are avoided by forcing each ("document_uri", "type", "hash")

Name	Description
inline	Externalized inline code
policy	Collected CSP rules
violations	CSP violation reports
warnings	Web interface warnings

Table 5.2.: Database tables

tuple to be unique.

Field	Description
id	Unique ID
document_uri	URI of the document
type	Type of inline code
source	Externalized source code
hash	SHA256 hash of the code
request_id	Request-identifying ID

Table 5.3.: Data structure of externalized code

Field	Description
id	Unique ID
document_uri	URI of the document
directive	Effective CSP directive
uri	Observed resource URI
request_id	Request-identifying ID
activated	Rule activation status

Table 5.4.: Data structure of policy rules

Policy generation is another core mechanism of the proxy. The "policy" database table contains the observed resource URIs and their respective CSP directives. This data structure is shown in Table 5.4 and will be referred to as a *policy rule*. Analogous to the "inline" database table, the data structure can be identified by "id" and "request_id". Furthermore, it also features a "document_uri" field. Policy rules are limited to the document they were gathered from, just as externalized code. If this would not be the case, a design goal of the proxy would be violated. The design goal states that every policy must be strict and hence not whitelist any unnecessary URIs. Obviously, this is violated, if two parts of the web application use differential resources and policy rules are not bound to a document URI.

As all resource types can be categorized into CSP directives, the effective directive for the policy rule is stored in "directive". Arguably the most important field of the data structure, "uri" stores the observed resource URI. Same-origin URIs are stored without the origin. This has a major advantage: When the proxy is changed to protect the same web application on a different domain, the only string which has to be changed is a setting value, describing the current origin of autoCSP. Rules are never deleted by the proxy but merely deactivated with the "activated" flag in the data structure. To avoid duplicate entries in the database each ("document_uri", "directive", "uri") tuple has to be unique. This is enforced on the database level.

In "violations" all violation reports of the *locked mode* are stored. They can be used to spot errors in the case CSPs still break website functionality despite all precautions. When a problem cannot be resolved automatically by the proxy, it will store a message in the "warnings" table. Both database tables are used to display information in the web interface (cf. Section 5.2.3).

5.1.3. Deployment

As a result of its design, the proxy is highly prone to attacks in *learning mode* (cf. Section 5.2). Thus, a two phase life cycle of the proxy may be advantageous. In the first phase, the proxy has to be protected from unauthorized access because it is deployed in *learning mode*. Access control can be achieved with the built-in HTTP Basic Access Authentication. Trusted users or automated user agents have to browse the proxied web application in this phase. By the time a good (or complete) coverage is reached, the *locked mode* can be activated. Finally, access to the proxied web application can be granted to all user agents. Note that the web application must be hidden behind the proxy at all times for the protection to be effective.

Already existing web applications require practical deployment strategies because access cannot simply be limited in *learning mode*. Consecutively, two deployment strategies will be described. The biggest risk of every deployment method is to expose malicious code to the proxy in *learning mode*. With this in mind, the optimal way of deploying the proxy is to use an isolated copy of the web application during the learning phase. Unchallengeable policies can be obtained with this method, making up for the effort which has to be put into creating a working copy of the website. A less preferred strategy is a parallel deployment. While leaving access to the web application open, the proxy can be deployed on a different origin. Although access to the proxy can be strictly limited, an adversary might be able to subvert the protection. If a persistent XSS vulnerability is actively being exploited, the trusted user agents, browsing the proxied web application, will trigger the payload. Thus, the proxy will whitelist the attacker's payload.

Summing up, the proxy should be only accessible to all clients in *locked mode*. If this requirement is fulfilled, design goal number two is not violated. All HTTP responses seen by regular clients are protected by a CSP.

5.2. Learning Mode

Learning mode is one of the proxy's two modes of operation. In this mode, the proxy tries to obtain strict CSPs. This policy generation strategy is described in Section 5.2.1. Furthermore, it externalizes inline code for use in the *locked mode* (cf. Section 5.3), as explained in Section 5.2.2. Details of the web interface are covered in Section 5.2.3.

If an adversary obtains access to the proxied web application in this mode, the protection of the proxy may be subverted. A successful content injection attack will not be mitigated by the proxy. Instead, it will lead to a compromised policy, allowing the attacker's payload to execute in all modes. This is because the proxy completely trusts the markup while in the *learning mode*. All markup in this phase is subject to policy generation and code externalization, albeit its origin. Therefore, a strict access control has to be in place when this mode is active.

Resources, which are not directly fetched from the proxy, can falsify results in this mode. This is because the proxy adds new headers to all resources, as explained in Section 5.2.1. These headers would not be obtained by user agents when they use resources from the cache. Caching is therefore disabled in *learning mode*. Table 5.5 shows the caching-related headers used to achieve this. Additionally, the "ETag" and "Last-Modified" headers from the web server's response are discarded to further prevent caching. However, this problem cannot be completely mitigated by using headers since user agents not receiving the headers was the problem in the first place. After all, the user agent's cache has to be cleared before browsing the web application in *learning mode*.

Header name	Value
Expires	-1
Pragma	no-cache
Cache-Control	no-cache, no-store, must-revalidate

Table 5.5.: Cache-disabling HTTP headers

5.2.1. Policy Generation

Policy generation is one of the two core mechanisms of the reverse HTTP proxy, the other one being inline code externalization (cf. Section 5.2.2). Amongst other possible approaches, the proxy's actual strategy for generating policies is described in Section 5.2.1.1. The implementation split into two parts: Section 5.2.1.2 explains the base policy generation and Section 5.2.1.3 a policy refinement mechanism.

Browsing the web application in this mode incrementally builds up rules for internal URIs which can be formed into a restrictive CSP in *locked mode*.

5.2.1.1. Strategy

Three policy generation strategies were considered for the proxy. It turned out that no approach is adequate on its own for the use case of autoCSP. Instead, a hybrid approach is presented at the end of this Section. Policy generation can generally be achieved by observing all resource URIs embedded in a document and their type. A document could, for example, contain an *image* from the URI "http://example.com/img.png". Apparently, this information is sufficient to categorize the resource into the correct CSP directive and whitelist the respective URI.

- A naive policy generation strategy would observe markup directly in the proxy to obtain the resource URIs. All traffic passing by the proxy could be scanned for markup. This markup could be parsed, using one of the multiple parsing[5] libraries[6] available for Python. Then, the constructed DOM may be traversed by an algorithm searching for resources and storing them in the database. In spite of being straightforward, this approach suffers from two major downsides. First, since parsing libraries do not implement a full browser stack, active content is not executed. Potential changes to the DOM at runtime, such as the addition of a resource, could not be observed. As a result, the generated policy may be incomplete. Secondly, parsing libraries might interpret the markup differently than browsers. If a mismatch happens, this may again lead to incomplete CSPs.

- Another policy generation strategy uses the Report-Only functionality of CSP to build policies. This solution is elegant: It makes good use of the trusted clients visiting the web application and leaves the interpretation of the markup completely to their user agents. Parsing mismatches are impossible (or at least for the used browser family), since the browser's own understanding of the markup is leveraged. Violation reports contain, amongst other information, the URI and the violated CSP directive of a resource. This is sufficient to generate a policy. Moreover, any future additions to the web platform will require no changes to the proxy's code if the format of the reports stays the same. Despite having many desirable traits, the Report-Only functionality is not optimal for the use case of the proxy. As explained in Section 4.3, violation reports lack file-level URIs in case of cross-origin resources. An analysis of the Alexa top 1000 most popular websites showed that 82 percent of the observed resources

[5] *Beautiful Soup*, `http://www.crummy.com/software/BeautifulSoup/`, Oct. 2013
[6] *html5lib*, `https://github.com/html5lib/html5lib-python`, Oct. 2013

were embedded from other origins (cf. Appendix A.1). To sum up, if relying on this policy generation strategy only, the proxy would not be able to create strict policies for many modern web applications.

- The last policy generation strategy considered for the proxy, was to use the browser itself. Instead of prompting browsers to provide the desired information through the Report-Only functionality, the proxy can inject a script into every document. Parsing mismatches are likewise impossible using this method because the script can use the DOM of the browser. Unlike violation reports this policy generation strategy enables the proxy to obtain file-level URIs of cross-origin resources. An injected script can simply traverse the DOM. Since all resource URIs are exposed by properties of the DOM or the CSSOM, they can be read by the script. However, there is a downside to this method. As all scripts of a document, the injected script is bound to the security boundaries the browser enforces. If a DOM property contains a cross-origin URI which only redirects to another URI, the script can neither follow the redirection nor reliably detect it. CSP requires both the redirecting URI and the target URI to be whitelisted. Therefore, this may again lead to incomplete policies.

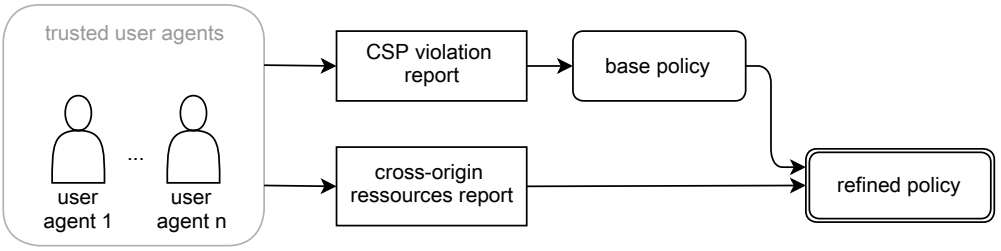

Figure 5.2.: Hybrid policy generation strategy

Due to the benefits of the latter two techniques, a hybrid strategy was chosen. A base policy is obtained by injecting a CSP Report-Only header into every served document and collecting all violation reports. These reports already contain file-level same-origin resource URIs. Instead of discarding the deficient cross-origin URIs, they will be saved to the database. An injected script tries to refine these cross-origin URIs by traversing the DOM. If the script does not succeed in doing so, the correctness of the policy still is ensured by the URIs obtained through the violation reports. A combination of both techniques can be used to reliably detect redirections (cf. Section 5.2.1.2). Figure 5.2 illustrates the general policy generation strategy of the proxy.

5.2.1.2. Base Policy Generation

Base policies are obtained with the help of CSP Report-Only headers. In *learning mode*, every resource will be served using this technique. While ignored for all static resources, the header is enforced in documents. As explained in Section 4.3, the browser will send violation reports to the sink specified with the `report-uri` directive. These reports can be used to gather the rules needed for a policy.

```
Content-Security-Policy-Report-Only:
    base-uri 'none'; connect-src 'none'; font-src 'none';
    form-action 'none'; frame-src 'none'; img-src 'none';
    media-src 'none'; object-src 'none'; script-src 'none';
    style-src 'none';
    report-uri /_autoCSP/_/report?id=Qr9T37QQyojjyU3R5wnyyk6Js7HbjDAH
```

Listing 5.1: CSP Report-Only header for unknown URIs

Two types of Report-Only headers are generated by the proxy in order to collect violation reports. Listing 5.1 shows the injected header for unknown document URIs. If a client visits a proxied URI which the proxy did not collect information about before, this policy will be used. Instead of whitelisting resources, the policy is disallowing almost everything. Accordingly, violation reports will yield detailed information about the violated directive. If just `default-src` would have been set to `'none'`, browsers would report the violated directive to be `default-src`. It may even be impossible to infer the correct CSP directive if this information is not offered by the report. However, the injected header ensures that reports will contain the correct CSP directive. A request ID is appended to the `report-uri` to uniquely identify all resource URIs gathered in the current request (cf. Section 5.1.2). For all document URIs known to the proxy, the policy of Listing 5.1 will be supplemented by already collected information. All observed resource URIs of the document will be whitelisted in subsequent CSP Report-Only headers. This suppresses violation reports for known resources on successive visits of the same URI. Note that this still enables the proxy to gather information about dynamically added scripts which might not have been observed on a previous visit of the document.

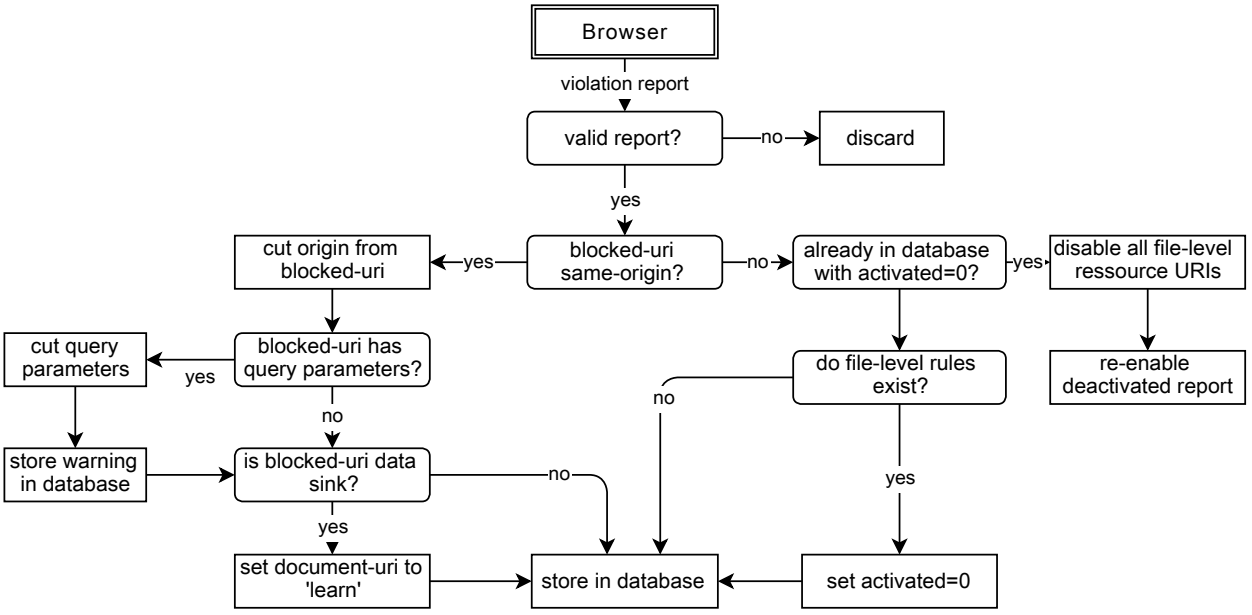

Figure 5.3.: Control flow of the *report sink*

When receiving a violation report, the the web interface will store the information in the data structure of Table 5.4. This part is called the *report sink*. Most fields of the data structure have been chosen to closely resemble the values of the violation report. Thus, "document-uri" of a report maps to "document_uri" in the data structure and the "violated-directive" will be stored in "directive". "blocked-uri" is the essential information the proxy wants to collect and it is copied to the "uri" field. Figure 5.3 illustrates the control flow, which is explained in the following paragraphs.

First, the algorithm deals with all sorts of malformed data and discards violation reports with insufficient information. Such reports are sent, when, for example, inline code violates the injected header. These reports lack a proper "blocked-uri" value. Then the algorithm branches based on the origin of the blocked URI.

Same-origin URIs are first checked for query strings. Query parameters are ignored by CSP (cf. Section 4.2) and thus can be removed from the URI. Since they mostly denote dynamic resources with content that is changing according to the parameters, they can even pose a security risk. If the query parameters get reflected in the output without proper sanitization, there is the risk of allowing a potentially unsafe script in the generated CSP. Therefore, a warning is stored in the database, when a query string is observed. If a resource of the web interface is encountered (cf. Section 5.2.3), the document URI of the data structure is set to "learn". All rules with this document URI will be whitelisted in the Report-Only headers, but only in *learning mode*. This way, injected resources from the back end, as used by the policy refinement mechanism, are handled gracefully by the proxy and do not need any hard coding in the source code. Note, that all document URIs normally start with a slash, so that the special value cannot interfere with regular document policies. At the end of the execution path, the data structure is stored in the database.

Cross-origin URIs have to be checked more thoroughly. The *report sink* generates base policies, meaning that cross-origin URIs will be refined by another sink afterwards (cf. Section 5.2.1.3). All cross-origin URIs of the *report sink* will be disabled when a refinement is conducted. In the case that the refinement was correct, no violation report of the cross-origin URI will be sent on subsequent visits of the same document. This is because the CSP Report-Only header will be filled with the already observed URIs. However, if these refinements yield an incomplete policy, the browser will send a violation report. In this case the *report sink* has to undo the policy refinements. For this reason, the proxy first queries the database for a disabled record with the same ("document_uri", "directive", "uri") tuple but a different request ID than the current one. If a database record was found, the *report sink* infers multiple facts from this: First, the document has already been visited by a browser and a policy has been stored in the database. This can be concluded from the fact that the database record has a different request ID. Secondly, the policy must be incomplete because a violation report was sent to the sink. Thirdly, since there exists a disabled rule, the policy refinement must have broken the policy. To fix this, all rules with file level URIs, matching the reported origin, document URI and violated directive, will be deactivated. Additionally, the disabled rule will be reactivated. If no disabled record with a matching tuple was found in the first place, the proxy will check for a race condition. It might happen, that the refinement takes place before the base policy is set. This can be detected by querying the database for file-level URIs with the same origin, request ID, directive and document URI. A matching request ID would indicate that the refinements were indeed just added. Instead of discarding the report in this case, it still is stored in the database, but with a disabled status. If no such record is found, the

data structure is saved without changing the status.

5.2.1.3. Policy Refinement

Element name	Directive	Action
*	img-src	If available, get background images
APPLET	object-src	Get `code`, `archive`, `codebase` attributes
AUDIO	media-src	Get `src` attribute
BASE	base-uri	Get `href` attribute
BUTTON	base-uri	Get `formaction` attribute
EMBED	object-src	Get `src` attribute
FORM	form-action	Get `action` attribute
FRAME	frame-src	Get `src` attribute
IFRAME	frame-src	Get `src` attribute
IMG	img-src	Get `src` attribute
INPUT	form-action	Get `formaction` attribute
LINK	img-src	If icon, get `href` attribute
LINK	style-src	Trigger style check
OBJECT	object-src	Get `data` attribute
SCRIPT	script-src	Get `src` attribute
SOURCE	media-src	Get `src` attribute
STYLE	style-src	Trigger style check
TRACK	media-src	Get `src` attribute
VIDEO	media-src	Get `src` attribute

Table 5.6.: Nodes visited by *policy.js* and associated actions

Policy refinement is conducted in two steps: First, a script is injected into every served document. This script will be referred to as *policy.js*. It will send reports with observed cross-origin URIs to the *policy.js sink*. Secondly, the sink will store the URIs and disable all cross-origin URIs obtained with the base policy generation.

Only policies of resources with “text/html” and “application/xhtml+xml” MIME-types are refined, regardless of charset. The *policy.js* script is injected into these documents. It should be the very first script on the page so that it can hide itself from other active content on the page. Two regular expressions are used to find either the end of the opening head-tag or the beginning of the opening title-tag. If both tags are not present in the markup the script will just be prepended to the document’s code, risking to put visiting browsers into Quirks Mode[7]. An attribute containing the request ID will be added to the first script tag, making it possible for the script to obtain the request ID. After reading out this attribute, the script immediately

[7] L. H. Silli, *No condition comments before the DOCTYPE*, `http://xn--mlform-iua.no/blog/no-condition-comments`

deletes itself from the DOM. All code is wrapped in an anonymous function to prevent leaking variables into the global execution scope. These precautions are taken because otherwise *policy.js* might interfere with other scripts on the website, possibly changing behavior.

After the "load"-event of the window object, the injected script traverses the complete DOM. While visiting all nodes, it executes the actions presented in Table 5.6. All future changes to the DOM are perceived by a Mutation Observer [39], which will call the according actions, too. Most actions only return an attribute of the currently traversed node, containing a resource URI. Other functions check for conditions to be true before returning a value: Since `link` elements can have many different functions [26], one action first checks if it contains an icon, before returning the resource URI. Other uses of the `link` element include embedding style sheets. Due to the ability to import other style sheets from within a style sheet (using the `@import` statement), a more elaborated check has to be performed. A recursive algorithm checks the CSSOM for yet unknown style sheets. In detail, it reads all style sheets and recursively follows all `@import` statements until no more can be found. The first action listed in Table 5.6 applies to all elements. Therefore, every node will be checked for having a background image.

As can be seen from the Table, `connect-src` rules are not inferred from the DOM. Instead, they are directly obtained by overwriting all JavaScript APIs which can violate the directive. In consequence, the *XMLHttpRequest* and *WebSocket* APIs have been changed to report cross-origin destinations to the *policy.js sink*. This happens transparently for the calling code.

Whenever the injected script finds cross-origin URIs, it sends a JSON-encoded data structure to the *policy.js sink*. An exemplary data structure is shown in Listing 5.2. As can be seen from the example, resources are directly categorized into CSP directives. Furthermore, each report of the injected script is able to transport multiple cross-origin resource URIs.

```
{"script-src":["http://a.com/first.js","http://b.com/second.js"],
 "img-src":["http://a.com/image.png"]}
```

Listing 5.2: JSON-encoded data structure used for communication between *policy.js* and its sink

After receiving this data structure in the *policy.js sink*, the proxy will scan it for errors. If the data is malformed, it is discarded. Each received URI will be written to the database separately if it is not a duplicate. Then, an attempt is made to refine already existing policy rules. If the database contains records with the same "document_uri" and "directive", their "uri" will be compared to the origin of the each stored URI. If a match occurs, it is highly likely that the record contains an URI gathered with the CSP Report-Only header. As the matching policy rule does not have a file-level URI, it is disabled by the *policy.js sink*. There is the possibility that this is leading to incomplete policies. These are detected and resolved in the *report sink* (cf. Section 5.2.1.2).

5.2.2. Code Externalization

CSP gains its effectiveness against XSS from disallowing inline code and eval-like constructs (cf. Section 4). This poses a problem to the code bases of most web applications today. An analysis presented in Appendix A.1 shows that 96 percent of the Alexa top 1000 websites use CSP-incompatible code patterns.

Section 5.2.2.1 describes the proxy's general strategy for code externalization. Then, a detailed description of the inline style sheet externalization is given in Section 5.2.2.2 and the inline script externalization is explained in Section 5.2.2.3. Section 5.3.2 explains, how the externalized code is embedded and triggered in *locked mode*.

5.2.2.1. Strategy

Code externalization is achieved with an injected script. This script will be referred to as *externalize.js*. Similar to *policy.js* (cf. Section 5.2.1.3), the script is injected into all resources with a "text/html" or "application/xhtml+xml" MIME-type, regardless of character set. In *learning mode*, it will gather all inline code from a document. The code will be hashed using the SHA256 algorithm and compared to a list of hashes of already externalized code. As it is sent to a sink of the web interface afterwards, this avoids transferring already externalized code again.

Inline code patterns differ from each other. More importantly, the security requirements differ, too. An inline event handler, for example, must be distinguishable from an inline style attribute in the database. Since both are completely different languages, they have to be embedded differently into the document in *locked mode*. Furthermore, while an attacker may inject the same CSS code a second time into the document, this might not always be safe for scripts. An attacker might be able to place elements with event handlers in such a way that they change the behavior of the site to his advantage. For this reason, a type is attached to every externalized code snippet. All types are described in Table 5.7.

Type	Description
css	Inline CSS code
css-attr	Inline style attributes
js	Inline JavaScript code
js-event	Inline event handler
js-link	Inline JavaScript link

Table 5.7.: Externalized code types

While the proxy is capable of rewriting most inline code, there are cases which cannot be automatically rewritten. These shortcomings are often revealed by dynamic behavior on client- and server-side which obscures the intent of the web application to the proxy. Inline code can be hard to rewrite when server-side logic dynamically injects data into the code. An example of this would be a user name which gets directly written into a script. Each user would obtain a different inline script, making an externalization impossible. Furthermore, most strategies of rewriting dynamic code could introduce huge security problems: One strategy could be to detect similarities between scripts and execute inline code which differs only in safe locations. CSP 1.1 nonces could be used to allow the inline code. However, detecting these safe locations can only be done if the proxy fully understands the intent and consequences of all differences. This clearly is not a simple task and out of scope for autoCSP.

5.2.2.2. Inline Style Sheet Externalization

Two variants of style sheets are incompatible with a strict CSP, the first one being a `style` element. Style attributes are the second anti pattern. Both types of inline CSS are explained in Section 3.4.

Element name	Type	Action
*	css-attr	If existent, get style attribute
STYLE	css	Get content

Table 5.8.: Nodes visited by *policy.js* for style sheet externalization

As the injected script traverses the DOM, it executes the actions shown in Table 5.8. Every node is checked for having a style attribute. When a style attribute is detected, it will be hashed and compared to a list of already externalized code. If a match occurs, the style attribute was already externalized or is identical to another style attribute in the document. As explained in Section 5.3.2.1, identical style attributes are always mapped to the same class, which is why they do not need to be externalized twice. Any `style` node the script encounters, will be externalized just as style attributes.

5.2.2.3. Inline Script Externalization

There are three variants of inline script which have to be externalized by the proxy to retain the functionality of the protected web application. Section 3.5 explains inline `script` elements, inline event handlers and the JavaScript pseudo protocol.

Element name	Type	Action
*	js-event	If existent, return event handlers one by one
A	js-link	If starting with “javascript:”, return href attribute
SCRIPT	js	If src attribute not set, get content

Table 5.9.: Nodes visited by *policy.js* for script externalization

Table 5.9 summarizes the actions *externalize.js* takes to obtain all three types of inline script. Every node is checked for inline event handlers. In contrast to inline style sheets, the hash does not only contain the event handler code. Instead, the event name and the node position are prepended to the source code before hashing. Node positions are the chain of nodes which are traversed, when moving from the node with the event handler up to the root of the DOM tree. The node position of the `span` element in Listing 5.3, for example, would be `span/div/body/html/#document`. If this node would be externalized, the complete hashed string would therefore be `click,span/div/body/html/#document,code()`.

```
<html>
    <body>
        <div>
            <span onclick="code()">Example</span>
        </div>
    </body>
</html>
```

Listing 5.3: HTML document presenting an onclick handler

Anchor elements are scanned for URIs, starting with the string “javascript:”. This string indicates a JavaScript pseudo protocol URI. Before externalizing `script` elements, they will be checked for a `src` attribute as it indicates an external script. If no such attribute is found, the content of the `script` element will be externalized.

5.2.3. Web Interface

In *learning mode*, the proxy exposes a web interface. It features a general overview of observed document URIs with subpages which show more details for an URI. Furthermore, an export function is available to retrieve all generated policies from the proxy’s database. This aids developers, only using the proxy for its policy generation capabilities, in using the information for the framework or language of their choice. A red color scheme is used in the header of the web interface to indicate that *learning mode* is not safe for unauthorized access.

Static files, templates and so-called “View” functions are the main components of the web interface. “View” functions control the logic of the interface and use templates to display information about the policy generation process to the user. All pages of the web interface are protected by restrictive CSPs, so that they are not the weak point in the web application’s protection. In contrast to interceptors, “View” functions are only accessible in *learning mode*. *Locked mode* access has to be enabled separately (cf. Section 5.3.3).

Policy rules can be updated and disabled in the web interface. This enables users to manually refine policies. If the proxy detects potential security problems (cf. Section 5.2.1.2), it stores a warning in the database. These warnings are shown in the web interface, too.

5.3. Locked Mode

In *locked mode*, the proxy tries to enforce strict CSPs for all served resources. Section 5.3.1 describes the policy enforcement algorithm. Since strict CSPs block inline code, the proxy externalizes such code in *learning mode* (cf. Section 5.2.2). This code is embedded into the respective documents and triggered by an algorithm explained in Section 5.3.2. A web interface can be used to monitor the effects of all enforced CSPs. An description of it is given in Section 5.3.3.

5.3.1. Policy Enforcement

Satisfying the main design goal of the proxy, strict CSPs are enforced on every served resource in *locked mode*. Each visited document has a unique document URI, which is used to obtain the correct policy rules from the database (cf. Section 5.1.2). As these rules already have a CSP directive assigned to them, the proxy only has to group resource URIs together. Due to the fact that the trusted clients are not required to browse the complete web application in *learning mode*, there might be documents which are not yet known to the proxy. This also applies to documents, which were created during *locked mode*. Since the proxy never generated a policy for these resources, it cannot use a correct CSP. Instead, unknown document URIs will be protected with the strongest possible policy, shown in Listing 5.4. While this violates another

design goal, stating that the proxy must ensure to not break websites, it is the only secure option. If not all resources of a web application are protected by a strict CSP, the adversary might succeed in attacking one of the unprotected resources. Then, client-side code execution in the context of the web application could be obtained. No security boundaries could prevent information extraction in that case anymore. This would lead to a total bypass of the proxy's protection.

However, there is a special case. If the unknown document URI contains a query string (denoted by a question mark), another database lookup is attempted. For this lookup, the query string is ignored. For instance, the URL `http://url/?param=val` will be reduced to `http://url/`. This heuristic assumes that the query string indicates a subpage. Thus, the base page policy could retain the functionality of the document.

```
Content-Security-Policy: default-src: 'none'; base-uri: 'none',
                         form-action: 'none';
```

Listing 5.4: CSP for unknown document URIs

If the web interface is enabled, it is able to monitor CSP violation reports. To enable monitoring, the enforced CSP is complemented by a `report-uri` directive. All reports are processed by a *data sink* of the web interface. Malformed reports will be discarded, others are directly stored in the database.

5.3.2. Triggering Externalized Code

In *learning mode*, inline code is stored in a database. This code must be embedded and triggered in *locked mode*. Section 5.3.2.1 describes the way inline style sheets are embedded. Triggering externalized scripts is covered by Section 5.3.2.2. Calls to *eval*-like constructs have to be intercepted in this mode, which is explained in Section 5.3.2.3.

Similar to *learning mode*, the proxy injects a script in *locked mode*, triggering the externalized code. These scripts are served by the proxy and contain the externalized code. Furthermore, their URI contains the full path to the protected document. For instance, a script, injected into a document with the URI "/p/f.html", is served at "/autoCSP/_/inline/p/f.html.js". As CSP 1.1 allows file-path URIs, the proxy makes use of that with this approach. In consequence, an attacker is not able take advantage of a different document's inline script.

5.3.2.1. Externalized Style Sheets

When visiting a document in *locked mode*, the database will be checked for externalized CSS code. If such code is found, a style sheet is injected into the document. Externalized `style` elements are output without change into this style sheet. Unlike style attributes, the code of `style` elements is proper CSS. Style attribute code, however, is lacking a selector. This means that the proxy must find a way to assign the CSS code to the correct nodes. In order to achieve this, a script is injected into the document, too. While traversing the DOM, it scans all nodes for style attributes. If such an attribute was found, the content is hashed with SHA256. This hash is prefixed with the string "autoCSP" and added to the classes of the node.

Cryptographic hash functions always yield the same result on the same input. Using this property, the proxy is able to add selectors to externalized style attribute code. Since the hash is stored in the database, alongside with the code, the proxy only has to concatenate the strings in the correct order. A beneficial property of this technique is that two completely identical style attributes will always result in the same class, reducing the overall count of additional classes.

As explained in Section 3.4, CSS declarations have a precedence. Style attributes have the highest precedence, only surpassed by declarations marked with the `!important` keyword. This behavior must be closely copied by the proxy to keep the layout and look of the web application intact. Hence, all semicolons are replaced by " !important;". This marks all declarations of every externalized style attribute as important, giving them a high precedence. Since the injected CSS is the first style sheet in the document, the declarations can still be overwritten by other declarations marked as important. Tests with Firefox 25 and Chrome 30 showed that this closely resembles the original behavior of style attributes. Listing 5.5 shows an exemplary behavior test. It is possible to omit the last semicolon of a style attribute. A regular expression is used to add this semicolon in this case.

```
<style>
div {
    color: red !important;
    background: red !important;
}
</style>
<style>
div {
    color: blue; /* is ignored */
    background: blue !important; /* overwrites the red color */
}
</style>
<div>test</div>
```

Listing 5.5: Test of the cascading behavior of CSS with the `!important` keyword

The injected style sheet has to deal with relative URIs, too. This is a problem when it is deployed from the web interface because browsers will resolve relative URIs using the style sheet's location as the base URI. Therefore, the proxy intercepts all requests to resources ending with "autoCSPinline.css", cutting the suffix and using the prepending string for the database lookup. For example, the style sheet injected into the document at the URI "/path/file.html" would point to "/path/file.htmlautoCSPinline.css" with the document URI "/path/file.html". This might hinder the web application from serving a resource with the name "autoCSPinline.css". To prevent this, the proxy does not intercept these requests when no inline CSS is found in the database for the document URI.

5.3.2.2. Externalized Scripts

In contrast to style sheets, externalizing scripts is not straightforward. Code in `script` elements can be highly position dependent and rely on DOM nodes to be available at execution time. Furthermore, the order of executed scripts matters, as one script might rely on changes done by another. In general, there are three types of externalized code, which have to be triggered in *locked mode*. A script is injected into every resource, for whose document URI externalized code can be found. Similar to style sheet externalization, the DOM will be traversed to set up the code and trigger it. Furthermore, the proxy will wait until the "DOMContentLoaded" event, before traversing the DOM and triggering any code. This event occurs, when the browser finished parsing the markup and the DOM tree is complete. All DOM nodes should be available to the externalized scripts after this event.

```
var inlineScripts = {
    '6e11c72f7cf6bc383152dd16ddd5903aba6bb1c99d6b6639a4bb0b838185fa92':
        138,
    // ...
}
```

Listing 5.6: Object storing externalized script IDs

Externalized scripts are not injected directly into the protected document. Instead, a SHA256 hash of the source and its database ID is stored in a JavaScript object, shown in Listing 5.6. When the DOM-traversing script encounters a `script` node in the DOM, it will check for inline code (cf. Section 5.2.2.3). If this is the case, the content of the `script` node will be hashed and compared to the aforementioned JavaScript object. Since a match means that the inline script was externalized in *learning mode*, a `script` node is then added to the DOM. The script points to a URI of the proxy, serving the correct script (which is identified by its ID).

```
var eventHandlers = {
    '522c8a4aa5a415d1050a23470fd41f4587cec2bc45655b9c4ffe90890ea42e64':
        function() { with(this) { alert(1) } },
    // ...
}
```

Listing 5.7: Object storing externalized event handler code

Inline event handlers are encapsulated in functions, as shown in Listing 5.7. These functions are stored in a JavaScript object with the key being the SHA256 hash of the externalized code. A `with` statement enables access to the properties of an object without having to expressively write it. As inline event handlers can access the properties of the node they were defined on in the same way, this only copies the original behavior of the externalized code. A regular expression is used to identify event handlers while traversing the DOM (`/^on([a-z]+$)/i`). If a matching attribute could be found, it will be hashed in the same way it was hashed in the *learning mode* (cf. Section 5.2.2.3). Event name, node path and source code are concatenated and hashed. If a match can be found in the object holding the externalized event handlers, the corresponding

function will be bound to the correct node. This sets the `this` variable to the node, the event handler is be added to in the next step. Naturally, the injected script sets the event handler in a CSP-compliant way.

Two event handlers have to be treated in a special way. “onload” and “onerror” events are triggered, when a resource completed loading. Due to caching, the resource might have loaded before the traversing script adds the event handler to the respective node. Therefore, the `src` attribute is reset, prompting another load of the same resource. In the worst case, this triggers another request to the server.

JavaScript pseudo protocol links are stored like externalized inline event handlers, but without the `with` statement. Externalizing the links in a CSP-compatible way is possible with an event handler reacting to clicks. Therefore, the injected script traverses the DOM and checks every anchor for containing the JavaScript pseudo protocol. If such an anchor is found, its node path and source code is hashed. This hash is then compared to the JavaScript object storing the externalized code. A “click” event handler, containing the externalized function, is then added the matching node.

5.3.2.3. Eval-like Constructs

A strict CSP will prevent *eval*-like constructs from executing. All calls to these functions are intercepted by an injected script in *locked mode*. Then, heuristics are employed to detect JSON input. If JSON could successfully be detected, the proxy will replace the original function call with *JSON.parse*. This may retain the functionality of the original code. However, all *eval*-like constructs used for different use cases than JSON parsing cannot be rewritten by the proxy. The proxy would have to understand the intent of the code to suggest replacements, which is out of scope for this thesis. Especially dynamic code building is affected by this shortcoming of the proxy. If the web interface is enabled in *locked mode*, a violation report will be emitted by the injected script. This allows developers to manually fix the problem.

```
[,,,1,,2,,,"value"]
```

Listing 5.8: Non-standard JSON breaking *JSON.parse*

Although most JSON can be detected and parsed, the *JSON.parse* API is much stricter than *eval*-like constructs. Non-standard JSON, as shown in Listing 5.8, leads to exceptions, breaking code which would run correctly using eval. Another heuristic tries to detect such cases and adds `null` keywords in between the commas, where needed.

5.3.3. Web Interface

The web interface will provide information about policy violations in *locked mode*. These monitoring capabilities can be used to find documents with incorrect CSPs. Despite the efforts of the proxy to retain the web applications functionality, strict CSPs may still break it in certain circumstances (cf. Section 5.3.1). Showing violation reports allows developers to manually fix problems. All reports are ordered by the document URI they were collected from. Duplicate entries are prevented on a database level. A green color scheme is used in the header of the web interface, indicating that the web application is currently protected by the proxy.

In this mode, protecting the web interface with HTTP basic access authentication is mandatory. Additionally, access to the web interface has to be verbosely activated in the proxy's settings file. All these requirements are meant to prevent leaving access open to attackers. While the web interface is limited to monitoring in *locked mode*, an adversary might still obtain information from the displayed violation reports. Critical information might be a document URI pointing to a secret resource or similar. "View" functions are not enabled by default in *locked mode*. They have to be activated separately, again forcing a intentional decision. Generally, no changes to the database are allowed in this mode, apart from storing violation reports. This makes it substantially harder (or even impossible) for adversaries to inject malicious policy rules and subvert the proxy's protection.

6. Evaluation

An evaluation of the proxy under two aspects is given in the following sections. First, the proxy's ability to mitigate content injection attacks was tested. Secondly, the impact of the protection on the web application's functionality was observed. Section 6.1 outlines two test suites, both focusing on exactly one aspect of the evaluation. A combination of both aspects is tested in Sections 6.2 and 6.3, where two prototypical vulnerable web applications are deployed behind the proxy. In order to find out the proxy's applicability to real world web applications, a web mailer is tested in Section 6.4.

6.1. Test Suites

Two very different test suites are described in this Section. Section 6.1.1 describes a test suite, which was built in the course of writing the proxy. Section 6.1.2, on the other hand, outlines an evaluation of the proxy's protection using a DOM-based XSS test suite by M. Heiderich.

6.1.1. Test Suite of the Proxy

In the course of implementing the proxy, various edge cases and problems surfaced. Every time such a problem was fixed or the proxy learned a new CSP directive, test cases were added to its suite. All test cases were categorized by the CSP directives they belong to and were given descriptive names. The test suite features 44 samples, associated to 9 CSP directives, and a total of 59 files.

```
<!DOCTYPE html>
<title>inline script adds inline event handlers</title>
<script>
    // possibility #1
    document.write('<img src="x" onerror="alert(1)">')

    window.addEventListener('load', function() {
        // possibility #2
        document.body.innerHTML = '<img src="x" onerror="alert(2)">';
    });
</script>
```

Listing 6.1: Exemplary test case presenting two ways of adding inline event handlers

Listing 6.1 shows one of the test cases. It is called “inline script adds inline event handlers” and demonstrates two ways of adding inline event handlers from inline JavaScript code. The proxy should externalize the inline JavaScript block correctly and detect all changes to the DOM. Then, all event handlers should be replaced by their externalized counterparts. Naturally, the proxy passes all test cases in the suite.

6.1.2. DOM-based XSS Test Suite

A test suite by M. Heiderich was used to evaluate the proxy’s protection regarding DOM-based XSS. Three vulnerabilities were exploited in *locked mode* to observe the mitigation effect. They are enumerated in Listing 6.2. Many of these vulnerable code snippets use the JQuery[1] library. A knowledge of this library is not needed to understand the attack vectors and the results. This evaluation focuses on the protection aspect only, as no notable functionality is exhibited by the test suite.

```
// vulnerability #1
$(location.hash)
// vulnerability #2
$('#search').html('You searched for: '+
                  $('#search-text').attr('value'));
// vulnerability #3
dispMonth = eval (document.cookie.substring(countbegin,countend));
```

Listing 6.2: Vulnerabilities of the DOM-based XSS test suite relevant for the evaluation

- The first vulnerability can be exploited by a simple content injection in the URL hash. Due to the enforcement of a strict CSP in *locked mode*, inline JavaScript and CSS is useless for an attacker. Other means of obtaining data leakage, like dangling markup injection attacks, are not effective because of the strict limitations to the document’s capabilities. It is not able to send data using forms or similar. A static content injection is still possible, but unlikely to result in a data leakage.
- Vulnerability number two is exploitable by filling the `input` element, identified by the ID “search”, with a XSS vector. In essence, the mitigation is identical to the one of the first flaw. Both code snippets output attacker-controlled values directly to the DOM of a document. As a matter of course, the proxy manages to prevent a meaningful attack.
- In the third test case, an *eval* statement is used to parse a cookie value. An attacker might be able to inject cookies into the web application to exploit this. For testing purposes, this is not important. Regardless of the attack scenario, the proxy must be able to mitigate this weakness in order to offer a complete protection. In *locked mode*, *eval*-like constructs are disabled and only rewritten, if they receive JSON. Thus, any possible attack is mitigated, but the functionality of the code snippet cannot be preserved.

[1] *jQuery*, `http://jquery.com/`, Oct. 2013

6.2. Google Gruyere

In order to evaluate the protection and usefulness of the proxy, Google Gruyere[2] was deployed. As Gruyere is deliberately vulnerable, the proxy had to prove that it is capable of defending flawed web applications. Furthermore, the ability to retain the functionality and look of a website was tested. Six XSS vulnerabilities of Gruyere were used for the evaluation. First, the vulnerable documents were visited in *learning mode* (if possible). Then, *locked mode* was activated and the vulnerabilities were exploited. All used XSS vectors utilized the *alert* function of JavaScript to give a clear indication of whether the attack was successful or not. Finally, the functioning and look of the web application was checked for flaws, which could have been introduced by the proxy.

A detailed description of the testing process is given in Section 6.2.1. Section 6.2.2 summarizes the findings.

6.2.1. Testing Process

The first XSS vulnerability can be found in Gruyere's upload function. An attacker is able to upload arbitrary files, and hence HTML documents with malicious payload. As the URI of the uploaded document changes according to the file name, the proxy is not able to generate a policy for all future uploaded files in *learning mode*. However, since a strict policy is used for unknown document URIs in *locked mode*, the proxy successfully mitigated the attack. Furthermore, no core functionality of the web application was lost because uploaded files were still viewable (but without active content).

A reflective XSS flaw can be found in one of Gruyere's error pages. Each time the error document for missing resources is displayed (HTTP status 404), it outputs the URI. An exemplary attack appends the malicious payload to the URI, like this: `http://gruyere/<script>alert(1)</script>`. Again, the URL is not static and changes constantly, so that the proxy cannot observe a policy for all future visits. While the attack was successfully mitigated in *locked mode*, the styling of the error page could not be retained due to its changing URL. In contrast to the file upload mechanism, this actually is highlighting a downside of the proxy.

Gruyere allows users to store snippets. Due to improper sanitization, malicious payloads can be saved, too. One example is the payload `<img src=x onerror=alert(1)>`, which was used for testing. After all inline code was externalized and all policies were generated in *learning mode*, the proxy was able to mitigate the attack. Furthermore, the functionality and style of the web site could be retained. As a result of that users were still able to store new snippets. Only active content was blocked by the enforced CSP.

Another persistent XSS vulnerability can be found in the color value setting of the profile. It enables users to customize the color of their nickname. However, an adversary might escape from the inline style attribute this value gets output to and inject his malicious payload. `' onmouseover=alert(1)` is an exemplary payload achieving this. In *locked mode*, the enforced CSPs successfully thwarts this attack. Nickname colors are only preserved, if they were already observed in *learning mode*. Colors are output into an inline style

[2] B. Leban, M. Bendre and P. Tabriz, *Web Application Exploits and Defenses*, `http://google-gruyere.appspot.com/`, Oct. 2013

attribute, so that the proxy will compare them to the externalized CSS it collected. If extensive usage of this feature is conducted in *learning mode*, the proxy can build a whitelist of allowed colors. Therefore, this counts as a test case in which the functionality was fully retained.

A DOM-based XSS vulnerability can be found in Gruyere's "Refresh" function. It retrieves JSON-encoded data from the server and parses it with *eval*. An attacker can escape from the JSON data structure with a payload like this `"-alert(1)-"`. As *eval*-like constructs are disallowed in *locked mode*, the vulnerability could not be exploited anymore. However, since the returned JSON is enclosed by a function call, the proxy was not able to rewrite it. Listing 6.3 shows the content of the JSON response, which is directly fed into the *eval* function.

```
_feed(["val1","val2","val3"])
```

Listing 6.3: JSON response, enclosed by a function call, which is parsed with *eval* in Gruyere's "Refresh" function

The last XSS vulnerability is reflective. It can be found in Gruyere's JSON API, where an attacker is able to use a query parameter for the injection (`[...]?uid=<script>alert(1)</script>`). Since this document is normally loaded via the XMLHttpRequest API, the proxy does not build a policy it in *learning mode*. In consequence, the attack was successfully mitigated in *locked mode* because a very prohibitive policy is employed on unknown document URIs. As this is the same JSON sink as shown in Listing 6.3, the functionality of the page is still broken due to the enclosing function call.

6.2.2. Summary

Six XSS vulnerabilities of Google Gruyere were tested under two aspects. First, the mitigation capabilities of the proxy were checked in order to evaluate the protection of the proxy. Secondly, the protected documents were tested for broken functionality, introduced by the proxy.

On the one hand, the proxy was able to mitigate all of the six XSS flaws. An attacker is not able to exploit these vulnerabilities, if the *locked mode* is activated. On the other hand, only three of six test cases led to a fully working web application in *locked mode*. One of the test cases had inline style sheets which could not be externalized correctly, while two others used a nonstandard form of JSON in an *eval* statement.

6.3. Damn Vulnerable Web Application

Damn Vulnerable Web Application (DVWA) demonstrates multiple security vulnerabilities in a live environment[3]. It was purposely built in a flawed way and serves educational purposes. All vulnerabilities are available in low, medium and high security levels. Tests were conducted with version 1.0.8 of DVWA, which features ten classes of flaws. Two of these were used for the tests because they were content injection vulnerabilities.

Section 6.3.1 describes the testing process and Section 6.3.2 summarizes the results.

[3] *Damn Vulnerable Web Application*, `http://www.dvwa.co.uk/`, Oct. 2013

6.3.1. Testing Process

```
// low reflected XSS
echo 'Hello ' . $_GET['name'];
// medium reflected XSS
echo 'Hello ' . str_replace('<script>', '', $_GET['name']);
// high reflected XSS
echo 'Hello ' . htmlspecialchars($_GET['name']);
```

Listing 6.4: Relevant code parts of the three reflected XSS security levels of DVWA

First, the reflected XSS flaws of DVWA were tested. Listing 6.4 shows the relevant code of the three reflected XSS security levels in ascending order. While the low and medium flaws are vulnerable to almost any potential XSS vector, the high security reflected XSS sample does not seem to be exploitable at all. An attacker can, for example, inject the vector `<script>alert(1)</script>` into a query parameter of the web application. In *learning mode*, the proxy observed four resources and externalized 19 inline event handlers. Two of the 19 event handlers were used to display pop-ups and the remaining 17 for navigation purposes. In *locked mode*, this retained the document's original functionality almost completely, while mitigating all attacks. Only the pop-up function, called in two of the event handlers, broke, due to the use of an *eval* statement. Listing 6.5 shows the problematic code. As the proxy does not understand the intent of the code snippet, it cannot securely rewrite the code. It should be noted, that this code can be rewritten to an *eval*-free form by a human.

```
eval("page" + id + " = window.open(URL, '" + id + "', /* ... */);");
```

Listing 6.5: Problematic call to *eval* in DVWA's pop-up function

DVWA's stored XSS vulnerabilities use the exact same vulnerable code as presented in Listing 6.4, even though the attack vectors are saved in a database. The flawed document contains two input fields which can be used to add entries to a guest book. In *learning mode*, the proxy observed four resource URIs. Additionally, 21 inline event handlers were externalized. A total of 17 thereof are used by DVWA for navigation purposes, while the others are utilized to open pop-ups (2) or validate the HTML form (2). Using this policy, the proxy managed to mitigate all XSS attacks in *locked mode*. More interestingly, the proxy managed to retain most of the functionality of the site. All event handlers executed correctly. Again, only the call to *eval* could not be successfully rewritten.

6.3.2. Summary

Four XSS vulnerabilities of DVWA were tested in order to evaluate the mitigation capabilities of the proxy. Additionally, the original functionality of the web application was compared to the functionality in *locked mode* to detect potential problems.

Similar to the analysis of Google Gruyere (cf. Section 6.2), the proxy managed to mitigate all vulnerabilities. Most of the functionality of the web application could be preserved, apart from a call to *eval*. This,

again, resembles closely the results of the Gruyere analysis.

6.4. Roundcube Mail

Roundcube[4] is a web mailer – it features E-Mail client capabilities in a web application. Furthermore, it offers a rich interface, making heavy use of JavaScript for navigation and data retrieval. Therefore, it was chosen for yet another test of the proxy's abilities to retain functionality.

While the complete style of the web application could be retained, a conceptual problem of the proxy prevented JavaScript code from running. The main page of the web mailer featured a dynamically changing script. In particular, a request token changed at every new login and was reflected into an inline script. Since the DOM-traversing script could not recognize the changing script in *locked mode*, it was not replaced by externalized code. Allowing changes to the code in *locked mode* (as needed if it is dynamically changing) may give an attacker the possibility to manipulate the control flow of the web application. Obviously, this might result in security problems. Therefore, the proxy avoids this at the cost of breaking web applications in such cases.

[4] *Roundcube - Free and Open Source Webmail Software*, `http://roundcube.net/`, Oct. 2013

7. Conclusion

This thesis introduced a reverse HTTP proxy, capable of protecting arbitrary web applications with CSP. It infers strict policies from the observed markup and externalizes inline code automatically. For the purpose of making documents compliant to strict CSPs, it attempts to rewrite calls to *eval*-like constructs. All capabilities were derived from a set of design goals. Additionally, a definition of a strict CSP was given, clarifying the requirements needed to obtain a solid protection from data leakage attacks.

An evaluation of the proxy was conducted, in order to estimate its effectiveness against data leakage and XSS payloads. Additionally, the ability to retain website functionality was tested. In summary, the chosen policy generation strategy proved to be robust, while problems arose in the mechanisms making arbitrary documents CSP-compliant. Particularly calls to *eval*-like constructs could often not be rewritten by the proxy. This clearly indicates that these constructs are used regularly for other use cases than JSON parsing. Future research needs to find better approaches to automatically rewrite *eval*-like constructs in a secure way.

Many problems of the proxy can be attributed to its conception. Due to being application agnostic and treating the server-side implementation as a black box, it cannot securely detect similar documents or externalize dynamically changing code. Future work may explore the possibilities offered by changing from fully automated mechanisms to user guided algorithms. For instance, users could aid the heuristic, detecting similar pages, by choosing base policies. These policies could be used for all URIs meeting certain criteria.

Achieving a complete coverage of the proxied web application in *learning mode* is a topic, not addressed by this thesis. Again, future work might find suitable approaches for this problem. Browsers could be controlled by the proxy and crawl the web application.

Several conclusions can be drawn in terms of CSP. It can be seen as an attempt to solve some of the problems depicted by Reis et al. in 2007 [3]. While still relying on origins to define security boundaries, unwanted code can be strictly limited in capabilities by a policy. This thesis intentionally used the notion "web application", expressing that CSP shifts the focus from traditional websites to applications in the web.

In conclusion, this thesis proves that automatic policy generation certainly is possible. Automatically rewriting *eval*-like constructs, on the other hand, remains to be a hard problem. Inline code externalization, as presented in this thesis, worked in the majority of tested cases. Only dynamically changing inline code cannot be externalized. More specific solutions, tailored to a class of web applications, will most likely yield even better results. Nevertheless, if the proxy's concept is continuously extended, it has the potential to substantially promote the use of CSP in the web.

A. Appendix

A.1. Alexa Top 1000 Analysis

Alexa's top 1000 most popular websites[1] were used for a study supporting various claims of this thesis. This analysis primarily features the proportion of same-origin compared to cross-origin resources and the percentage of hosts using code patterns incompatible with CSP. The study was conducted with a recursion depth of one.

Collecting this data is a nontrivial task. Dynamic changes of the DOM have to be considered, just as resources, which occur more than once. It may also prove to be difficult to generate a comprehensive list of all code patterns which are able to embed a resource. There are a multitude of technologies implemented in modern browsers with such capabilities [26, 28].

Many of these problems are countered by a dynamic analysis. Browsers already comply with a lot of requirements needed to analyze highly dynamic websites. An instrumented web browser is able to execute all active content, such as JavaScript. This makes it possible to observe DOM changes and calls to CSP-incompatible functions, like `eval`. Detecting all embedded resources may still be a problem because of the aforementioned reasons. Since CSP violation reports resemble the browser's understanding of the website, they can be used to observe all resources of a document (cf. Section 4.3). A strict policy, such as the one presented in Section 5.2.1.2, allows for inline code detection, too. For this reason a web proxy was implemented, using the Python libmproxy library[2]. It injects a strict CSP Report-Only header into every served resource and stores the resulting violation reports for later analysis. Duplicates are prevented on a database level. A proxy can be deployed centrally, but serve many clients. Hence, multiple user agents were instrumented using Selenium WebDriver[3] to browse the websites.

Although many precautions were taken to ensure the correctness of the results, there are limitations. Dynamic analyses are not exhaustive. Instead, they can only observe one execution path of active content at a time, potentially missing others. Moreover, no user input was simulated. Thus no event handler could be triggered, which could have changed the DOM or executed CSP-incompatible functions. However, these concerns should not affect the conclusions drawn from the results because only tendencies are derived from the numbers.

As shown by Table A.1, overall 1398 distinct hosts were visited in the course of the analysis. This is a consequence of documents framing or redirecting to other hosts, which are not part of Alexa's top 1000 list. A total of 85163 unique reports could be gathered, which are comprised of 76821 external resources, 1119 connection attempts, 893 *eval*-like constructs and 4330 inline code blocks. All connection attempts were conducted using APIs like *XMLHttpRequest* or similar.

Of all observed hosts, about 96 percent used at least one CSP-incompatible code pattern. On average, 3.74 of such patterns exist per site. This result clearly shows that inline code and *eval*-like constructs are prevalent on today's websites.

Figure A.1 indicates a tendency towards cross-origin resources. Many websites use Content Delivery

[1] `http://www.alexa.com/topsites`, Oct. 2013

[2] A. Cortesi, *libmproxy: mitmproxy as a library*, `http://mitmproxy.org/doc/library.html`, Oct. 2013

[3] *Selenium WebDriver*, `http://www.seleniumhq.org/projects/webdriver/`, Oct. 2013

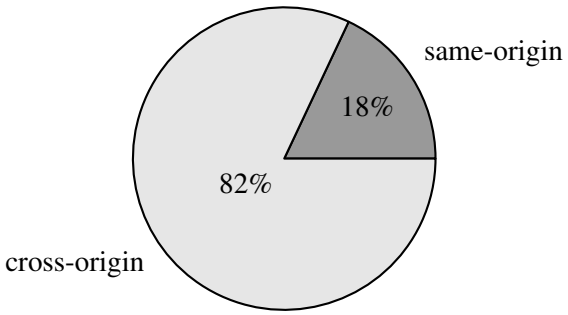

Figure A.1.: Proportions of resource origins

Name	Count
Overall distinct hosts	1398
Overall gathered reports	85163
External resources	78821
Inline scripts	2313
Inline styles	2017
Executed eval-like constructs	893
Connections	1119
Hosts incompatible with CSP	1351
Same-origin resources	13932
Cross-origin resources	64889

Table A.1.: Results of the Alexa analysis

Networks (CDNs) and separate domains to serve static resources of the web application[4]. Hence, many style sheets, scripts, images and other resources are served by cross-origin domains.

[4]BuiltWith, *Content Delivery Network Usage Statistics*, `http://trends.builtwith.com/CDN/Content-Delivery-Network`, Oct. 2013

Bibliography

[1] T. Berners-Lee and M. Fischetti, *Weaving the Web: The Original Design and Ultimate Destiny of the World Wide Web by Its Inventor*. HarperInformation, 2000.

[2] C. Marrin, "WebGL Specification - Version 1.0.2," 2013, https://www.khronos.org/registry/webgl/specs/1.0/.

[3] C. Reis, S. D. Gribble, and H. M. Levy, "Architectural Principles for Safe Web Programs," in *Sixth Workshop on Hot Topics in Networks (HotNets)*, vol. 2007, 2007.

[4] OWASP, "OWASP Top 10 - The Ten Most Critical Web Application Security Risks," 2013, http://owasptop10.googlecode.com/files/OWASP%20Top%2010%20-%202013.pdf.

[5] M. R. Stytz, "Considering Defense in Depth for Software Applications," *Security & Privacy, IEEE*, vol. 2, no. 1, pp. 72–75, 2004.

[6] S. Stamm, B. Sterne, and G. Markham, "Reining in the Web with Content Security Policy," in *Proceedings of the 19th international conference on World wide web*, ser. WWW '10. New York, NY, USA: ACM, 2010, pp. 921–930.

[7] M. Johns, B. Engelmann, and J. Posegga, "XSSDS: Server-Side Detection of Cross-Site Scripting Attacks," in *Computer Security Applications Conference, 2008. ACSAC 2008. Annual*, 2008, pp. 335–344.

[8] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic, "Noxes: A Client-Side Solution for Mitigating Cross-Site Scripting Attacks," in *Proceedings of the 2006 ACM symposium on Applied computing*. ACM, 2006, pp. 330–337.

[9] W. K. Robertson and G. Vigna, "Static Enforcement of Web Application Integrity Through Strong Typing," in *USENIX Security Symposium*, 2009, pp. 283–298.

[10] M. Ter Louw, P. Bisht, and V. Venkatakrishnan, "Analysis of Hypertext Isolation Techniques for XSS Prevention," *Web 2.0 Security and Privacy 2008*, 2008.

[11] Y. Nadji, P. Saxena, and D. Song, "Document Structure Integrity: A Robust Basis for Cross-site Scripting Defense," in *NDSS*, 2009.

[12] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay, "Safe active content in sanitized JavaScript," Technical report, Tech. Rep., Google, Inc, Tech. Rep., 2008.

[13] M. Ter Louw and V. Venkatakrishnan, "Blueprint: Robust Prevention of Cross-site Scripting Attacks for Existing Browsers," in *Security and Privacy, 2009 30th IEEE Symposium on*. IEEE, 2009, pp. 331–346.

[14] T. Jim, N. Swamy, and M. Hicks, "Defeating Script Injection Attacks with Browser-Enforced Embedded Policies," in *Proceedings of the 16th international conference on World Wide Web*. ACM, 2007, pp. 601–610.

[15] M. Heiderich, "Towards Elimination of XSS Attacks with a Trusted and Capability Controlled DOM," Ph.D. dissertation, Ruhr-University Bochum, 2012.

[16] J. Weinberger, A. Barth, and D. Song, "Towards Client-side HTML Security Policies," in *Workshop on Hot Topics on Security (HotSec)*, 2011.

[17] K. Patil, T. Vyas, F. Braun, M. Goodwin, and Z. Liang, "Poster: UserCSP-User Specified Content Security Policies," 2013.

[18] A. Javed, "CSP AiDer: An Automated Recommendation of Content Security Policy for Web Applications," 2011.

[19] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "RFC 2616: Hypertext Transfer Protocol – HTTP/1.1," RFC, 1999, http://tools.ietf.org/html/rfc2616.

[20] T. Berners-Lee, R. Fielding, and H. Frystyk, "RFC 1945: Hypertext Transfer Protocol – HTTP/1.0," RFC, 1996, http://tools.ietf.org/html/rfc1945.

[21] M. Shapiro, "Structure and Encapsulation in Distributed Systems: the Proxy Principle," in *icdcs*. Cambridge, MA, USA, États-Unis: IEEE, 1986, pp. 198–204.

[22] A. Luotonen and K. Altis, "World-Wide Web proxies," *Computer Networks and ISDN Systems* , vol. 27, no. 2, pp. 147 – 154, 1994, selected Papers of the First World-Wide Web Conference.

[23] M. Abrams, C. R. Standridge, G. Abdulla, S. Williams, and E. A. Fox, "Caching proxies: Limitations and potentials," 1995.

[24] D. Koblas and M. R. Koblas, "Socks," in *UNIX Security III Symposium (1992 USENIX Security Symposium)*, 1992, pp. 77–83.

[25] T. Berners-Lee, "HTML," 1992, http://www.w3.org/History/19921103-hypertext/hypertext/WWW/MarkUp/MarkUp.html.

[26] R. Berjon, S. Faulkner, T. Leithead, E. D. Navara, E. O'Connor, S. Pfeiffer, and I. Hickson, "HTML5," W3C Candidate Recommendation, 2013, http://www.w3.org/TR/html5/.

[27] I. Hickson, "HTML," Living Standard, http://www.whatwg.org/specs/web-apps/current-work/multipage/.

[28] B. Bos, T. Çelik, I. Hickson, and H. W. Lie, "Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification," W3C Recommendation, 2011, http://www.w3.org/TR/2011/REC-CSS2-20110607/.

[29] T. Çelik, E. J. Etemad, D. Glazman, I. Hickson, P. Linss, and J. Williams, "Selectors Level 3," W3C Recommendation, 2011, http://www.w3.org/TR/css3-selectors/.

[30] A. van Kesteren, T. Çelik, D. Glazman, and H. W. Lie, "Media Queries," W3C Recommendation, 2012, http://www.w3.org/TR/css3-mediaqueries/.

[31] P. Linss and C. Lilley, "CSS Namespaces Module," W3C Recommendation, 2011, http://www.w3.org/TR/css3-namespace/.

[32] T. Çelik, C. Lilley, L. D. Baron, S. Pemberton, and B. Pettit, "CSS Color Module Level 3," W3C Recommendation, 2011, http://www.w3.org/TR/css3-color/.

[33] "ECMAScript Language Specification - 5.1 Edition," ECMA Standard, 2011, http://www.ecma-international.org/ecma-262/5.1/.

[34] S. Tilkov and S. Vinoski, "Node. js: Using JavaScript to Build High-Performance Network Programs," *Internet Computing, IEEE*, vol. 14, no. 6, pp. 80–83, 2010.

[35] J. Aubourg, J. Song, H. R. M. Steen, and A. van Kesteren, "XMLHttpRequest," W3C Working Draft, 2012, http://www.w3.org/TR/XMLHttpRequest/.

[36] I. Hickson, "The WebSocket API," W3C Candidate Recommendation, 2012, http://www.w3.org/TR/websockets/.

[37] D. Crockford, "RFC 4627: The application/json Media Type for JavaScript Object Notation (JSON)," RFC, 2006, http://tools.ietf.org/html/rfc4627.

[38] A. L. Hors, P. L. Hégaret, L. Wood, G. Nicol, J. Robie, M. Champion, and S. Byrne, "Document Object Model (DOM) Level 3 Core Specification," W3C Recommendation, 2004, http://www.w3.org/TR/DOM-Level-3-Core/.

[39] A. van Kesteren, A. Gregor, L. Hunt, and Ms2ger, "DOM4," W3C Working Draft, 2012, http://www.w3.org/TR/domcore/.

[40] A. Barth, "RFC 6454: The Web Origin Concept," RFC, 2011, http://tools.ietf.org/html/rfc6454.

[41] F. Braun, "Origin Policy Enforcement in Modern Browsers," Diploma Thesis, 2012.

[42] P. Gordon, "Data Leakage - Threats and Mitigation," 2007.

[43] N. Barrett, "Penetration Testing and Social Engineering: Hacking the Weakest Link," *Information Security Technical Report*, vol. 8, no. 4, pp. 56–64, 2003.

[44] A. Klein, "DOM Based Cross Site Scripting or XSS of the Third Kind," 2005, http://www.webappsec.org/projects/articles/071105.html.

[45] M. Heiderich, J. Schwenk, T. Frosch, J. Magazinius, and E. Z. Yang, "mXSS Attacks: Attacking well-secured Web-Applications by using innerHTML Mutations," 2013.

[46] A. Barth and B. Sterne, "Content Security Policy 1.0," W3C Candidate Recommendation, 2012, http://www.w3.org/TR/CSP/.

[47] M. Heiderich, M. Niemietz, F. Schuster, T. Holz, and J. Schwenk, "Scriptless Attacks: Stealing the Pie Without Touching the Sill," in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 760–771.

[48] J. Daggett, "CSS Fonts Module Level 3," W3C Candidate Recommendation, 2013, http://www.w3.org/TR/2013/CR-css-fonts-3-20131003/.

[49] J. Kew, T. Leming, and E. van Blokland, "WOFF File Format 1.0," W3C Recommendation, 2012, http://www.w3.org/TR/2012/REC-WOFF-20121213/.

[50] M. Zalewski, "Postcards from the post-xss world," http://lcamtuf.coredump.cx/postxss/.

[51] D. Bates, A. Barth, and C. Jackson, "Regular expressions considered harmful in client-side XSS filters," in *Proceedings of the 19th international conference on World wide web*, ser. WWW '10. New York, NY, USA: ACM, 2010, pp. 91–100.

[52] C. Karlof, U. Shankar, J. D. Tygar, and D. Wagner, "Dynamic pharming attacks and locked same-origin policies for web browsers," in *Proceedings of the 14th ACM conference on Computer and communications security*, ser. CCS '07. New York, NY, USA: ACM, 2007, pp. 58–71.

[53] D. Atkins and R. Austein, "RFC 3833: Threat Analysis of the Domain Name System (DNS)," RFC, 2004, http://tools.ietf.org/html/rfc3833.

[54] S. Kleiman, D. Shah, and B. Smaalders, *Programming with threads*. Sun Soft Press, 1996.