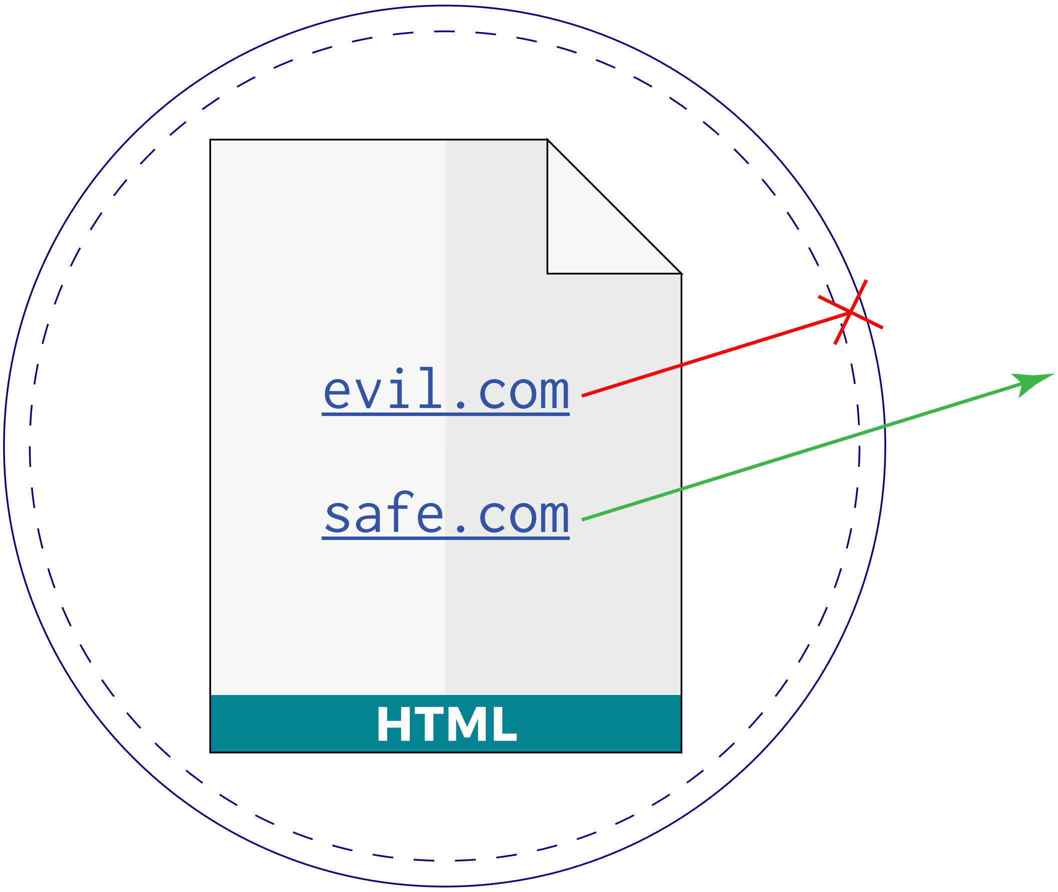
CHALMERS
UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG



These Aren't the Links You're Looking For

A Security Policy for Web Navigation

Master's thesis in Computer Science and Engineering

Adrian Bjugård
Kim Kling

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2017

Master's Thesis 2017

These Aren't the Links You're Looking For

A Security Policy for Web Navigation

Adrian Bjugård
Kim Kling

Department of Computer Science and Engineering
Division of Computer Systems and Networks
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2017

These Aren't the Links You're Looking For
A Security Policy for Web Navigation

Academic supervisors:	Alejandro Russo & Daniel Hausknecht Department of Computer Science and Engineering
Industry supervisors:	Daniel Kvist & Kasper Karlsson Omegapoint Göteborg AB
Examiner:	Andrei Sabelfeld Department of Computer Science and Engineering

Master's Thesis 2017
Department of Computer Science and Engineering
Division of Computer Systems and Networks
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover illustration: An abstract representation of a blocked attempt to navigate to `evil.com`

Typeset in LaTeX
Gothenburg, Sweden 2017

These Aren't the Links You're Looking For
A Security Policy for Web Navigation

Adrian Bjugård
Kim Kling

Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

The workings of the World Wide Web (WWW) has a history of security problems with a root cause that is hard to fix. Cross-Site Scripting (XSS) and other types of injection attacks can be mitigated to some extent, but for regular Hypertext Markup Language (HTML) and web navigation no mitigation or detection mechanism exist. In the work, attacks to introduce navigation unwanted by the web application are shown together with the current state of the art defences. In the evaluation of defences, it is found that no mechanism has the ability to defend the web application from the injected web navigation.

As a solution to this problem the Navigational Security Policy (NSP) is introduced. The NSP is an easy to understand schema enabling web developers to define the set of legal destinations originating from a web application. The NSP is shown to work as expected through the implementation, and an evaluation, of a proof of concept.

Keywords: web security, client-side security, web navigation, security policy.

Acknowledgements

We would like to thank our academic supervisor Daniel Hausknecht for his tremendous support during the writing of this thesis. Daniel, your insights into the areas of web security and academia have been invaluable resources for us. We would also like to thank Daniel Kvist and Kasper Karlsson, with their years of experience in the industry, for taking time out of their schedules to give us feedback on our work. Finally, we would like to thank Omegapoint for providing a workplace where we could perform our research, in the peace and quiet of their offices.

Adrian Bjugård & Kim Kling, Gothenburg, June 2017

Contents

1

Introduction

When the Hypertext Transfer Protocol (HTTP) emerged to become the primary standard for the web, security was not part of the specification. Multiple attempts have since been made to add security on top of the web. The use of Transport Layer Security (TLS) removes the ability of malicious entities to hijack communication channels, and Content Security Policy (CSP) restricts the locations that a web application can load content from, as well as which code is allowed to be executed.

With content inclusion from multiple sources, it is more important than ever before to ensure that web pages are rendered as intended by a web application. An instance of web navigation should never transfer users to destinations not intended by the publisher. For more complex web applications this can be a difficult task as it is hard to guarantee that both the server-side and client-side code contain no vulnerabilities and may therefore enable attacks that modify page content, such as adding web navigation.

A web application for displaying Bitcoin statistics, `blockchain.info`, had a reflected Cross-Site Scripting (XSS) vulnerability which illustrates the problem well. User input in the form of a Uniform Resource Locator (URL) parameter was used by the server and inserted into the Document Object Model (DOM) without being properly escaped. Even though blockchain.info employed CSP to deny client-side JavaScript from executing, it was still possible to use this vulnerability to inject arbitrary Hypertext Markup Language (HTML) into the resulting DOM. In turn this made it possible to inject web navigation, both in the form of links as well as redirects requiring no user interaction. This resulted in the attacker being able to take users to destinations unwanted by the web application.

Even though web applications such as `blockchain.info` can predict the set of legal destinations, the current state of the art provides no facility for enforcing restrictions to web navigation on the client side. In this thesis a schema which provides this functionality for web developers is developed.

1.1 Purpose

The purpose of this thesis is to highlight security problems related to web navigation. A lot of work has been put into securing HTTP and web communication with huge efforts put into securing the web clients. However, no protection or mechanism exists to prevent users from being taken to unwanted destinations via content injected in a web application. An investigation is conducted into the current state of the art in navigational web security, identifying potential threats, and researching ways for a web application to define navigational restrictions that can be enforced by the web client. It is investigated whether such a solution is feasible under the chosen threat model.

The thesis has two goals: (1) Identify and evaluate types of web navigation possible in a web browser, finding similarities and differences as well as possible attacks on these types of web navigation. (2) Investigate existing methods to prevent attacks on web navigation and their effectiveness as well as present a schema that enables web developers to specify a security policy regarding web navigation that the web client can enforce. A proof of concept based on the schema will be implemented and evaluated together with existing defences with respect to security offered, ease of deployment, and impact on common web applications.

1.2 Limitations

The schema presented in this thesis will not prevent the web server from providing clients with web navigation deemed unwanted, but instead identify what is unwanted navigation on the client side. If the server is fully compromised in such a way that an attacker can control all output and headers, the schema will not be able to enable safe navigation on the client side. The desires of the user will not be taken into account by the schema when blocking navigation on the client side, but instead focus on allowing only web navigation to a set of destinations sanctioned by the requested web application.

1.3 Structure

The thesis contains two major parts; defining a schema that enables security policies for web navigation, and evaluating it by performing tests using a proof of concept that implements said schema.

Chapter 1 introduces the problem and scope. Chapter 2 outlines the concepts of web navigation and details the threat model. Chapter 3 describes the attacks to defend against as well as what kind of protection the existing defences offer. Furthermore, Chapter 4 and Chapter 5 introduces the proposed schema and the proof of concept implementation respectively. The proposed schema is then evaluated together with the existing solutions in Chapter 6. At the end of the thesis, Chapter 7 includes discussion around the problem, solution and related topics. Finally, the work is concluded and recommendations for future work is presented in Chapter 8.

2

Background

In order to understand the security implications of web navigation, this section serves as a foundation and describes required knowledge within types of web navigation, the browser model, and the threat model, as they are used throughout the thesis.

2.1 Web clients

The general model of a web client contains a number of concepts important for understanding web clients, and their relevance with regards to web navigation. Those specifically relevant to the thesis topic are web contexts, web client capabilities, and the user interface (UI) of a web client.

2.1.1 Web contexts

A web context begins with the web client requesting a certain URL, the HTML response received representing the web application page requested. The web context ends when the client leaves the web application page. During the lifespan of a web context a web client may load content such as images, style sheets, fonts, and scripts, or transition the web client to a new web context.

A web application always has at least one context, the main context, but may have several sub-contexts for example in the form of one or more `iframe`-tags.

2.1.2 Web client capabilities

Generic web clients like web browsers provide the same core capabilities; user-controlled web application selection, an HTML-renderer, a JavaScript interpreter, and a facility for showing web application security status. Optionally, web browsers may provide an extension Application Programming Interface (API) for allowing users to extend the capabilities of their web browser, for example with new security protocols.

Web clients that are purpose-built for some specific web application may not have the capability of accessing other web applications. Such clients may not even provide a JavaScript interpreter, or may instead provide some application-specific language interpreter instead.

2.1.3 Web client user interfaces

All web client UIs provide at least a UI component for users to display a web context, but must not provide anything else. More generic web clients such as browsers generally provide UI components for controlling the web context display, such as an address bar where users can enter web application addresses registered in the Domain Name System (DNS). Back, forward, home, and refresh buttons are also commonly found in the UI of web browsers.

2.2 Web navigation

A web navigation is defined as the act of transitioning a web client from one web context to another. This can be triggered in multiple ways, from both the client and the web server. A non-exhaustive list of methods to navigate in a web client is shown in Table 2.1.

Table 2.1: Methods of web navigation

Type	Initiator
Changing the URL in URL bar	User
Follow a link	User or client-side script
Meta refresh	Server or client-side script
History	User or client-side script
Changing `window.location`	Client-side script
Changing `window.location.hash`	Client-side script
Form submit	User or client-side script
HTTP status code	Server

The first method of web navigation is directly changing the URL in the web browsers address bar to another location. This method is initiated by the user.

All `<a>` HTML elements provide the JavaScript function `click()` which enables web navigation to the destination of the `href` attribute. The task of following a link can be initiated either by the user clicking on the link, or programmatically by a script running in the web browser.

The history of a web browser provides means of going back and forth in the navigational history for a browser context. The UI provides buttons to go back, forth, and also several pages back and forth in one jump. Visual feedback for showing which page will be navigated to is available in web browsers that implement history buttons. The World Wide Web Consortium (W3C) introduced the same concept to client-side code via the HTML5 standard [1]. Web browsers implementing the functionality provide the JavaScript engine with the ability to trigger web navigation just like the UI does, within the navigational history with the exception of not having the ability to read from the navigation history. Client-side code issues commands via a API called the History API. Besides navigating to pages in the history, client-side code also has the ability to insert states in the history that uses the same origin as from where it is executed.

By changing the JavaScript variable `window.location`, a client-side script can initiate web navigation to any destination. Changing the `window.location.hash` is another way to trigger web navigation. Unlike changing `window.location`, it does not end script execution. This does not force a load of a new page, but instead allows the current script to decide what actions to perform. For example, the script could load different resources and display different information based on the value in `window.location.hash`. This method of web navigation is used by single-page web applications, such as those based on AngularJS, or to navigate to a specific element within the page, such as a heading.

When a form is submitted, the web client navigates to the destination specified in the `action` attribute. This can be exploited to navigate to any location by changing the `action` attribute on an existing form, or by creating a new form element. When a malicious destination exists in a form, the attacker can either wait for the form to be submitted or submit the form from JavaScript. The method `submit()` on a form object initiates web navigation to the form location specified in the `action` attribute.

A common method for servers to trigger a web navigation is to set the HTTP status code to one between 300 and 308, accompanied with setting the HTTP header `Location` to the desired destination this triggers an instantaneous web navigation by the client, without the need to further parse the page content.

2.3 Threat model

Akhawe et al. describes a threat model suitable for a thesis on navigational security for web applications [2]. The *gadget attacker*, as described in the work by Akhawe et al., takes the role of the attacker in this thesis. Other actors taking part in the threat model are the web application itself, and the users.

2.3.1 The attacker

An attacker injecting web navigation may do so for many reasons. Such attacks may be used by an attacker to, amongst other things, gather personal details from users, generate bad publicity for the targeted web application, deny access to web resources, gain control of a users client, or disseminate misinformation via otherwise reputable sources.

The gadget attacker described by Akhawe et al. has the ability to include content into benign websites, via for example a user input that may or may not be properly escaped, or via an advertisement. The gadget attacker also controls a web server with a DNS name which may be used to include content from, or redirect users to.

It is assumed that the injected content is being treated as code, enabling an HTML injection or XSS attack. The capabilities of XSS attacks are defined by the vulnerability being exploited to get control of the web application. They may be limited to content inclusion without code execution privileges while still allowing addition of web navigation, such as the non-scripted attacks described by Zalewski [3].

The methods of web navigation defined in Section 2.2 can be utilised by the gadget attacker to create web navigation that is unwanted by the web application.

2.3.2 The web application

It is assumed that web applications have the need to protect their web navigation, and require a method to do so. The web application must have a vulnerability allowing content inclusion in order for unwanted web navigation to be considered an issue.

For the purpose of this thesis it is assumed that a web application only wants to provide some predetermined functionality to its users. In this functionality the destinations to which users are sent by the web application is included.

2.3.3 The users

The International Telecommunications Union reports that over three billion people have access to the internet as of 2016 [4]. All of these people are potential users of any internet-facing application.

Given the great amount of potential users of any web application, it can be safely assumed that every web navigation available from a web application will be followed by some user.

For the intents and purposes of this research it is also assumed that users trust the web application they are using, meaning that they believe that any navigation they make is intended by the web application.

2.4 Destinations

Sets of destinations, as they are used in this thesis, are defined using a subset of components based on the generic Uniform Resource Identifier (URI) as defined in RFC 2396 and RFC 3986; `[scheme:][[//]host[:port]][/path]` [5, 6]. This pattern is used for URL-type destinations. If a destination is not a URL-type URI it does not fall into this pattern, and instead uses the more generic pattern defined in RFC2718; `[scheme:][scheme-specific-component]` [7].

The *scheme* represents the application that should handle the request, such as http, ftp, file, mailto, amongst others. The *host* and *port* together identifies a *service*. If the port is omitted, it is assumed that the protocol standard port applies. The *path* is used to identify the base of specific locations to which rules should apply. If any component; scheme, service, or path, were to be omitted, it is assumed that the rule applies without restriction regarding the missing components. Listing 2.1 shows an example destination with the scheme component `http:`, the service component `//example.org`, and the path component `/test`.

Listing 2.1: A destination with all components populated

```
http://example.org/test
```

2.5 Existing defences

There exists strategies for mitigating, or avoiding, attacks on web navigation. These are grouped into solutions using client side blacklisting or application-defined policies.

The effectiveness of all defences with respect to web navigation controls and attacks is evaluated in Section 3.2.

2.5.1 Client side blacklisting

Users have access to several defences on the client side to prevent them from visiting unwanted locations. The defences detailed here have in common the usage of a blacklist that blocks access to these locations.

In the web browsers there is usually a built-in defence that blocks content that is known to be malicious. The information can come from Google through their API called Safe Browsing, that keeps track of known to be malicious locations and content. Criteria to be malicious is that the web site is a phishing site, serves malware and/or unwanted software. Google Chrome, Apple Safari, and Mozilla Firefox are some of the web browsers that use the Google Safe Browsing API. The Google Safe Browsing API is used by an estimated one billion users [8].

If the web browser does not have support for Google Safe Browsing or the user wants to block web sites other than what is provided by Google Safe Browsing, there are several content blockers to help out. Content blockers can be installed as extension for the web browser, or as software that hooks into network communication. Applications aiding to block access on the network level are software such as anti-virus and firewalls.

2.5.2 Application-defined policies

CSP is a schema allowing a web application to provide a policy containing rules on what content a web client should be allowed to include. Scripts, styling, images, fonts, and media are some of the content types that can be blocked using CSP.

A policy contains directives that for each content type whitelists locations from where content can be loaded. It enables the policy to allow sources based on the application protocol, service and path. It also includes directives to apply a sandbox to included content, state whether it should be allowed in iframes and to where violation reports should be posted. The most current version of CSP is CSP Level 2 [9].

3

Protection against attacks on web navigation

In order to improve the current state of the art, it is necessary to define what a defence against attacks on web navigation must provide. This chapter contains attacks and code used when evaluating the solutions later in the chapter.

3.1 Performing attacks on web navigation

As per the assumptions made in Section 2.3, the attacker has an incentive to take users away from the current web application, and cause them to navigate to the attackers or some other unintended web application. Possible incentives are financial gain through, for example, phishing, Denial of Service (DoS), or disseminating misinformation via otherwise reputable sources.

Based on the methods of web navigation outlined in Section 2.2, in the event of a successful code injection there are several ways for an attacker to cause a web client to navigate. This section details both HTML-injection and JavaScript based methods, describing the attacks and providing code used to test protective measures.

3.1.1 HTML-injection

HTML provides different methods of initiating a web navigation. While normally used by web applications to provide intended navigation, such elements can also be created with malicious intent, and injected by an attacker. Link navigation, the `meta`-tag `http-equiv="refresh"` functionality, and the `form`-tag action attribute are three such methods.

The threat model makes the assumption that all web navigation included in the HTML DOM will eventually be followed by a user. As a result, all an attacker needs to do is include web navigation in the form of a link and some users will end up clicking on it, taking them to the destination chosen by the attacker.

Listing 3.1 shows the content of an HTML injection that initiates navigation to `https://example.org` when clicked on by a user.

Listing 3.1: Inserting a link clickable by users

```
<a href="https://example.org">Click me!</a>
```

The meta-tag `http-equiv="refresh"` functionality can also be used by an attacker to force a web client to perform an automatic web navigation to a specified destination with a given delay. This delay may be zero seconds, implying an immediate redirect. Listing 3.2 shows the code used to cause instant navigation to `https://example.org` without any user involvement. The tag can be inserted into the page either by the server, or by a client script. Both methods trigger navigation as soon as the meta-tag has been parsed by the client.

Listing 3.2: Using a meta-tag to cause navigation

```
<meta http-equiv="refresh" content="0;URL='https://example.org'" />
```

A navigation can also be triggered when a web client submits a form, by setting the `action` attribute of a `form` tag. Listing 3.3 shows an example piece of code which, if added to the DOM of a web application, will cause the web client to introduce navigation to `https://example.org` into the page. The navigation will occur when the form is submitted.

Listing 3.3: Introduce navigation by inserting a form

```
<form action="https://example.org"><input type="submit"></form>
```

3.1.2 JavaScript-based

In addition to HTML injection, further attacks on web navigation are possible using JavaScript. Changing the `window.location` variable, calling history API functions, as well as automating the attacks described in Section 3.1.1, are a few examples of navigation enabled by JavaScript.

By changing the JavaScript variable `window.location`, a client-side script can initiate navigation to any destination chosen by the attacker. This method of navigation can be triggered in multiple ways by the attacker, the most basic example of which can be seen in Listing 3.4, where the malicious script simply sets the location as soon as it is parsed by the JavaScript-engine in the client, causing the navigation to take place. Another approach for the attacker is to place the code inside a function call, and register that to an eventlistener, which has the benefit of being less noticeable than the first example.

Listing 3.4: Using window.location to navigate

```
<script>window.location = "https://example.org"</script>
```

The History API of a web browser provides a means of going back and forth in the navigational history for the current browsing context. Listing 3.5 shows some code which when run causes the web client to navigate to the previous page in history without user interaction. Attacks using the History API does not allow the attacker to set an arbitary destination. It only allows the attacker to jump a certain steps back or forth in history and can thus only use this attack as a denial of service rather than redirecting the users to the attackers web application.

Listing 3.5: Navigating back one step in History using JavaScript

```
<script>window.history.go(-1)</script>
```

Additionally, the attacks previously described in Section 3.1.1 can be automated using JavaScript.

Attacks based on link navigation can be automated using JavaScript on the client side, by calling the `click()` function on the included link. Listing 3.6 shows the JavaScript-enhanced version of the attack from Listing 3.1. This attack initiates an automatic navigation to `https://example.org` as soon as the code is executed, without any user interaction at all.

Listing 3.6: Insert a link and follow from JavaScript

```
<a id="link" href="https://example.org">Click me</a>
<script>document.getElementById('link').click();</script>
```

JavaScript also enables automatic submission of forms, which triggers a navigation in the web client. Listing 3.7 demonstrates a JavaScript-enhanced version of the `form`-based navigation shown in Listing 3.3. When the code is run by the web client, this attack initiates an automatic navigation to `https://example.org`, without user interaction required.

Listing 3.7: Cause navigation by submitting a form

```
<form id="form" action="https://example.org"></form>
<script>document.getElementById('form').submit()</script>
```

3.2 Evaluation of existing defences

This section looks at the existing defences identified in Section 2.5, and evaluates them in order to establish their usefulness in solving the problem at hand.

3.2.1 Client side blacklisting

Blacklisting, in general, blocks known malicious locations. Examples of client side blacklisting are Google Safe Browsing, content blockers, anti-virus and firewalls.

Google Safe Browsing is based on a centrally managed blacklist provided by Google. It restricts on locations that are harmful to the device, while content blockers in general provides blocking for different types of content such as tracking, ads and malware. Users can also add their own content to general content blockers. Anti-virus and firewalls work in a similar manner, blocking requests to Internet-based destinations that are known to serve malicious content.

These types of protections do not allow the developer to specify the set of legal destinations, and instead uses predefined lists, or rules added by the user. From the attackers point of view, none of the navigational attacks from Section 3.1 are prevented if the destination is not considered to be harmful to the device, or if it has not been added to the blacklist by the user.

If the attacker is targeting a certain web application and notices they are being blocked, it is trivial to choose a different DNS name or Internet Protocol (IP) address in the attack, rendering a blacklist useless in this scenario. Other ways to fingerprint malicious content such as computing hash values and compare can also be circumvented by repackaging the malicious content.

From this it is clear that blacklisting content or navigation is not the best way to secure neither the web client nor the web application. The absence of functionality for a web application to provide a policy to the web client is also a big drawback since the goal is for the web application to restrict any navigation that it does not provide.

3.2.2 Content Security Policy

Since CSP is a schema designed to restrict content, it does not aim to prevent navigation that is unwanted by the web application. It does, however, affect navigation through its ability to prevent JavaScript from being loaded. Besides restricting JavaScript, CSP Level 2 includes the directive `form-action`. This directive limits the destinations to which a web client can submit its form data. As a result, CSP prevents some types of navigation as a side effect of restricting content to be loaded.

The following paragraphs enumerates the attacks from Section 3.1 and describe their effectiveness against a web application using a strict policy containing only `script-src 'none'; form-action 'none'`.

From the two groups of attacks, HTML-injection and JavaScript-based, it can be seen that CSP prevents some types of navigation, while not preventing other types. Form submissions to destinations unwanted by the web application are blocked by the use of the `form-action` directive. Links and `meta`-tags are still being allowed to be inserted and followed since they does not fall under any of the directive CSP Level 2 offers. The web navigation will be followed as per the assumptions.

All of the JavaScript-based attacks and improvements through scripted navigation is being prevented from happening due to the directive `script-src` with the `'none'` value.

The above scenario assumes a best case scenario for CSP. This is rarely the case in web applications as JavaScript provides interactivity and functionality that enriches web applications. CSP relies on preventing untrusted sources from being included, and assuming that the web application does allow some sources of scripts, the attacker can attempt to get their scripts to be delivered from those trusted sources. If an attacker manages to deliver the attack payload from a trusted source, CSP would allow the navigation. In practice there are only a few web applications that have policies as strict as needed to prevent all untrusted JavaScript to run. During research on the subject Weichselbaum et al. found that “94.68% of policies that attempt to limit script execution are ineffective, and that 99.34% of hosts with CSP use policies that offer no benefit against XSS” [10].

3.3 Requirements for a solution

In order to successfully block navigation that is unwanted by the web application, a few requirements must be met. Such a schema must be able to block web navigation that is not intended by the web application at the current location, not affect navigation or requests that are intended, require no actions by the user, and have a simple way of defining rules by the web developer. On top of these general requirements, the following features are also required.

Allowing destinations

There must be a method for the web application to define legal destinations. A whitelist should be used to define destinations to where navigation should be allowed. Navigational requests to destinations not present in the set of legal destinations must not be permitted to proceed. It is necessary to define a language for the whitelist which provides high flexibility and enables destinations based on protocol, host, port, and path. The language for legal destinations should enable sufficient granularity.

It is also required that there be a simple way to allow navigation to the current protocol and host, regardless of if the site is accessible from multiple domains.

Intermediary navigation

Besides having the binary choice of either allowing or denying a web navigation, a developer must have the option of displaying an intermediary page before navigation. This is intended to be used for destinations that are not trusted, but may be a part of content that adds value. An example is user content on forums or in comments. The intermediary page may be either a web page controlled by the web developer, or a page provided by the web client. This enables the ability of defining an action to be performed for destinations that are not whitelisted.

Policy lifespan

The lifespan of a navigational policy must last for the entirety of a web applications main context lifespan. When the main context changes, the navigational policy must be dropped. This criteria is important because a navigational policy implementation that does not fully follow this requirement will cause issues with the legal navigation of other web applications.

Increased protection

Implementing a navigational policy may not decrease the level of protection achieved by a web application, it must result in increased security for web clients accessing the web application.

Compatibility

Implementing a navigational policy enforcer in a web client should have no impact on web applications that do not provide a navigational policy. Similarly, web applications that target web browsers as clients should be able to serve content to web clients that do not implement a navigational policy enforcer.

Reporting

Developers may find it useful to collect information about violations of the navigational policy, particularly in the process of adopting a navigational policy. There must be a way for developers to specify the destination of such reports. The reports should contain information of the current location, the blocked destination, the policy in use and the violated rule.

Ease of deployment

It is of high relevance that a web developer or an administrator can grasp the idea and workings of a solution, and that policies for the solution are easy to create. The policy language should be developer friendly, be flexible enough to create complex policies if needed, and portable enough to be used regardless of underlying technologies.

4

Design of the Navigational Security Policy

As a means to improve the state of the art, and remove the underlying problem identified in the purpose of this thesis, this chapter introduces the Navigational Security Policy. Building upon the requirements identified in Section 3.3, it is a proposal for a schema to stop client navigation that is not intended by a web application.

4.1 Overview

The Navigational Security Policy (NSP) is a proposal for a schema solving the problems presented in this thesis. The design is broken down into how policies are constructed, how policy delivery is handled, and how client security is ensured.

When a web application implementing NSP receives a request, it must construct a policy before issuing a response to the client. This could for example mean reading a static policy, or generating one based on the context of the requested page. The resulting policy must only include navigation that the web application intends as legal destinations for web clients to navigate to.

The task of delivering a policy to the client can be accomplished in multiple different ways, each offering different levels of flexibility and security. A non-exhaustive list of possible policy delivery mechanisms is presented here.

- Bundling a static policy with a client application.
- Including a policy with content sent when the client requests data.
- Including a link to a policy which is fetched independently of the content sent to a the client.

However the policy is delivered it is important that the policy is accessible on the client prior to any navigation taking place.

Once a policy has been received by an NSP-aware web client, it can be applied to client-side web navigation. When a web navigation is initiated, the NSP-aware web client checks if the target destination is allowed by the policy and intervenes if it is not. An overview of the client-side NSP application flow is depicted in Figure 4.1.

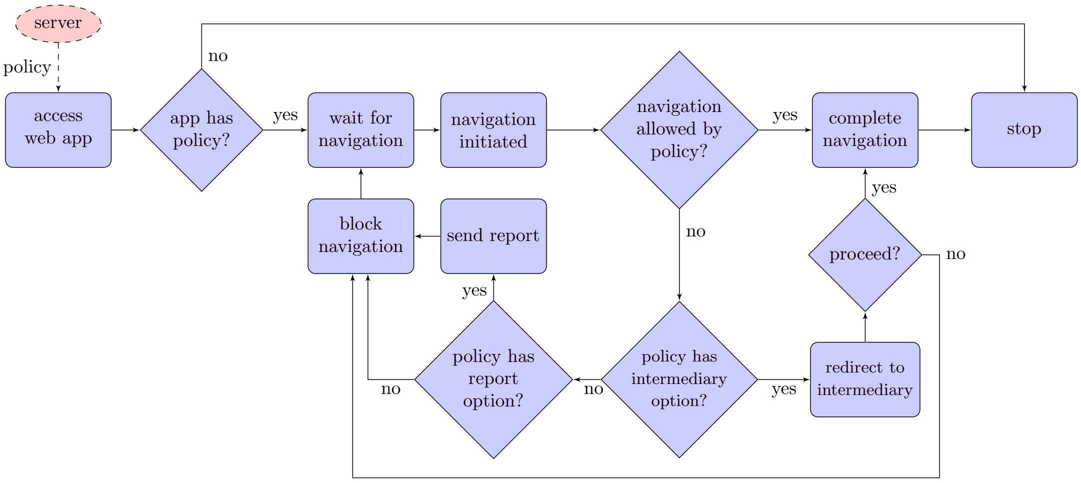


Figure 4.1: NSP-aware client application flow

4.2 Policy definition

In order to keep the policy language easy to read, JavaScript Object Notation (JSON) was chosen as it is a well known format that is commonly used in web applications already. JSON provides a good representation that is small, easy to read for humans, easy to process by the web browser, and uses a key-value structure which enables nested objects, allowing high flexibility for developers [11].

4.2.1 Syntax

The syntax for destinations in the policy follow the basic style set out in Section 2.4, with some extensions.

To allow more flexibility, the definition of destinations allows the wildcard character "*" to represent one or more legal characters. Listing 4.1 shows an example of how the wildcard character can be utilised. It shows the set of destinations that uses the protocol `https` and the hosts with all subdomains for `example.org`, excluding the naked `example.org` domain, without subdomain.

Listing 4.1: A destination using the wildcard character

```
https://*.example.org
```

DNS-based service names can be specified using the wildcard character, but only at the start of the service component, never in the middle or end. The wildcard character may not be used in the scheme, or path components of a destination. A specified path component denotes the root of a legal set of destinations, everything under the specified path is implied to be part of the set. Listing 4.2 shows examples of illegal wildcard placements.

Listing 4.2: Illegal uses of the wildcard character in a destination

```
www.example.*
www.*.com
*://example.org
example.org/app/*
```

The self-referencing string `'origin'` is used to refer to the current scheme and service. A destination to `'origin'` does not restrict the path. Examples for the use of `'origin'` is shown in Listing A. Line 1 targets destinations matching the current location's protocol and service. Line 2 targets the destination with path `/test` on the current protocol and service.

Listing 4.3: Usage for `'origin'`

```
1 'origin'
2 'origin'/test
```

Listing 4.4 shows how the previous destinations are translated if the current service is `example.org`.

Listing 4.4: Translation for `'origin'` on example.org

```
example.org
example.org/test
```

4.2.2 Policy structure

In order to define a policy, a set of *directives* are used. These directives combined form the policy and allows, or denies navigation to destinations. A policy can also include the option to send a report to the web application every time the client side code tries to break a rule, or display an intermediary page before a navigation is performed. The directives used in a policy is shown in the following sections.

The `allow` directive

Specifying legal destinations for navigation in the web application is done using the `allow` directive. Only the set of destinations formed by the values in the corresponding list is allowed as destinations for navigation. Listing 4.5 shows a simple policy containing only an `allow` directive with a corresponding list of values. In this case it only contains the set of destinations that matches `example.org`, disallowing anything else.

Listing 4.5: A policy allowing navigation only to `example.org`

```
{
    "allow": [
        "example.org"
    ]
}
```

Listing 4.6 provides an example with several values in the list corresponding to the `allow` directive. The set of allowed destinations is formed by taking the union of the individual sets formed by each value. In the example, the union; $\{$`'origin'`$\} \cup \{$`https://example.org`$\} \cup \{$`google.com/example`$\}$ forms the set of allowed destinations.

Listing 4.6: A policy allowing navigation to the current location, `example.org`, and `google.com`

```
{
    "allow": [
        "'origin'",
        "https://example.org",
        "google.com/example"
    ]
}
```

The `except` directive

To enable policies to remove destinations from the set of allowed destinations, the `except` directive is introduced. In cases where there is a need to use wildcards in the `allow` list, `except` can be used to remove some of the destinations that match the wildard. Similar to the `allow` directive, the `except` directive has a list of values that each represent a set of destinations. Together they form the set to be excepted from the allowed destinations. The legal set of destinations is retrieved by taking the difference of the allowed set and the excepted set: $\{allowed\} - \{excepted\} = \{legal\}$.

An example of the `except` directive is shown in Listing 4.7. In this case, the set of destinations formed by the host `evil.example.org` is removed from the set of destinations formed by all subdomains of `example.org` and the protocol `https`. Resulting in the legal set of destinations $\{$`https://*.example.org`$\} - \{$`evil.example.org`$\}$.

Listing 4.7: Policy that excepts `evil.example.org` from allowed destinations

```
{
    "allow": [
        "https://*.example.org"
    ],
    "except": [
        "evil.example.org"
    ]
}
```

The report-uri directive

While deploying a policy it may at an early stage be useful to receive information about policy violations from clients. The `report-uri` directive allows a developer to specify where such reports should be sent. Navigational requests with a legal destination do not trigger reports, only attempting to navigate to illegal destinations will cause a report to be sent to the defined `report-uri`. The value of the `report-uri` directive follows the style defined in Section 2.4. A policy using the `report-uri` directive is shown in Listing 4.8, defining "/nsp-reports" as the destination where reports are to be sent.

Listing 4.8: A policy containing the `report-uri` directive

```
{
    "allow": [
        "'origin'"
    ],
    "report-uri": "/nsp-reports"
}
```

The intermediary-uri directive

The NSP enables web applications to dictate navigation via an intermediary page, enforced by the web client, via the `intermediary-uri` directive. Target destinations which are not in the set of legal destinations are redirected to this URI by the web client, prior to completing the navigation. The URI specified by the `intermediary-uri` directive is provided the original navigation target by the web client, and can perform some task prior to deciding how to proceed.

The `intermediary-uri` directive is expected to contain a destination as defined in Section 2.4, extended with the special destination string `'client'`. The special destination string `'client'` is used to refer to a generic intermediary page, provided by the NSP-aware client itself. An example policy containing the `intermediary-uri` directive is shown in Listing 4.9.

Listing 4.9: A policy containing the `intermediary-uri` directive

```
{
    "allow": [
        "'origin'"
    ],
    "intermediary-uri": "'client'"
}
```

4.3 Special considerations

This section details aspects of the schema that require special considerations for an implementation.

4.3.1 Multiple occurrences

The first legal occurrence of a policy is used to define the policy. Any consecutive occurrences of policies, when one has already been defined by a header, can only restrict the policy. This means that you can except more destinations, but never allow more destinations, in consecutive policies.

4.3.2 Partial support

Any implementation of NSP, which follows the schema defined in this thesis, can only provide client security if both the server and client have implemented support for it. This is important to enable smooth rollout of NSP.

If only the server is NSP-aware, the policy is inserted and the client ignores it. In the case where the web client is NSP-aware, but the web server is not, no policy will be provided by the web server. The web client can in this case not enforce anything and does not restrict navigation.

These properties allow for fast adoption by both web application servers and web clients, as there is no reason introducing support on either side would impact the other side. If such problems existed, a more coordinated effort would be necessary, and implementations for more general web clients, such as browsers, would be difficult to deploy.

4.3.3 Redirects

Navigation beyond the first navigation is not considered by NSP. This means that navigation that causes redirects or instant navigation is not prevented by NSP. The reason for this is that each each redirect creates a new context. This causes the previous policy to be dropped and, if applicable, a new policy to be loaded.

5

Proof of concept implementation

In order to demonstrate that the ideas presented in this thesis are feasible, a proof of concept was implemented. The proof of concept uses a Chromium browser extension on the client side to enforce navigation based on a policy received from the server as part of an HTTP header. The Chromium browser was chosen as a target for implementation because it is open source and has a well-documented extension API.

The implementation relies on a server-side component providing a policy that a client-side component can enforce. The client side is more complex than the server side, implementing a parser component for interpreting the received policy, and an enforcer component to govern client navigation based on the received policy.

The code for the proof of concept implementation is available upon request.

5.1 Server-side architecture

The server-side component required for compatibility with the proof of concept client must supply policies in a HTTP header named `X-Navigational-Security-Policy`. It must supply the policy using valid JSON, containing the `allow` directive, and optionally also the `intermediary-uri` directive. The language of the directives follows the format defined in Section 4.2.

For the proof of concept implementation, such a component was integrated into a `python-flask` application, as well as in the client-side server-component emulator NSP Injector.

The NSP Injector is designed to facilitate the process of implementing policies for real-world applications. To do so it emulates the server-side component on the client-side, on top of existing web applications. Web clients with the NSP Injector installed will add locally stored NSP policies to responses received by the web browser, before the NSP Enforcer parses a received policy.

5.2 Client-side architecture

In order to apply server-generated navigational policies on the client, it is necessary to have some control over the client. A browser extension was chosen for this task as it is generic by nature, allowing NSP deployment on a broad variety of web applications.

NSP Enforcer is the name of the client-side implementation. It is a browser extension built around the Chromium extension API, utilising the `webRequest`, `omnibox`, and `tabs` APIs.

When navigating to a web application that provides a policy, NSP Enforcer parses the received policy as defined in Section 4.2. The NSP Enforcer implementation supports the directives `allow` and `intermediary-uri`, according to their definitions in Section 4.2.

The `intermediary-uri` implementation in NSP Enforcer provides the intermediary page with the original target destination via the URL parameter `b64url`. The original target destination is base-64 encoded in order to be able to supply any valid URI to the intermediary page.

Any attempts to trigger a navigation will cause NSP Enforcer to check the target of the navigation against the list provided in the `allow` section of the provided policy JSON-object. If the provided policy does not list the target in the `allow` section, and does not provide the `intermediary-uri` directive, the navigation is blocked from continuing. If the `intermediary-uri` directive is provided, the navigation is redirected to the specified intermediary page. A detailed flowchart of the NSP Enforcer implementation is depicted in Figure 5.1. The difference from the flowchart in Figure 4.1 comes from the fact that the NSP Enforcer does not implement every feature outlined in Chapter 4.

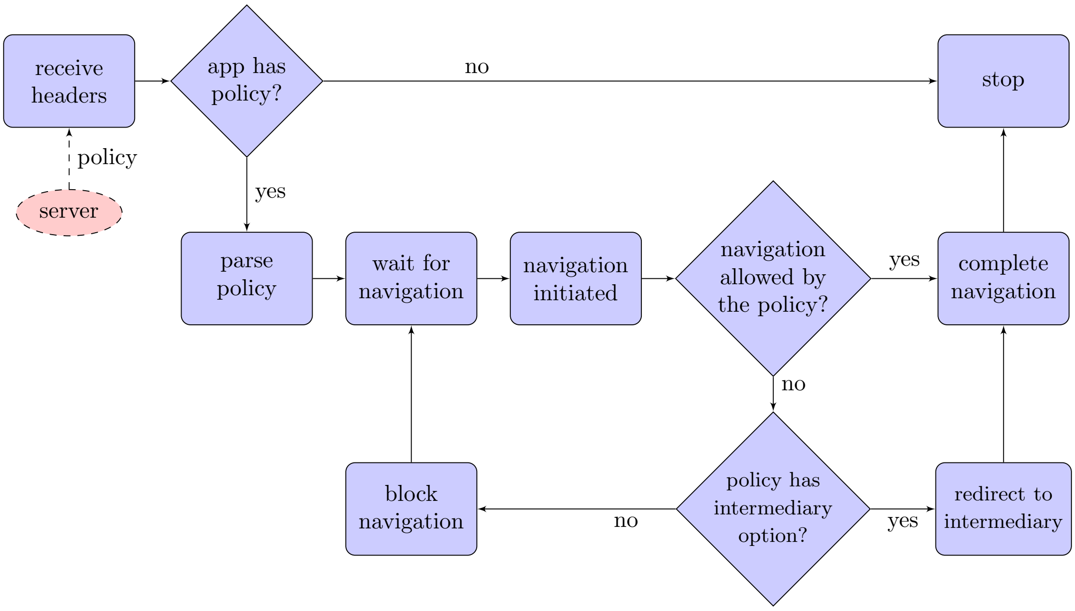


Figure 5.1: NSP Enforcer application flow

5.3 Limitations

While useful for flexibility and ease of distribution, implementing the proof of concept as a browser extension introduces some limitations due to shortcomings in the extension API of Chromium, the chosen target browser.

In order to fulfil the policy lifespan requirement specified in Section 3.3, it was necessary to trade away the ability to govern navigational targets opened in a new window. This is due to the inability, in the Chromium extension API, to directly relate a newly opened window to its parent browsing context. This limitation makes it impossible to apply a policy from the parent browsing context to any requests made in a new window, without also capturing new windows created in the traditional way.

This could be solved with a relation between the `webNavigation` and `webRequest` APIs, as the `webNavigation` API contains the information required to distinguish whether a navigation originated in another tab. The `webNavigation` API can not stop a navigation from taking place, forcing us to use the `webRequest` API instead, in blocking mode.

This same shortcoming in the Chromium extension API is also why initiating a new navigation via the address bar is unintentionally captured by NSP Enforcer, as the `webRequest` API can not tell the difference between a navigation initiated from the address bar or from within the current browsing context. The `webNavigation` API provides this functionality, however without the ability to stop a navigation.

A workaround to this shortcoming is provided via the `omnibox` API. By prefixing an address bar navigation with “nsp”, NSP Enforcer discards the current NSP context prior to initiating navigation.

Only a subset of the design proposed in Chapter 4, specifically the directives `allow` and `intermediary-uri`, were implemented in the proof of concept, as these were deemed enough to demonstrate the usefulness of the NSP.

6

Evaluation

In this chapter the NSP schema and the proof of concept implementation is evaluated. The evaluation focuses on identifying whether the schema proposed in Chapter 4 fulfils the requirements set out in Section 3.3. Additionally the potential impact of using the NSP on `blockchain.info`, and Twitter is examined. The chapter concludes with an empirical study on two real world applications, using the proof of concept implementation to simulate an NSP deployment.

6.1 Fulfilment of requirements

NSP does not affect legal navigation as it acts strictly on a policy that is defined by an authoritative entity, while still blocking undesired navigation. Since the directives use sets of destinations, it allows flexibility in order to not block destinations that should be considered legal. The proof of concept successfully enforces this behaviour and implements the destination concept. While the proof of concept does not implement any way to except destinations from a set of destinations, it still shows the most important parts of defining a policy.

The requirement that no action be required by the user is fulfilled, as the browser enforces a policy regardless of what the user thinks are desired destinations. NSP however allows policies that enable users to decide whether they want to proceed with a navigation or not.

In order to allow defining policies that are easily understood at a glance, NSP uses the commonly used object notation language JSON. Working with the proof of concept shows that this is the case.

The proof of concept implements the requirement for allowing an intermediary page when navigating to untrusted destinations. It successfully intercepts illegal navigation and notifies the user. There is support for using an arbitrary page as the intermediary page, allowing the possibility of building advanced server-side logic besides just prompting the user to either accept or decline to proceed with the navigation.

The proof of concept implementation does not include support for the reporting directive from the schema definition. This was not required in order to test the benefits, but is important for adoption.

Policy lifespan in the proof of concept matches that described in the requirements. It acts on a per context basis and enforces web navigation restrictions until an allowed web navigation takes place. That could in turn load a new policy, always keeping the user secured with a policy.

It is required that a security mechanism for web navigation does not decrease the level of protection offered for a user. Such a mechanism must either keep the same level of protection as without the mechanism, or increase the level of protection for the user. This requirement is fulfilled by the proof of concept implementation. Using a policy that does not perform any restrictions provides the same level of security as not providing a policy at all.

6.2 Mitigating attacks

An evaluation of the NSP schema, and proof of concept, requires that the effect of the attacks mentioned in Section 3.1 be considered with NSP in use.

All of the mentioned attacks eventually cause a request for a new context, regardless of the method. This fact is what makes NSP effective in intercepting and examining all requests. It is also an indication that the NSP has a high chance of catching some type of web navigation that is not considered in this thesis. As long as a navigation causes a context load, NSP will be able to intercept it.

6.3 Deploying Navigational Security Policy

This section investigates what effect an NSP deployment would have on two real world web applications; `blockchain.info` from the perspective described in Chapter 1, and Twitter, evaluating whether Twitters own intermediary page solution could be improved using NSP.

6.3.1 Revisiting the `blockchain.info` example

By not properly escaping the user input, the `blockchain.info` web application unintentionally enables arbitrary content to be included in the DOM returned to users of the application.

The web application `blockchain.info` already makes use of a strict CSP policy which stops injected JavaScript from executing and taking over the client-side application. The policy in question is included in Appendix A. The only avenue available, for an attacker wishing to perform a phishing attack, is to attempt to cause a web navigation via HTML.

By utilising NSP, with a sufficiently strict policy, the attacker is blocked from causing unwanted web navigation via both HTML and JavaScript. This successfully stops the phishing attack from having any real effect on the `blockchain.info` web application.

6.3.2 Twitter

The Twitter web application uses a bouncing page for all web navigation available in the DOM sent to web clients. This functionality makes it possible for the web server to perform validation, or record statistics, on web navigation that passes through the bouncing page. While the intended functionality of this bouncing page can be used to improve web navigation security, if it is not enforced client-side it is possible for attackers to circumvent it.

A policy with the `intermediary-uri` directive can be used to enforce navigation via the bouncing page on the client side. Using NSP, any web navigation introduced by an attacker is also navigated to via the bouncing page. This makes it possible for server-side security functions to validate or reject the injected web navigation.

6.4 Policy generation for real world web applications

In order to verify that the proof of concept is effective, and by extension also the NSP schema, an empirical study on designing safe policies for existing web applications is conducted. This study is performed in order to see whether the policies work as intended, and to verify that it does not break functionality in the web application.

6.4.1 Study methodology

To facilitate evaluating how an existing web application behaves with a policy, the NSP Injector extension was implemented as described in Section 5.1. NSP Injector hooks in before the NSP Enforcer described in Section 5.2, so that the header is accessible when required. In order to have as minimal effect on the original transmission as possible, the extension only injects an NSP header and leaves the rest of the response as it was received. An illustration of how the request passes between the extensions is shown in Figure 6.1.

To manually build policies all navigation available in a web application is enumerated using a script. For the purpose of this study only links are considered. Each destination is evaluated based on the scheme, service, and path components, as well as where in the web application it occurs. It is then decided whether it is a trusted destination.

A policy is built using an iterative approach, with rule additions based on the occurrence of destinations. Each destination can, besides be individually evaluated, be is tested against the constructed policy to see whether it is already an allowed destination or not. Based on the result, rules can then be added or omitted in order to improve the policy. When developing a policy the amount of user content the web application should contain, and how strict a policy would need to be, must be considered.

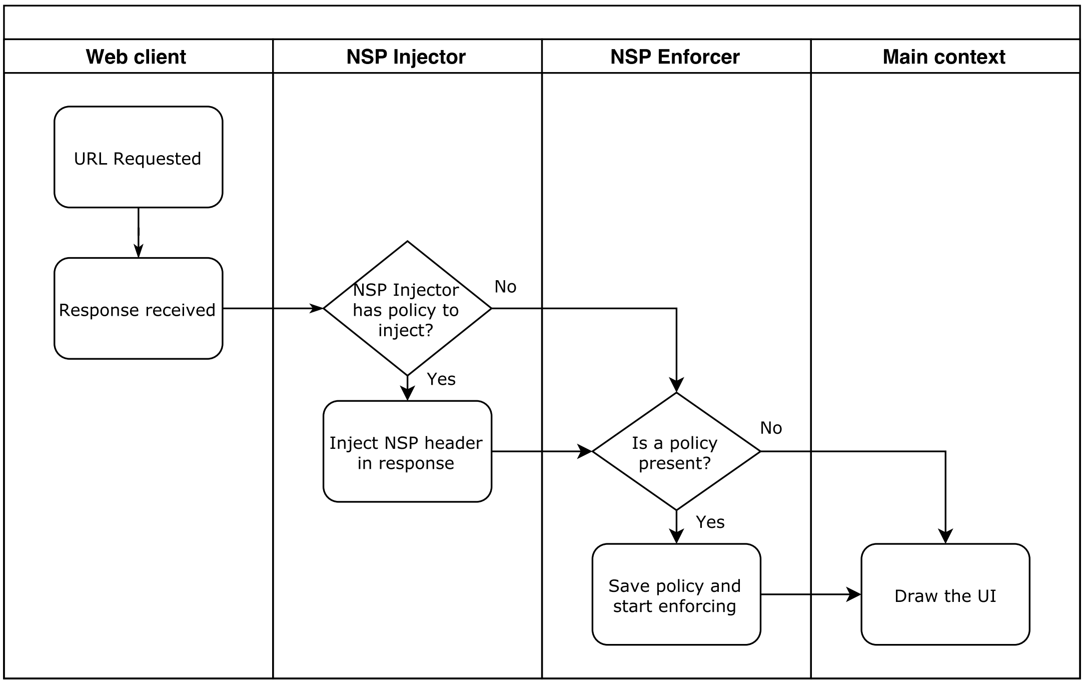


Figure 6.1: Request flow through the extensions

The evaluation is carried out on both user content driven and editorial web applications. User content driven web applications contain user content where users are allowed and encouraged to provide content, including web navigation, to the web application. Examples of such web applications are discussion forums, social networks, and web applications allowing users to post questions and comments. Editorial web applications have a trusted set of editors that are allowed to provide content to the web application while the majority of users are visitors that are only able to read the content.

For the evaluation `chalmers.se` and `stackoverflow.com` are selected as targets to secure with the NSP proof of concept. `chalmers.se` represents a class of web sites that contains content written by an authorised editor, which can be assumed to be benign. `stackoverflow.com` is chosen because it represents a web application with a lot of user content. Being able to apply policies that increase user security on such applications would be a positive result for this study.

6.4.2 The web application `chalmers.se`

The `chalmers.se` website contains mostly static editorial content since it functions as a repository for information regarding Chalmers University of Technology. A strict policy allowing nothing is assumed to start. There are 83 links on the front page which are all blocked at this stage. With `'client'` added in the list of destinations, 73 of these are eliminated, and ten remaining links point to destinations that are not allowed. It is assumed that all subdomains of `chalmers.se` are legal navigation, and this leaves six links to destinations that are not allowed. The policy at this stage of development is shown in Listing 6.1.

Listing 6.1: First policy for `chalmers.se`

```
{
    "allow": [
        "'origin'",
        "*.chalmers.se"
    ]
}
```

The six links to destinations not allowed are JavaScript anchors and all link to external resources. By adding these with their full path, the policy is kept as strict as possible.

Pages on `chalmers.se` follow a template with navigation menus and content on fixed points. To find links to destinations the policy does not allow in these, a random page is chosen that uses the described template. The chosen page contains information and links to all the master's degree programmes that are offered at Chalmers University of Technology.

When testing the current policy on this page, only one link to a destination on an external web application is not allowed. That web application is deemed to be benign and added to the allow list. The final policy is shown in Listing 6.2.

Listing 6.2: Final policy for `chalmers.se`

```
{
    "allow": [
        "'origin'",
        "*.chalmers.se",
        "http://www.ituniv.se/",
        "http://www.opic.com/org/chalmers_tekniska_hogskola",
        "http://www.chalmersstudentkar.se/en"
    ]
}
```

The policy for pages on `chalmers.se` ends up being small, involving only a few destinations. Since `chalmers.se` is an editorial web application where almost all content is trusted, a policy could possibly be generated based on the underlying content management system (CMS), or include a policy editor for each page. While the template is similar for many pages, the content is different and always trusted. This web application does not have the need for an intermediary page here as no user provided content should appear on the pages evaluated.

6.4.3 The web application `stackoverflow.com`

Stackoverflow is a website that contains a lot of user content. Users are allowed to post content using the markup language `Markdown`, and basic HTML with limited tags and attributes available. The ability to post content with markup enables users to post links that are made clickable on the client side. These links should be followed with care since they may take the user to an undesired destination. Besides user content, the website contains web navigation to static pages and to other pages within the Stackexchange network.

Stackoverflow is a good test subject since it mixes trusted navigation with untrusted navigation submitted by users. A policy needs to be flexible enough to not deny navigation to any page trusted by the the developers, while showing an intermediate page for user submitted navigation.

A desired policy for Stackoverflow would be a policy that allows internal web navigation of the Stackoverflow web application as well as web navigation to all web applications in the Stackexchange network. The policy would treat all navigation besides the previously mentioned as untrusted and use an intermediary page to either display information about navigating away from Stackoverflow, or have different behaviour based on a server-side decision.

The initial policy does not allow anything and uses the intermediate page as specified. This policy is shown in Listing 6.3.

Listing 6.3: First policy for `stackoverflow.com`

```
{
    "allow": [
    ],
    "intermediary-uri": "'client'"
}
```

In order to construct a suitable policy, the front page is first inspected, collecting all links supplied that are not part of user content. The front page contains most of the template used throughout the web application regarding web navigation such as the menu bar. These destinations are added to the set of allowed destinations.

At this stage 712 links remain that are not in the set of legal destinations. 605 of these 712 go to `stackoverflow.com`, the same origin. By adding `"'origin'"` to the list of allowed destinations, these are included in the set of legal destinations. Subdomains to `stackexchange.com` as well as web applications within the Stackexchange network are also added to the set of legal destinations. The policy at this stage is shown in Listing 6.4.

This policy only denies one link, to an external page with information about Creative Commons. This can safely be assumed to be an allowed destination, so that is also added to the set of legal destinations. Web navigation is not provided to any other destinations on that domain and as such the most secure policy is to use the full path.

As there are no more blocked destinations on the front page, the next step is to look at a page with user content. Stackoverflow has question pages that contain a question, answers to the question, and comments. To find what is part of the template for the page, a question containing no links, with no comments and with no answers was chosen. The page contains 187 destinations of which five are not allowed. Four of those are destinations to share the question on various social media platforms, as well as via email. The last one is to an advertisement network from an ad on the page. These URLs are added to the policy. Listing 6.5 shows the current

Listing 6.4: Second policy for `stackoverflow.com`

```
{
    "allow": [
        "'origin'",
        "stackoverflow.com",
        "*.stackoverflow.com",
        "stackexchange.com",
        "*.stackexchange.com",
        "superuser.com",
        "serverfault.com",
        "stackoverflow.blog",
        "askubuntu.com",
        "mathoverflow.net",
        "serverfault.com",
        "stackapps.com",
        "www.stackoverflowbusiness.com"
    ],
    "intermediary-uri": "'client'"
}
```

policy, suitable both for the front page and a question page.

Listing 6.5: Third policy for `stackoverflow.com`

```
{
    "allow": [
        "'origin'",
        "stackoverflow.com",
        "*.stackoverflow.com",
        "stackexchange.com",
        "*.stackexchange.com",
        "superuser.com",
        "askubuntu.com",
        "serverfault.com",
        "stackoverflow.blog",
        "mathoverflow.net",
        "serverfault.com",
        "stackapps.com",
        "www.stackoverflowbusiness.com",
        "https://creativecommons.org/licenses/by-sa/3.0/",
        "mailto:?subject=Stack%20Overflow%20Question",
        "http://engine.adzerk.net",
        "https://plus.google.com/share?url=http%3a%2f%2fstackoverflow.com",
        "http://twitter.com/share?url=http%3a%2f%2fstackoverflow.com",
        "http://www.facebook.com/sharer.php?u=http%3a%2f%2fstackoverflow.com"
    ],
    "intermediary-uri": "'client'"
}
```

To test this policy on a page with user provided content, a random page containing a question and answers with resources attached is chosen. With the policy applied to the chosen page, 408 destinations are discovered of which 62 are not allowed. These are all links to destinations submitted in the user provided question, answers, and comments. These should be navigated to via the intermediary page so the policy is deemed complete at this stage. The final policy can be used effectively on the front page and all pages that use the template for question pages.

Stackoverflow turns out to be a web application that has a great use case for the NSP, as it has a clear distinction between trusted content and user content. The use of an intermediary page serves as a way for Stackoverflow to either display a warning or do some server-side check on the destination before allowing the navigation. In this study it was decided to use the intermediary page provided by the NSP Enforcer, but in a live deployment any URL could be used as the intermediary page.

7

Discussion

This chapter brings up discussion regarding the NSP, the proof of concept implementation, the evaluation, the bigger picture in regards to web security, as well as the relevant ethical considerations.

7.1 Navigational Security Policy

There are several special considerations regarding the design of NSP, and practical policy implementations, which are discussed in this section.

7.1.1 Predicting web navigation

One difficulty that arises with the proposed NSP schema is that a web application must be able to fully predict all legal destinations beforehand. Imagine that between the time when a user visits some page, and the time when that user requests more content via JavaScript, the web application receives some new content with web navigation included. It is impossible for the policy received by the web client when receiving the DOM to contain the newly received content as it did not exist when the policy was created.

A web application with the desire that all provided navigation should be allowed, without an intermediary-page visit, faces a problem here.

A solution to this problem is for the web navigation to be made identifiable as safe, when included in the response to the request for more content via JavaScript. This is doable using cryptographic signatures for the destinations, including a public key that can be used to verify that the signatures are valid in the policy.

7.1.2 Redirects

A way to circumvent the protections added by NSP is to bounce navigation through an allowed page that responds with a redirect. Since the policy is unloaded as soon as a the context it protects is unloaded, it does not protect further than the first allowed request. An allowed page that responds with either a HTTP redirect, or a page that causes a redirect, can therefore be an issue for the NSP. Since the redirecting page is treated as a new context it is possible for it to provide a policy as well, but since the goal of such pages is often similar to the NSP intermediary page it is likely such policies will be less restrictive. Trusted web applications that enable open redirects pose a great difficulty for NSP deployment.

Redirects as a service, commonly known as link shorteners, also introduce problems. Allowing a link shortener in the navigational policy of a web application means that anyone can use the link shortener to bounce via when circumventing the NSP. Therefore it is important in NSP deployments to only allow specific paths of such services, or not allow them at all.

7.1.3 Different ways of delivering the policy

There are multiple ways of delivering a policy to the web client. In the proof of concept a header that supplies a policy for each page load is used. Policies could also be stored at a web server and fetched, with the header only providing a reference to the policy location, which is fetched individually. A policy may also be defined in a manifest file for the web application, such as the proposed Web App Manifest that aims to store all web application meta data in a single file [12]. By not appending a policy in each request, the overhead could be decreased due to the ability for the client to fetch the policy independently and cache policies for web applications that either uses a site wide policy, or for pages where the policy has already been fetched.

7.1.4 Generating policies

Web applications that provide dynamic or user generated content may face issues defining an application wide static policy if the wish is to cover all navigation that the web application provides. In these situations there are a few choices available, including but not limited to; using an intermediary page for the non-static web navigation, dynamically generating a policy based on the content of the generated DOM, or not using a policy at all.

All web applications have the desire to ensure that only web navigation that is intended by the application is followed by the client. Thus, all web applications stand to benefit from utilising NSP. Intermediary pages are a good solution for most web applications providing user-generated content, but may not be desirable in all applications. Dynamically generating a policy based on the content of the generated DOM is an option for such applications.

When generating dynamic policies it is important to consider the security implications what is being done. Automatically adding all web navigation in the DOM implies the same level of trust for user-generated content as for web navigation that is provided by the application itself. Such an NSP deployment would protect users against DOM-based XSS, but not reflected or persistent XSS, as the injected web navigation would be included in the dynamically generated policy.

Dynamic generation of policies could be beneficial when integrated into a CMS. The CMS itself can determine CMS-wide legal destinations, the CMS theme can in turn determine the theme-wide legal destinations, templates provided by the theme can determine any destinations that are common to that specific template, and finally anything that is actual content for the template can be safely parsed for legal destinations, as it is all provided by an authorised user of the CMS system. Put together, these parts provide the CMS a safe way to dynamically generate an adequate policy for each request.

7.1.5 Application-wide versus page-wide policies

There are two approaches to creating policies; application-wide policies and page-wide policies.

An application-wide policy is the same for each web page of a web application and must cover all web navigation in the entire web application. These policies can therefore include a lot of destinations. It is believed that these policies more often will use the `intermediate-uri` directive to avoid blocking web navigation to destinations not present in the `allow` block, while still offering increased security.

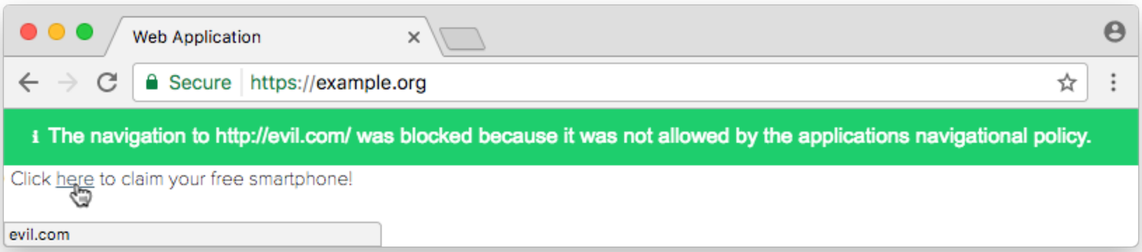
Page-wide policies can be smaller, and unique per web page. This makes each page easier to manage for one person, but in return forces a larger amount of policies to be maintained.

7.1.6 User interface considerations

While it is known that an NSP-enabled web client is able to restrict navigation according to a server defined policy, it is not certain how this should be reflected in the user experience.

The simplest strategy is to just stop the navigation without informing the user at all. The downside of this is that it is likely to confuse users, and in the extreme case may even cause them to believe that the web application itself is to blame, potentially resulting in decreased trust in the web application.

An improvement on this strategy would be to introduce some visual feedback to the user when the NSP has blocked a navigation. Figure 7.1 shows an example of how such visual feedback could look.



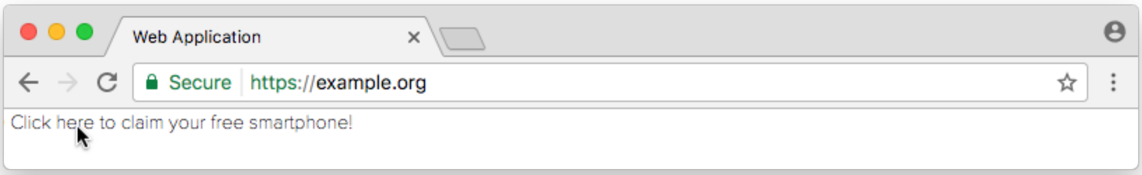
ℹ The navigation to http://evil.com/ was blocked because it was not allowed by the applications navigational policy.

Click here to claim your free smartphone!

evil.com

Figure 7.1: Example of using an information bar as visual feedback for blocked navigation

Another strategy is to remove all the visual signs of illegal navigation from the rendered page, as is demonstrated in Figure 7.2. In the given example, the content “`<a href="http://evil.com">here</a>`” was changed by the NSP implementation to “here”, prior to being rendered in the web client view.



Click here to claim your free smartphone!

Figure 7.2: Example of removing the link as visual feedback for blocked navigation

7.1.7 Responsibility for meaningful policies

A core concept in web navigation security is legal destinations. As discussed in the thesis, the web application server must be able to define a set of legal destinations that the client side can then enforce.

When defining how that set of destinations is specified it is up to the schema developers to decide the level of flexibility that should be possible. As detailed in Section 2.4, destinations in NSP use the format `[scheme:][[//]service][/path]` for URLs, and allows the policy creator to specify any or all of the `scheme`, `service`, and `path` components on their own to denote one set of destinations.

This is a highly flexible approach as is desired, but it opens up for a couple of rather useless destination sets. Consider the destination set defined by `"/security"`. This is a legal set of destinations specifying any `scheme` and `service` combination that contains legal characters for a URL, as long as the path begins with `/security`. Is this a useful destination set for a web application to define as legal? Probably not. The question for the schema developers is whether or not such sets should be legal at all.

The NSP schema, as defined in Chapter 4, applies no such restrictions. On the one hand this is a highly flexible approach that empowers policy creators and may enable useful destination sets not predicted by the schema developers, but on the other hand this approach also enables the `"/security"` example above. For the NSP it is believed that the benefits of a flexible approach outweigh the disadvantages of enabling useless policies.

7.2 Proof of concept

Implementing the proof of concept solution came with certain challenges. This section details the considerations involved in the decisions we made when designing NSP Enforcer.

7.2.1 Implementation design choices

Before implementing the proof of concept solution the choice had to be made between building an implementation directly into a browser, or building an extension.

Both of these choices have positive and negative aspects in regards to NSP implementation. Choosing to build the implementation directly into a browser gives the freedom of being able to implement any kind of functionality desired without unwanted side-effects. But it is more difficult and takes a longer time. Choosing to build an extension means lower complexity, but sacrificing some functionality that is not be possible using the Chromium extension API.

Designing a good schema is of higher importance than building a proof of concept implementation, and the time required to build the implementation directly into Chromium would not have allowed enough focus on designing the schema. For these reasons it was decided to build the implementation an extension.

7.2.2 Working with limitations

As it was decided to build the proof of concept implementation in the form of a Chromium browser extension, this meant that any functionality desired had to be possible using the Chromium extension API.

Building a system designed to block navigation in the manner which is required by the NSP schema, without introducing unwanted side-effects, is impossible.

One such side-effect is that web navigation using the address bar is unintentionally filtered if NSP is deployed by the application. During the implementation phase, a workaround was designed that enables the user to navigate to destinations outside the set of legal destinations by using an `omnibox` keyword.

The resulting implementation works great as a proof of concept, correctly blocking illegal destinations from being navigated to in web applications that supply a policy. However, the limitations of implementing a browser extension, rather than building a complete browser integration, render the proof of concept implementation difficult to work with and unsuitable for deployment.

7.3 Evaluation

The evaluation shows that it is feasible for a web developer or administrator to create page-wide policies, or application-wide policies for web applications that contain only a few pages. However it becomes clear during the evaluation that such a way to create policies is not optimal for large scale web application since each page needs to be parsed and examined. For a large number of pages, it quickly becomes infeasible.

The evaluation covers the most naïve approach to creating policies, static policies for a one or a few pages. Since there exist four ways (static application-wide, static page-wide, dynamic application-wide and dynamic page-wide) to create policies, the evaluation only covers a small part of policy creating.

The selected groups and targets is highly relevant. The sample size for the evaluation is small and could have been larger. Combining the different approaches to creating policies with a larger scale empirical study would increase the usefulness of the study.

7.4 Ethical considerations

This work interferes with certain aspects of the normal operation of web applications, both through the design of the proposed schema but also through the research itself.

7.4.1 Navigational Security Policy used for tracking

It is possible to use the NSP to track a users navigation within a web application, and also when the user navigates away from the web application. One such way of tracking users would be to not allow any navigation and configure the policy to be report only. By doing this, the client will report all violations while not actively enforcing them. One way to avoid this problem is to omit the `report-uri` directive, but that in turn removes the ability to detect problems with either the web application or the policy.

The intermediate page could also be used to track users. Similarly as the previous example, a policy stating that all navigation (or all out going navigation) should go through an intermediate page that the web application control enables that application to track how and when the user navigates. This could be avoided if the intermediate page is not fetched from the web server, but instead part of the client. But imposing this limitation could decrease the over all usefulness of an intermediate page.

7.4.2 Effect on existing services

During the testing of NSP using the proof of concept, real world web applications were analysed. This testing did in no way harm or affect the web applications tested since the policies were applied on the client side. The NSP Injector extension only added a header locally, while the NSP Enforcer only restricted web navigation in the web client that installed the extension. As such, no real world web applications were harmed in the making of this thesis.

7.5 Mitigation vs. prevention

While the NSP mitigates the problem with web navigation that is unwanted by the web application, it does not attempt to mitigate similar problems nor fix the common root cause.

The web of today has several fundamental design problems that enable the closely related security problems; content injection, HTML-injection and XSS. The common denominator with these problems is that an attacker has found some way of including a string of characters that is interpreted as code in the web client. While the root cause is the same, there are a large number of vulnerabilities that can cause this. The vulnerabilities can be both in the web developers code or in the code of the underlying technology stack, making it hard to fully secure any web application. The same can be said about the problems this causes. Ranging from injecting simple text to fully hijacking a web application and changing content or stealing user information.

This puts the work on the NSP into perspective. The solution fulfils its purpose which is to mitigate attacks on web navigation. While an attacker might not be allowed to inject web navigation, a huge number of other attacks are still possible that may be worse. In the long run the web needs an improved architecture to more securely isolate and sandbox web applications.

7.6 Related work

The idea of providing policies in a separate channel to enforce behaviour in the web client is not new. However most of the works aim to mitigate all or a large set of the problems. No such works consider just the problem of web navigation as something that should be governed by a policy.

Stamm et al. presents CSP, a security policy designed to mitigate vulnerabilities in a web application [13]. CSP achieves this by restricting web clients from including content based on an application-defined policy delivered via an HTTP header. This work introduces the notion of distrusting the web application sent to the web client and using a policy delivered via a secure channel to enforce restrictions on what is delivered to the web client. While CSP is a great step to a more secure web, it does not prevent injected HTML from being interpreted as code and therefore does not prevent navigation from being introduced.

Oda et al. introduces a policy to control information flow between domains [14]. It uses approval from both the sender and receiver before the web client fetches the content from an external server. The policy is not transferred to the client, instead it asks the origin and external server for approval receiving a simple yes or no answer. Their paper is related to this work since it deals with the idea of client enforced policies supplied by the web application. However, the paper explicitly excludes navigation since hyperlinks are not affected by the mechanism.

A way to assure the integrity of the HTML document sent to the client is proposed by Van Gundy et al. where they describe how nonces can be used to indicate that tags in the HTML are trusted [15]. The web server creating the markup inserts the randomised nonce in each tag leaving user content inserted tags without the nonce and therefore interpreted as data, instead of code. The paper by Van Gundy et al. relates to the work on the NSP in the aspect that they establish trust between the server and client for some type of data. Using their work a web client can determine if a hyperlink in the DOM was intended by the server, as it is accompanied by the correct nonce. Their approach is different from the one proposed in this thesis due to the use of nonces instead of an application-defined policy.

Bates et al. evaluates ways for the web client to use filters to prevent XSS, an attack that aims to execute JavaScript through inclusion of JavaScript code [16]. They discover that XSS filters were either slow or easily circumvented, and in some cases even introduced security problems. There are also cases where false positives arose, breaking functionality in web applications that were not vulnerable. These findings show once again that a one size fits all mitigation for vulnerable web applications is hard to achieve. While XSS protection in the web client increases difficulty level for an attacker, it does not eliminate the problem.

Client side XSS can also be mitigated through an external application running on the client device. Kirda et al. proposes Noxes, a web proxy that performs inspection on HTTP traffic in order to find XSS attacks [17]. Heavily inspired by firewalls, it aims to increase security while requiring none or very little user interaction. This solution is a bit outdated, but the idea is highly relevant today. Using a middlewear to protect the user can work but, as mentioned earlier, is prone to false positive and new security vulnerabilities. Applying policies that are unique to each web application is a superior approach as it diminishes this effect. On the other side, such policies require the web application to actively prioritise security which introduces more work for the developer.

Several papers cover a fundamental problem of the world wide web, that modern web applications are very different from the static pages that the creators of the world wide web originally envisioned. On a more abstract level, Reis et al. wrote about how web security is related to four key problems in the architecture of the web [18]. The paper describes web security from a perspective where the web is a collection of web applications. This thesis takes on the problem of web application isolation, as discussed in the paper, by allowing a web application to separate navigational policies from data. The NSP is a concrete solution to some of the problems discussed but not solved in the work by Reis et al.

8
Conclusion

There is no method for applying application-defined navigational security policies on web clients in the current state of the art. Existing solutions only cover the wishes of the client, not the web application.

This thesis presents the Navigational Security Policy (NSP), a proposal for a solution to this problem. The NSP provides several directives enabling the web developer to define a policy containing the set of legal destinations. This policy is then applied to any web navigation available from a certain web context. Furthermore, the NSP also defines directives used to inform the client how to handle policy violations, introducing the concept of client enforced intermediary pages and violation reporting.

This work on the NSP, and the proof of concept implementation, demonstrates that the concept of an application-defined policy for web navigation is feasible. This is shown by performing a case study on real world web applications, where it is envisioned what a policy for that web application should look like.

A suggestion for further work is looking at ways to solve the difficulty surrounding the creation of navigational policies for web application with frequently updated user content. Considering a web application which updates its content without ending the web context illustrates the difficulty well. While resource heavy, it is possible to mitigate this by building an intermediary page which verifies the navigation. Another approach to solving this difficulty could be for the web server to include cryptographic signatures with any web navigation, allowing web clients to verify the legality of the destination locally. Another suggestion is to build an implementation of the NSP schema without the negative side-effects that an extension implementation comes with. Finally, conducting a more extensive study on deploying the NSP in real world applications would provide more insight into the policy construction process. Such a study could include applications from multiple different categories and attempt to build meaningful policies in an automated manner.

Bibliography

[1] S. Faulkner, I. Hickson, S. Pfeiffer, E. D. Navara, R. Berjon, T. O'Connor, and T. Leithead, "Html5", W3C, W3C Recommendation, Oct. 2014. [Online]. Available: `http://www.w3.org/TR/2014/REC-html5-20141028/`.

[2] D. Akhawe, A. Barth, P. E. Lam, J. Mitchell, and D. Song, "Towards a formal foundation of web security", in *Computer Security Foundations Symposium (CSF), 2010 23rd IEEE*, IEEE, 2010, pp. 290–304.

[3] M. Zalewski, "Postcards from the post-xss world", 2011. [Online]. Available: `http://lcamtuf.coredump.cx/postxss`.

[4] B. Sanou, "The world in 2016: ict facts and figures", *International Telecommunications Union*, 2016.

[5] T. Berners-Lee, L. M. Masinter, and R. T. Fielding, *Uniform resource identifiers (uri): generic syntax*, RFC 2396, Aug. 1998. DOI: `10.17487/rfc2396`. [Online]. Available: `https://rfc-editor.org/rfc/rfc2396.txt`.

[6] T. Berners-Lee, R. T. Fielding, and L. M. Masinter, *Uniform resource identifier (uri): generic syntax*, RFC 3986, Jan. 2005. DOI: `10.17487/rfc3986`. [Online]. Available: `https://rfc-editor.org/rfc/rfc3986.txt`.

[7] L. M. Masinter, D. Zigmond, H. T. Alvestrand, and R. A. Petke, *Guidelines for new url schemes*, RFC 2718, Nov. 1999. DOI: `10.17487/rfc2718`. [Online]. Available: `https://rfc-editor.org/rfc/rfc2718.txt`.

[8] *Making the web safer*, `https://www.google.com/transparencyreport/safebrowsing/`, [Online; accessed 2017-05-16].

[9] M. West, D. Veditz, and A. Barth, "Content security policy level 2", W3C, W3C Recommendation, Dec. 2016. [Online]. Available: `https://www.w3.org/TR/2016/REC-CSP2-20161215/`.

[10] L. Weichselbaum, M. Spagnuolo, S. Lekies, and A. Janc, "Csp is dead, long live csp! on the insecurity of whitelists and the future of content security policy", in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ACM, 2016, pp. 1376–1387.

[11] E. Ecma, "404 the json data interchange standard", *Technical specification, Ecma International*, 2013.

[12] M. Caceres, K. Christiansen, M. Lamouri, and A. Kostiainen, "Web app manifest", W3C, W3C Working Draft, Mar. 2017, https://www.w3.org/TR/2017 /WD-appmanifest-20170302/.

[13] S. Stamm, B. Sterne, and G. Markham, "Reining in the web with content security policy", in *Proceedings of the 19th international conference on World wide web*, ACM, 2010, pp. 921–930.

[14] T. Oda, G. Wurster, P. C. van Oorschot, and A. Somayaji, “Soma: Mutual approval for included content in web pages”, in *Proceedings of the 15th ACM conference on Computer and communications security*, ACM, 2008, pp. 89–98.

[15] M. Van Gundy and H. Chen, “Noncespaces: Using randomization to enforce information flow tracking and thwart cross-site scripting attacks.”, in *NDSS*, 2009.

[16] D. Bates, A. Barth, and C. Jackson, “Regular expressions considered harmful in client-side xss filters”, in *Proceedings of the 19th international conference on World wide web*, ACM, 2010, pp. 91–100.

[17] E. Kirda, N. Jovanovic, C. Kruegel, and G. Vigna, “Client-side cross-site scripting protection”, *Computers & security*, vol. 28, no. 7, pp. 592–604, 2009.

[18] C. Reis, S. D. Gribble, and H. M. Levy, “Architectural principles for safe web programs.”, in *HotNets*, Citeseer, 2007.

A

The blockchain.info CSP

```
img-src 'self' data: https://blockchain.info https://www.google-analytics.com;
style-src 'self' 'unsafe-inline';
frame-src 'none'; child-src 'none';
script-src 'self' https://www.google-analytics.com;
connect-src 'self' *.blockchain.info wss://*.blockchain.info
    https://blockchain.info;
object-src 'none';
media-src 'none';
font-src 'self';
```