Imperial College
London

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

Using Content Security Policy Report Only to Detect Malicious JavaScript

Author:
Michal Glinski

Supervisor:
Dr. Sergio Maffeis

Submitted in partial fulfillment of the requirements for the MSc degree in Computing (Security and Reliability) of Imperial College London

September 2023

Abstract

Cross site scripting attacks are one of many ways a malicious JavaScript file can be included in the users browser. Scripts provide attackers a fully automatic framework to exfiltrate user data or to perform actions in the name of an unsuspecting victim. As clients expect more features to be available online, browsers contain more sensitive information than ever before. Services such as banks and payment services spend significant amount of resources to guarantee the security of their services. Testing through different test suites to ensure significant coverage of their code, or hiring dedicated security specialists to manually test their live product are among many ways such companies try to protect their clients from exposing their data. Many of those methods work by performing an exhaustive search on the host.

The goal of this work is to develop an autonomous server which persistently scans for new and modified scripts running on the hosts website. By using Content Security Policy Report Only, the server may involve genuine users to guarantee maximum coverage. By exclusively using the non enforcing version of CSP the server can achieve the highest form of security without compromising user experience or hindering development of new features.

Contents

Chapter 1

Introduction

As of writing this report the IP/TCP protocol which functions as a back bone of the internet was first demonstrated exactly 40 years ago. The HTTP protocol which we use to transfer web application was created shortly after. Throughout those 40 years the the web landscape has change multiple times. Currently every single service imaginable can be accessed online, from banking, through education, to entertainment. Everybody across the globe has grown used to the vast possibilities that the internet provides.

With all the positive impact of the internet there are also risks. As all our data has at some point been exposed to the World Wide Web, there have appeared bad actors who are interested in collecting and abusing our information. A common way of extorting information is to access it directly in the users browser. Old reports claiming that 65% of websites are vulnerable to Cross Site Scripting(XSS) attacks [12] and newer reports showing that almost 40% of all attacks are performed using XSS [13]. Hacker one, one of the companies that unites ethical hackers and provides other security services, ranks XSS as the most rewarded vulnerability [19].

This report, after a brief introduction to cross site scripting and content security polices(CSP), presents a fresh and innovative approach to detect ongoing attacks. After detailing the inner workings of the newly designed system, it is tested by simulating real world scenarios which use currently accessible websites. After the results are collected, the paper concludes with a discussion on many third party issues that were encountered throughout the development which if resolved could increase either security or performance of the final product.

Chapter 2

Background

2.1 Cross Site Scripting

Cross Site Scripting is an injection type attack, where the malicious user exploits a vulnerability to execute JavaScript code in the unsuspecting client's browser. XSS is one of the oldest vulnerabilities with many different ways an attacker can abuse the web infostructure. Even after the code that allows for the attack is fixed, the impact if the it needs to be examined as each breach can have a different motive. OWASP foundation classified Cross Site Scripting as the seventh most critical security risk in their OWASP Top 10 report for year 2017 [9]. In year 2021 OWASP Top 10 team has moved XSS to position 3 and they have merged in with other injection type attacks [18].

2.1.1 Types of Cross Site Scripting

As Cross Site Scripting is rather a result of malicious behaviour, the attack types are grouped by their root cause. Types of XSS are grouped by whether they target the server or the client infostructure. They are also grouped by the way the payload is delivered to the user, as it can be sent in a one-off request or stored in a database. Depending on the type there are different ways to protect and detect such attacks.

Reflected XSS

Reflected XSS happens when the server sends the content of the request back to the user without proper sanitization. This form of an attack is often performed in combination with a phishing campaign. In this case the attacker would try to convince the user to click a carefully crafted link, which due to the reflated nature, executes the script encoded in the request. As they payload almost always needs to be included in a GET query the attack leaves a significant trace in the server logs. The best way to detect reflected XSS is to monitor the server logs either manually or with one of many automatic tools.

Stored XSS

Stored Cross Site Scripting attacks happen most often on forums, message boards or in the comment sections of articles. They are possible when the server does not sanitize the input of the malicious user before it is displayed to the victim. This attack is particularly dangerous as it can affect any passerby and when executed may go unnoticed, while slowly collecting sensitive user data. A good way to detect it is to routinely perform database audits to check for any unexpected values.

A more dangerous type of Stored XSS is when the payload is saved directly in the users browser. This can be accomplished by using a HTML5 database, localStorage or browser cache. This attack can be untraceable from the servers perspective as all data is stored and executed on the client side. The approach to client protection in this paper allows for detection of this form of attacks, but it is unable to retrieve the exact payload.

DOM Based XSS

DOM Based XSS attack happens when the website tries to display parts of the request query itself. Compared to reflected XSS where the server is exploitable, in DOM based XSS the client side code is to blame. With more advanced techniques, the payload in the request can be hidden from the server [10], resulting in hard to detect attacks. Similarly to stored XSS the defence techniques described in this paper may help detecting such attacks as they focus on client side protection.

2.1.2 XSS Attack goals

Cross site scripting is a very serious risk for hosts, as a successful attack may lead to confidential information disclosure. Through XSS attackers gain control of the JavaScript engine. With this control they may accomplish multiple goals to eventually compromise the user.

Steal session cookies

If session cookies are not HttpOnly, which prevents them from being accessible by scripts, the attackers may try to retrieve them. By doing so, they will be able to authenticate themselves as the victim and perform any actions they could do. This gives the attackers total control over the account, from which they can extort money or sensitive information. Oftentimes banks and other institutions protect their clients by requiring additional 2 factor authentication method before any data is presented or before any transfers are executed.

Impersonate the user

When session cookie is unavailable, the attacker may still retrieve compromised users data and perform actions within the browser. They may perform requests in the name of the user and the browser will automatically include the HttpOnly cookie to the request. In this way attackers' script may retrieve or even change the users data. They may interact with the site by clicking buttons or filling out forms and depending on the attack type even persist between sessions.

Deface or modify the application

The attacker may also change the website to trick the user into inputting their credentials. this is especially prevalent if the attack is only available on a specific part of the website, where the user may be unauthenticated. As in other goals, the attacker eventually tries to obtain the full access to the account. In the end if the website does not have any authentication process, the attacker may be interested in causing financial or reputation loss to the host. To achieve this goal they may remove all content from the site or insert offensive imagery.

2.2 Content Security Policy

Content Security Policy(CSP) was introduced as a protection layer, which allows web hosts to control origin and type of resources loaded in the client browser. W3 organization recommends using CSP as a defence-in-depth tool, which can help reduce the harm caused by malicious users, but it should not be the sole security measure taken to prevent attacks. [7] CSP works by providing the user browser a list of directives, where each resource type, like scripts, style sheets and images, have its own set of rules. Those can include allowed URL endpoints, hashes of expected objects or nonces which may allow for otherwise unsafe inline scripts.

2.2.1 CSP Use Cases

Content Security Policy is currently used to protect against three major attack vectors within browsers.

Cross Site Scripting

The most common usage of CSP in literature is to limit the resources loaded to prevent attacks such as XSS. By using allowlists which include only the trusted hosts, the site can prevent scripts from being loaded from malicious URLs. If the site were to remove all instances of inline scripts, they may block all inline scripts from executing, blocking many DOM based and reflected XSS attacks.

Cascading Style Sheets attacks can be bundled together with XSS as CSP provides very similar set of mechanisms to block both of them. Both of them distinguish inline scripts or styles and in newer CSP version they both get `-elem` and `-attr` suffixes. CSS attacks are very new and currently mostly seen in theoretical examples, as They require big carefully crafted payloads. Still, CSP is theoretically capable of stopping these kind of attacks.

Packet Sniffing

To protect users from packet sniffing, developers can use `upgrade-insecure-requests`. Using this directive will instruct the browser to send all requests via https, which will encrypt all data transferred. If the browser failes to upgrade the request the resource will fail to load. This directive prevents any 3rd party from reading confidential information that is sent in unencrypted requests.

Using CSP with `default-src https:` will drop all insecure requests without trying to upgrade them. If this is combined with a reporting directive, the host is able to find all insecure endpoints which should be promptly upgraded.

Click jacking

CSPs can also protect from click jacking and phishing attacks with specialized frame directives. `frame-src`, `frame-ancestors`, `sandbox` all control what behaviours the application may allow.

1. `frame-src` controls which URLs can be included as frames in the website.
2. `frame-ancestors` controls what websites can include this page in a frame.
3. `sandbox` controls what the loaded website can perform as if it was loaded into an sandbox iframe. With this it is possible to forbid downloads or scripts all together.

Paper by Roth Et Al. analyzes tendencies in CSP deployment between 2012 and 2018 and shows increased interest in deploying CSP to limit framing options of websites. [17] It is theorised this may be due to the limited control that `X-Frame-Options` provide, as CSP allows to specify frame origins and is better supported by browsers. Developing a CSP with framing options is also relatively easier, as there are fewer directives, fewer sources and the attack surface is smaller.

2.2.2 Content Security Policy version 3

CSP is constantly being developed with currently 3rd version being in draft. Major parts of it are already implemented into all modern browsers with computer based browsers often implementing many features before mobile ones. With CSP3 `scirpt-src` directive has been further split into `script-src-elem` and `script-src-attr` directives. These directives allow hosts to have separate rules for JavaScript coming from `<script>` markup tags and attributes such as `onClick` event handlers. Additionally it allows to specify hashes as sources to allow only matching scripts to run. This new feature is heavily used within this project to achieve the highest possible security. CSPv3 has also brought enhancements to the `sandbox` directive, which allows for more control over potentially untrusted content running in its isolated environment.

The newest experimental feature of CSP3 are `require-trusted-types-for` and `trusted-types` directives. [15] [8] To further prevent DOM based XSS attacks, they allow developers to control data that comes into unsafe sinks. By using specially designed parsers and filters all elements can be checked before they are dynamically added to the website. In this way trusted types reduce the attack surface even further and require the attacker to work much harder when creating an exploit.

CSP3 is designed to be backwards compatible with CSP2, which is to ensure that new features do not prevent web pages from displaying correctly on older clients. For example when `strict-dynamic` source is included, the `unsafe-inline` rule is ignored by the browsers. `strict-dynamic` forces the browser to only use nonce and hash based sources. It also permits any dynamically loaded inline script, as long as it was loaded from another nonce or hash allowed script. In that case `unsafe-inline` may be necessary for users using older browsers that do not support `strict-dynamic` source. Otherwise all dynamically loaded scripts would be blocked, leading to poor user experience. This behaviour allows for new features to be deployed sooner, but it sacrifices the security of older clients.

2.2.3 Implementing CSP

Implementing CSP, although potential security benefits, comes with its own fair share of problems. They span multiple layers and are especially noticeable when trying to add CSP to a big and long lasting project.

One of the hardest problems to overcome is when the web application uses inline scripts or event attributes that allow for scripting, like the `onClick` event handler. While it is possible to still use inline scripts with hashes or nonces, CSP disincentives the usage of inline JavaScript as it is a common entry point for XSS attacks.

Strong cryptographic security of hashes ensures that only allowed scripts will execute. Attacker in this situation can execute the same script multiple times and abuse the side effects that it may have. When using `unsafe-hashes`, hashes can also be used for event handlers, which still may prove to be better than nothing. The biggest downside of CSP hashes is when using them to secure 3rd party sources, as the CSP will need to be updated with every source code change.

The other option requires that every script is accompanied with a random nonce value. In theory when nonce is regenerated with every response, the attacker will be unable to execute scripts as they will be unable to predict the value. Unfortunately, nonces have been shown to be potentially insecure and fall to more advanced attacks. [14] Those attacks oftentimes rely on confusing the browser, which despite invalid html syntax still tries to display the page.

All the issues and methods can also be applied to style-sheets, as those may also be DOM injected in similar manner to JavaScript. [16] [11] With more modern attacks that may use pure style-sheets to exfiltrate data about users, it is important to secure CSS sources too. While those attacks are usually more complex in execution and the payload size, they can still be mitigated by a well designed Content Security Policy.

Another big problem when developing a CSP is enumerating legitimate dependencies of the website. Addition of new dependencies may go unnoticed in a quick developing environments of many web applications. Users may also inject their own code in form of plugins into browsers, which will generate false reports. This behaviour is well highlighted by a blog post by Dropbox when they were developing a CSP for their own application. [5] The Dropbox team had to create a list of strings, some of which included endpoints of other APIs, to filter those reports.

While deploying a CSP developers may also use tools already available on the web. `csp-evaluator.withgoogle.com` allows for checking the CSP or the website itself against a big set of vulnerabilities, described in a paper by Weichselbaum Et Al. [20] Those were collected in a comprehensive research aggregating CSP usage and vulnerabilities which can result in the attacker bypassing the policies.

Hosts may also use one of many commercially available CSP generation tools. `csper.io` helps in deploying and maintaining a CSP header by automatically gathering, filtering and sorting reports. Tools like that can increase the overall security of the website, but automation may not be perfect too. It may help create a CSP with holes created by plugins loaded by users. This can later lead to a CSP bypasses where the malicious user leverages small misconfigurations to gain control. Additionally when an attack is already ongoing it may be difficult to differentiate its reports from genuine traffic.

When deploying a Content Security Policy it is important to remember that maintaining it requires constant effort from the developers. Unupdated CSP can make the web page misbehave or be completely unusable when required sources are blocked from loading. This is a risk that may negatively impact user experience, which may make some opt out of using a CSP.

```
upgrade-insecure-requests ;
default-src 'none' ;
script-src 'report-sample' ;
script-src-elem
'report-sample'
https://testing.site/index.js
'sha256-f2KDab0BheatnkXlUGlWthbaCaKmlDSpV5682oXCFAw='
'nonce-0EdqEL1Z';
script-src-attr 'report-sample' ;
frame-src testing.site ;
font-src *;
img-src https: ;
report-uri https://reporting.project
```

Figure 2.1: Example Content-Security-Policy, it upgrades all requests, allows for couple of scripts and demonstrates how CSP sources are organized

2.2.4 Content Security Policy Issues

Even though CSP can prevent many basic attacks it may be unsuccessful at blocking more advanced exploits. A study by Weichselbaum Et Al. done in 2016 showcases many ways in which CSP may be ineffective. [20] It also performs a comprehensive analysis on CSP directives which concludes that most of them are trivially bypassable through methods they have introduced. A lot of the insecure CSPs in the paper are using directives that allow the execution of inline scripts. Inline JavaScript in event handler attributes is a big part of DOM injection based XSS attacks.

Another issue within currently deployed policies is the lack of `report-uri` directive. Without it the host will never be aware of any CSP violations, as the browser is unable to send reports. It is reported that only 22% of websites from top 10,000 that use CSP have a `report-uri` directive. In our own tests done on top 1 million sites provided by NetCraft, only 4% of policies have a `report-uri` directive. Additionally only 50% of `report-only` policies have a `report-uri` directive.

Although Content Security Policy was never intended to protect against data exfiltration attacks, the paper by Acker et. Al shows how even with very strict policies data can still be exfiltrated from user's browser by using DNS prefetching.

2.2.5 Report-Only Header

Content Security Policy offers many benefits to protect users from attacks by selectively allowing safe resources to be loaded. But as established it takes a lot of team effort to implement and maintain, where small errors can lead to loss of security or user experience. Thankfully CSP provides one more tool in form of the report only header. It simulates the browser enforcing the CSP and allows for all the violations to be reported to the specified endpoints.

This tool is of major help when starting to create a new policy for a website. It allows for testing and fine tuning of the policy before enabling it. The reports provide great insight into the structure of the application. Additionally disabling all resources from loading is a quick and dirty way of obtaining a list of all resources that are loaded by clients. The Dropbox team used the report only feature extensively when deploying their own CSP, where they enumerated sources and tested their own CSP. [5]

Report only violations can give a valuable insight into attacks that are currently ongoing With careful monitoring attacks may be detected quicker, allowing developers to patch the issue and limit the impact of the adversary. Additionally reports provide many insights into the violation, the type of the resource, the URL and even the line in which a rule was broken. With the `report-sample` source, the report may include the first 40 characters of inlined script or style-sheet. An example report which showcases all those features can be seen in figure 2.2.

In literature CSP reporting seems to be an underutilized tool, as less than one in five websites that deploy CSP ever deployed CSPRO header. [17] This may correspond to the poor effectiveness of deployed CSPs as developers may have never tried more strict policies in report only mode. Even less websites use the report-only header alongside CSP header, which may be one of the very few ways to progressively try and remove old dependencies from the already deployed CSP.

The CSP report-only header can be added to any website without consequences to the user experience and without modifying large pieces of code. This may lead to overall increase to the security by solving security issues quicker. It can also be deployed alongside an existing CSP to more strictly monitor sources which are known to be temporary or problematic. Those may include external APIs or libraries, which may update and add new features which compromise security.

```
{"csp-report":{
"document-uri":"https://testing.site/index2.html",
"referrer":"https://testing.site/index2.html",
"violated-directive":"script-src",
"effective-directive":"script-src-elem",
"original-policy": Full policy redacted for brevity,
"disposition":"report",
"blocked-uri":"inline",
"line-number":11,
"column-number":13,
"source-file":"https://testing.site/index2.html",
"status-code":200,
"script-sample":"document.write('hello')"
}}
```

Figure 2.2: Example report, they provide necessary information to classify the violation. Reports may also include line and column numbers of the violation alongside the scipt sample.

2.3 Ethics

The project focuses on a server which is designed to collect and act on reports potentially sent by genuine users of the web applications. When the reporting endpoint is controlled by a third party, the reports sent will expose it to the queries performed by said users of the applications. Usually reports only provide benign information for loaded resources, but occasionally sensitive data may be sent using GET queries. In the report, this query will be included within blocked-uri field, which may look like this:

```
https://id.cxense.com/public/user/id?json=%7B%22identities%22%3A%5B%7B
%22type%22%3A%22ckp%22%2C%22id%22%3A%22lm9kdx9144wtgfya%22%7D%2C%7B%22
type%22%3A%22cst%22%2C%22id%22%3A%223dmijrrz7kkn62ie7ib1yw5z6l%22%7D%5
D%2C%22siteId%22%3A%221136227972865927410%22%2C%22location%22%3A%22htt
ps%3A%2F%2Fwww.libertatea.ro%2F%22%7D&callback=cXJsonpCB2
```

Sensitive data exposure through GET queries is a known vulnerability, which was ranked as the third most critical vulnerability by OWASP top 10 in 2017 [9] In 2021 OWASP classified it in position 2 as cryptographic failures [18], which is a broader category that encompasses many security vulnerabilities of the previous sensitive data exposure category. In the case of this project it may be better suited under position 4: insecure design, as the websites design is at fault to send sensitive information in GET requests.

The reports may also expose the browsing habits of specific users. As all reports are sent from the browser, a third party may collect reports coming from the same IP address to estimate the users interests by knowing which sites have they visited. This issue becomes more prevalent when there is a single company controlling many reporting endpoints for different sites. In that case the same user can be tracked across multiple websites.

Both of those information breaches are most threatening when a third party is responsible for maintaing the Policy Maker. The hosts already posses the information about the users browsing habits by serving their requests, and the query parameters are often included in the logs of the server. When the reporting endpoint is run by the host itself, the only additional information that may be gathered is the knowledge of the resources loaded in the client. This additional information is the main incentive behind deploying a Content Security Policy as it allows for finding resources that should not be loaded in the users browser.

During the development and testing of this project no data was collected from genuine users avoiding the issues described. When the Policy Maker is to be deployed on real hosts the ethical issues should be revisited and accounted for.

Chapter 3

The Automated Policy Maker

With very low usage of `content-security-policy-report-only` header this project aimed to create a fully automated policy creation and reporting tool. It would help increase the security of websites by alerting the developers early about any potential breaches and attacks that are deployed.

3.1 Ideology

Although the report only header does not limit any resources from being loaded, it can be used for monitoring purposes. When using the standard content-security-policy header any attack bypassing the policy will never be reported as enforcing and reporting go hand in hand, leading to untraceable attacks. The approach taken in this paper tries to make a report only policy that forces a report for each new source file. This results in a system where every attack leaves a trace and can be acted upon.

The Policy Maker takes advantage of the `report-only` part in Content-Security-Policy-Report-Only header, as developing a policy that is never intended to be in the enforcement mode allows to explore new behaviours. When using the report only header the policy does not need to contain all the loaded resources, that are used by the web application. With that in mind the Policy Maker may periodically remove certain sources, as it does not risk breaking the application. In this way the server may check whether a source is still in use, ensuring that the policy is always as tight as possible. When queried the server may also provide an up to date website map from the point of view of all the users.

Additionally, by using report only the policies do not need to be made future proof. Contrarily, it is beneficial for the Policy Maker server to be informed of all changes within the site. This allows for maximum surveillance of the site where no attack would go unnoticed. The server accomplishes that by using the using hashes for every inline script and exact url matches for standard script elements. If a script is added by an attack it would not match any hash or url signatures present in the report only policy. When user loads the compromised site a report will be generated allowing for quick response, minimizing the attack spread.

The requirement for a user to be compromised is unavoidable by the nature of the Content-Security-Policy-Report-Only deployment. As such the solution should be used as a defence in depth tool, which does not replace other measures necessary to secure the application.

3.2 Development

The project is mainly written in JavaScript and uses Node as runtime. This combination was used as it provides high level, powerful and very customizable tools required for efficient server creation. It also natively supports JSON data format, which is used for Content-Security-Policy reports. Many tools used within this project are available in the Node Package Manager allowing for better interoperability. Components that are not available under Node were connected by creating separate servers which communicated using GET and POST requests.

The high level view of the project structure is desplayed in figure 3.1. The project uses multiple interdependent modules, which expose limited amount of functions to support modularity. The key development within this project is on the diagram denoted as the Policy Maker. It is designed as a separate entity, which does not require the other components used within this project to function. When deployed in a real environment, it may be directly connected to the server serving HTTP requests, while providing an up to date policy for the observed reports. The Policy maker consists of 3 other smaller modules

Policy Engine Receives reports and generates policies

Oracle Takes script URLs from reports and aims and collects them from the internet

Evaluator Rates collected scripts and evaluates them

Other modules displayed in the structure were used to automate and test the Policy Maker.

Main Controller Initiates all other components, connects Policy Maker and proxy, and compiles statistics

MITMProxy Injects policies to simulate deployment alongside tested host and collects network metadata

Puppeteer Simulates the user for autonomous testing

Policy Engine

The server is the main component which exposes an open port dedicated to collecting reports. It is also the outermost layer of communication responsible for issuing warnings as they are encountered. This module exposes an `EventEmitter` interface,

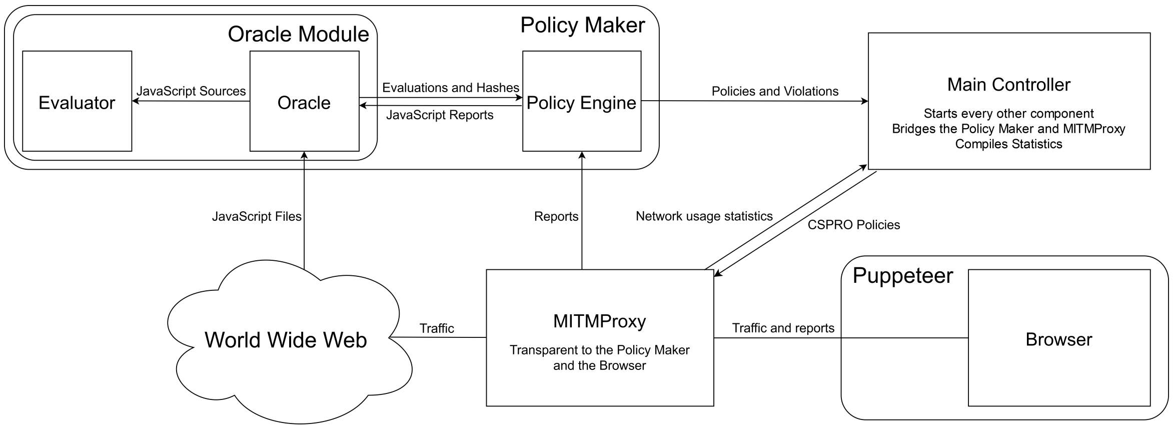


Figure 3.1: Structure of the project

which allows other components to run specific code when an event occurs. There are 5 possible events, 2 for security, 2 for logging and 1 for CSP updates:

`warning` Used to signify rarely used content which should be monitored. Practically it is used for iframes and small inline attribute scripts.

`violation` Used to signify a malicious script, use of `eval` or a long attribute script that does not fit in the 40 character report limit.

`new` Signifies that a new script has been spotted.

`changed` Signifies that a previously spotted script has been changed.

`cspro-change` Emitted whenever a policy is changed, notifying any module to update their CSPRO value.

When reports are received, they are parsed and the most important information is stored in the memory. Depending on the effective violated directive different actions are performed. Sources like images, fonts or styles are permanently added to the policy, as this project focuses on securing web applications from malicious scripts. IFrames trigger a warning event as they may be used for clickjacking attacks, after which they are added to the policy, as they are not the main focus of this paper.

The full contents of the reports are stored in an SQL database alongside the timestamp when they arrived. The server uses a PostgreSQL client and the reports stored were used for statistical analysis within the impact estimation section. The server also allows to be run on old reports stored in the database. This allowed to remove bugs during the development process, while reducing the strain of the project on the selected hosts.

The most logic within the server is dedicated to scripts. Attribute scripts always trigger a warning or a violation event. Then depending whether they fit into the 40 character limit within the `script-sample` field in the report, their hash may be added to the policy for a short duration to limit the amount of duplicate reports. Reports signifying use of `eval` function emit a violation event, after which they are ignored. Lastly all reports about scripts within HTML `<script>` tags are sent to the oracle to calculate their hash and evaluate their maliciousness. All hashes are stored and based on them the server recognises whether a script has changed, in which case it emits a `changed` event.

3.2.1 Oracle

The oracle is the inner module of the Policy Maker which is crucial for the functionality of the server. Its main goal is to aid the server in collecting scripts described in reports and to provide their evaluation. The oracle is a separate module to allow for a redesign without affecting the core logic of the server. In this project scripts are fetched from the hosts using GET requests, but this mechanism could be adjusted to instead fetch specific scripts from the inner code repository.

3.2.2 Evaluator

The evaluator in the Oracle Module is used to evaluate scripts from the source code that was collected by the oracle. Compared to the oracle it is not necessary for the function of the Policy Maker, but without one only new and changed events will be emitted. The evaluator included in the source code uses a machine learning model created by Maria Zorkaltseva intended to find obfuscated malicious JavaScript files [21]. As the model is written in Python, downloaded scripts are passed to a python http server using POST requests. The server calculates the maliciousness score and returns the result in the response.

3.2.3 Main Controller

Main Controller is the entry module of the project. Every other module described requires some form of a context to run properly. The main controller sets up each module with a context of a specific test scenario. It collects all events emitted from the server and logs them to the console for a clear view of the servers operation. Whenever a new policy is generated, the main controller passes it from the server to the mitmproxy so it can be injected into server responses. Lastly the main controller logs statistical data collected by mitmproxy.

3.2.4 MITMProxy

MITMProxy is an interactive https proxy which can be used to intercept, read and modify traffic coming through it. It supports scripting, which is used to automatically insert `content-security-policy-report-only` header before it is sent to the browser. In this way, the browser sees the modified response as if it was sent directly from the host. This behaviour directly emulates the scenario where the policy maker were to be deployed alongside the hosting server. As all traffic is directed through mitmproxy, it has also been used to collect statistics required to evaluate the project.

In the code this integration works by creating a separate server to which the mitmproxy module connects periodically. Whenever this happens all data collected is sent away and a current CSPRO generated by the reporting server is returned. Due to this design there is slight latency between generation of a new policy and deployment to the user. Although this delay exists, it is insignificant and in real world many hosts use proxies to reduce the load on servers or clients use caching to reduce their network usage. In both of those cases the delay between deployment of a new CSPRO and its perceivable effects may be much larger than what is observed in this project.

3.2.5 Puppeteer

The second tool used that does not benefit the security of the server, but was extensively used during testing was Puppeteer. Puppeteer is a Node.js module designed to simulate user interaction in a chromium client. It uses a programmable interface, which results in high precision and reproducibility between executions. For those reasons Puppeteer was used for all impact estimation tests. In the code a separate module was dedicated to Puppeteer which starts it up connected to the proxy with parameters simulating a regular user. After that it is used to crawl the currently tested host.

3.3 Initial Content-Security-Policy

Before the server can be started there are multiple choices that need to be made relating to the initial CSPRO header. Those decision dictate which parts of the site are protected and they heavily impact both the security and the performance of the project.

3.3.1 unsafe-inline

`unsafe-inline` allows all inline scripts to execute. This source is disincentivized by Mozilla Docs [1] and CSP Quick Reference Guide [2]. It is also shown to be insecure in multiple papers [20] [17].

This is largely due to the fact that it allows for dynamic execution of scripts, by inserting them directly into the website code. It also prevents any reports to be generated in case of a DOM injection attack.

This directive may be considered as the oracle uses a best effort approach to extract inline scripts. This leads to many scripts being unable to be extracted without executing the loader script. Executing unknown scripts, even in an isolated environment, is very unsafe and consequently the server may never be able to know the source code of all possible scripts. It means that the browser will always send a report to the server whenever a script is loaded in this way, heavily impacting the performance.

In the end, even though there are practical implications to including `unsafe-inline` source into the CSP, it has too severe negative security implications. With `unsafe-inline` in the source list the CSP becomes very easily bypassable and provides little to no benefit against attacks.

3.3.2 self

Adding `'self'` to the source list of scripts would allow for all scripts coming from the host to be executed without sending the report. It still sends a report for every inline script and it may be a good option for a host that employs many security measures towards its own code. When used it reduces the amount of reports and reduces the average size of the report as none of sites own scripts will need a source value to execute. This method successfully changes the threat surface from all scripts used on a web application to monitoring changes only in external and inline scripts.

In most of the experiments `'self'` source is not used as the project aims to develop a policy maker that protects from any script based threats. As this source improves the performance the most while compromising the security the least, some tests were done to compare the results.

3.3.3 strict-dynamic

`strict-dynamic` is a successor to `unsafe-inline`, where it will allow dynamically loaded scripts as long as they are loaded from a script that is allowed by a nonce of a hash. It also changes the behaviour of the policy where it instructs the browser to ignore `unsafe-inline`, `'self'`, `unsafe-eval` or many other wildcard statements.

By using `strict-dynamic` the server can prevent the dynamically loaded scripts from generating reports. Unfortunately it also prevents non-inline dynamically added scripts from generating a report, which the server would be able to retrieve. It also prevents the browser from honouring URL based allowlists which fully undermines the policies generated by the server. Due to those issues `strict-dynamic` source is not used.

3.3.4 unsafe-hashes

Adding `unsafe-hashes` to the source list does allow hashes to be used for inline script attributes such as `onClick` or `onLoad`. It uses the prefix *unsafe*, but hashes in `script-src-attr` directive by default are ignored as they will never have a valid target. As such the server uses `unsafe-hashes` for this specific directive. When an attribute script fully fits in the 40 character limit of the script sample field in the report, the hash is added to the policy to prevent further reports. In either case, whether it fits or not, an event is generated to allow for the script to be removed, for example by being moved to a separate file.

With all those considerations in mind the initial CSP for the server is displayed in figure 3.2. It allows for capture of all necessary reports for scripts running on the host. By using `default-src 'none'` the CSP will also capture non script resources hosted on the website. Styles compared to other sources can be used inlined and as such `unsafe-inline` is used in their directives to prevent reports which are not easily preventable otherwise. This work focuses on preventing from malicious JavaScript files, but as CSS attacks were described earlier, similar approach can be taken to secure the web app from malicious style sheets.

```
default-src ’none’ ;
script-src ’report-sample’ ;
script-src-elem ’report-sample’ ;
script-src-attr ’report-sample’ ’unsafe-hashes’ ;
style-src-elem ’unsafe-inline’ ’self’ ;
style-src-attr ’unsafe-inline’ ;
report-uri <URL>
```

Figure 3.2: Initial policy for the Policy Maker

Chapter 4

Evaluation

The Automated Policy Maker correctness is evaluated by using a set of empirical tests and the potential impact is tested in a simulation. Empirical tests attempt to show that the program is working correctly and achieves the goals described in the previous section. This work also aims to estimate the impact of deploying the solution in the real world. This set of tests compares the network impact of the Policy Maker to the loads that the hosts are already experiencing during their normal operation.

4.1 Empirical tests

This project consists of multiple modules which all work together to achieve a bigger goal.

Oracle is a module that is used to generate hashes of loaded resources and communicate to the python evaluation server. When tested on a local server, it can successfully fetch files from it and generate hashes equal to the ones that are generated with system tools on file data. After receiving line and file information it is sometimes able to extract data from inline scripts embedded in the HTML of the page. Occasionally, due to formatting, encoding or other issues related to dynamic inclusion of data into the page, it may return no or a wrong hash. In that case a violation event is emitted to warn about suspicious behaviour. Due to security concerns the oracle does not dynamically execute scripts or change the HTML code, which does happen within the browser. Most notably a script may change the inner html of a page to one with another script. In this case the oracle is unable to generate the script that was included as it is not retrieved in HTML response. Throughout testing of this project, different scripts responsible for advertisements and user tagging were found to use this technique. The influence that those scripts have on the effectiveness of this project is clearly seen when the impact of the policing server is estimated.

The second role of the oracle is to pass the script extracted to the evaluation server. When a script is passed to the evaluator a simple binary response is returned representing whether the script was found to be malicious. As this work focuses mainly on developing the policy maker for CSPRO header and the evaluator is a machine learning model described in a different paper, its functioning is not thoroughly tested. While its results may be occasionally inaccurate, it still clearly demonstrates the inner workings of the Policy Maker server.

The server which is the main outcome of this project uses the oracle extensively. Each script that is reported is sent through the oracle, then depending on the evaluation they are added to the policy for a specific amount of time. Eventually after enough time has passed the server successfully removes the entry from its policy. When a new report for the same resource arrives it compares the new evaluation from the oracle to the previous one and does emit events related to the result. Each time the CSPRO value is changed it is communicated out of the server and subsequently embedded into HTML responses by mitmproxy.

During testing this result is easily observable in the browser connected to the proxy. When loading the host page on a newly created server, multiple reports are generated back to the server. On the server multiple new events can be seen and in a case where there is a script deemed malicious a violation is observed too. Subsequent refreshes of the page in the browser no longer produce reports until the scripts present are removed from the policy. Even if there are more reports arriving the server will not generate new events unless a scripts has changed.

The server also keeps note of other resources loaded on the page with a lower level of detail. It stores only hosts of those resources, which are added to the policy of the server and they are not removed over time. When a new source is added only the update of the CSPRO is triggered, but no additional event is emitted.

The only exception to this rule the `frame-src` directive. When a report about a violation to the frame-src is received a new warning is emitted. As previously described iframes can be used in clickjacking attacks and due to lower amount of iframes used on pages it is important to verify that no malicious iframes are running.

After testing the server on a model website a Content-Security-Policy-Report-Only header has been generated that encapsulates all the described behaviours. It includes 2 different hashes for 2 different types of inline scripts and one URL for 1 script on the site. Additionally it has 1 frames source, as the website contains an iframe.

```
default-src 'none' ;
script-src 'report-sample' ;
script-src-elem 'report-sample'
https://testing.site
'sha256-f2KDabOBheatnkXlUGlWthbaCaKmlDSpV5682oXCFAw=' ;
script-src-attr 'report-sample' 'unsafe-hashes'
'sha256-ChDYROnXdAFVmpHR/3aN8o1C7WPipQjRBCtCfYIS8CM=' ;
style-src-elem 'unsafe-inline' 'self' ;
style-src-attr 'unsafe-inline' ;
frame-src testing.site ;
```

```
report-uri <URL>
```

And the server had emitted the following events.

```
cspro-change
warning:     frame-src    https://testing.site/frame.html
cspro-change
new:         https://testing.site/index.js
cspro-change
cspro-change
new:         https://testing.site/index.js
cspro-change
warning:     script-src-attr    inline
cspro-change
cspro-change
cspro-change
changed:     https://testing.site/index.js
cspro-change
violation:   script-src-elem    https://testing.site/index.js
        Hash: 'sha256-FTmBzDS1vGvEUt2gfXA1NhjGNt4vZ1mrB7AMyyQzr5U='
cspro-change
violation:   script-src-elem    https://testing.site/index.js
        Hash: 'sha256-FTmBzDS1vGvEUt2gfXA1NhjGNt4vZ1mrB7AMyyQzr5U='
cspro-change
cspro-change
```

Between many `cspro-change` events it may be seen that at some point a newly detected script changed and became malicious.

The Policy Maker is also able to detect client side scripts. On the second page a link is embedded which results in reflected script execution.

```
/index2.html#value=<script>document.write('XSS injection')</script>
```

When clicked only the request of `/index2.html` is seen on the server, but the payload produces no trace in the logs. The Policy Maker on the other hand receives the report from the user and after it is unsuccessful in retrieving the script, produces a violation with an undefined hash:

```
violation:   script-src-elem    inline    Hash: undefined
```

The report, which is passed with a violation event, includes the file uri and line and column numbers of the specific function that resulted in the violation. In this way the server can notify the maintainers of the server and reduce the response time for incidents.

4.2 Impact estimation

To try to provide most accurate results the Policy maker was deployed on a mixture of randomly selected and hand picked hosts from top 10,000 hosts.

In this paper results are presented from 4 different websites which best demonstrate both good results as well as issues that may be encountered when trying to implement the server into the host. None of the sites which were used to test the policy maker have deployed their own CSP or CSPRO headers. This was done to avoid the policy overwritten in the proxy to be further modified by the meta tags included in the page source. Choosing such sites would also skew the results as those websites would already have been adjusted to work with CSP.

During the tests the server is deployed separately on all the described hosts. In each of the tests a set of 171 sub pages are opened over a time span of about 45 minutes, as depending on the time to load a page the final time total may vary. The first page opened is always the main page for the host, after which a random page is chosen out of all links that were seen throughout the duration of the test. For the tests the server will remove a JavaScript sources after 10 minutes, so that a single script may be added to the policy and better simulate the real life situation.

As this tests are done on real hosts it is impossible to assume maliciousness of any of the scripts returned as malicious. From all of the reported scripts that were analysed by hand, the most common issues for those reports were dubious programming practices or unnecessary obfuscation of seemingly benign scripts.

The Policy Maker can create additional traffic between server and client in two ways, by adding the policy to the response header and by users generating reports. It also generates server to server traffic when retrieving scripts for hash calculation and analysis. As most of the scripts retrieved from by the oracle are inlined or hosted on the host, the impact of this communication can be greatly reduced by having servers virtually close together. It is also not on a critical path to the client, where additional traffic may hinder user experience. For those two reasons this traffic is not accounted for in this impact estimation.

During the tests mitmproxy plugin collected network metadata to evaluate the impact of the server. It summed all the traffic coming from the host domain and all its subdomains before anything was added to the response. Separately it summed up all occurrences where the server would be adding bytes to the responses and all the lengths of all reports coming through the proxy. As all reports are stored in an SQL database, the length of all reports can be calculated with and SQL query.

4.2.1 www.libertatea.ro

`www.libertatea.ro` is a Romanian news website which yields the worst results out of all tested hosts. The site is dynamically loading many scripts, making it impossible to retrieve the executed scripts. With this behaviour the oracle is unable to rate the scripts or assign hashes to them. This results in duplicate reports which drag down the performance.

Due to those dynamically loaded scripts, the site transfers 3.5 times more data in reports back to the server than what was originally sent from the host.

This behaviour could be partially prevented by using `'strict-dynamic'` source in scripts directive, which would result in all loaded scripts to be allowed to run, as long as the loader is allowed by the hash. This solution would reduce the amount of reports sent, but it would also result in drastic reduction in security as the server no longer able to properly monitor the site. In such case, if one of the loader sites became compromised and started receiving malicious scripts to load, the server would not be able to detect such change.

4.2.2 quran.com

`quran.com` is a digital provider of Quran. This website is an example of a host that would be perfect to deploy the Policy Maker on. The application uses very few JavaScript files, it does not use any inline scripts or externally hosted sources. Due to that the policy is small and very efficient.

4.2.3 www.professormesser.com

Professor Messer is a group dedicated to providing teaching courses and resources related to technical certificates. This website shows good results in short term deployment, but due to dynamically loaded scripts performs poorly during longer tests. It dynamically changes URLs of loaded script files, which pollute the policy and reduce performance.

4.2.4 www.caixabank.es

As most of randomly chosen hosts provide no good results `www.caixabank.com` was chosen as an example of a site that may already be interested in security and avoids the pitfalls of the previous hosts. It is the only hand picked site that this project was ever tested on. It has much better results, but still shows that deploying such a server will take non insignificant amount of resources. Towards the end of the experiment an extensive policy was created, yet the site still generated reports due to the use of `eval` function in their code.

Eval is a notoriously unsafe function used very often in malicious scripts. It allows for any string of characters to be executed as JavaScript code. In Content-Security-Policy this function is separately recognised with its own source directive for scripts: `unsafe-eval`. The reports also use a unique value of `eval` in their `blocked-uri` field. The only way to stop those reports without changing the pages source is to add `unsafe-eval` to the script sources. Unfortunately in that way all malicious uses of the `eval` function will also be omitted. Due to that `unsafe-eval` is never added to the policy and through countless violations emitted by the server, the developers may be inclined to stop their bad practices and refactor the code.

4.2.5 Test results

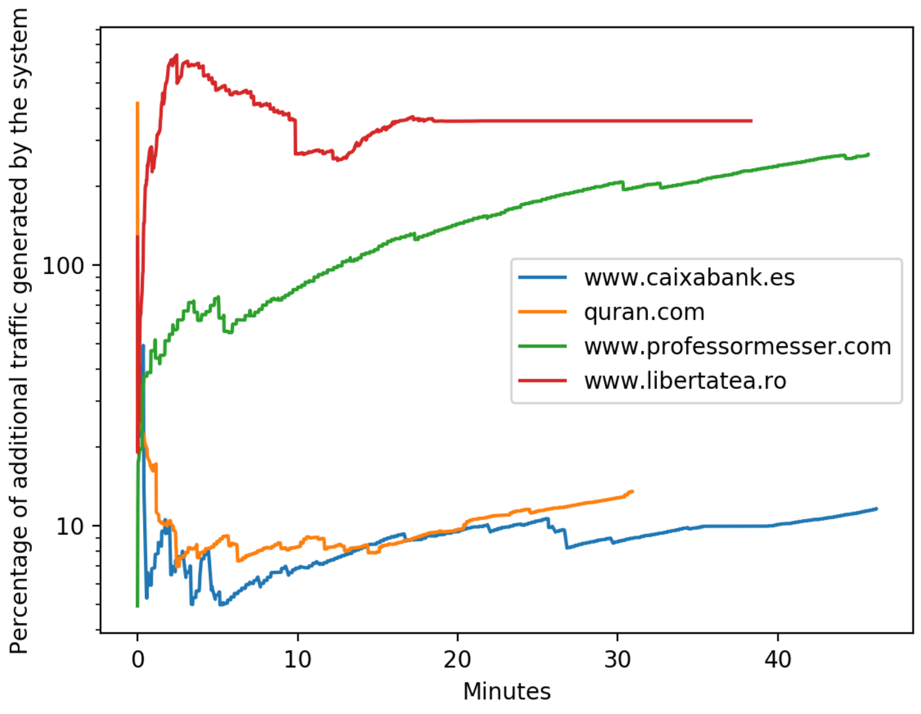
The results of the impact estimation are mixed. Some websites show very high costs of running the Policy Maker, while other show more optimistic results. Table 4.1 displays the final data usage statistics for tested hosts. Percentage added signifies the amount of additional network traffic that would be generated by using the Policy Maker when compared to the already existing traffic. The figure 4.1 shows how this percentage changed over the duration of the experiment. It increased when there more reports than content sent to the browser and decreased otherwise.

Quran.com and caixabank.es see a 12 to 14 percent increase in traffic, which may be an acceptable compromise between cost and security. The figure is reinforced by the article by Deloitte [6], which details that companies, which they questioned spend an average of 10.9% of their IT budget on cybersecurity. This traffic also happens mostly after the website is already loaded, which will not compromise the user experience by increasing the website's load time. When deployed on a compliant website the results may be even better as Caixabank generates unavoidable reports by using the eval function.

Host	Total Host Traffic	Additial Traffic	Percentage Added
www.caixabank.es	339.4 MB	39.4 MB	12%
quran.com	16.4 MB	2.2 MB	14%
www.professormesser.com	3.4 MB	9.1 MB	266%
www.libertatea.ro	8.5 MB	30.2 MB	357%

Table 4.1: Final network usage statistics for the host and the Policy Maker

The other two hosts show that in a long term the Policy Maker is a bad tool to improve the sites security. Even though professormesser.com had good results in the beginning of the experiment, as shown in the figure 4.1. As the policy size increased the additional network usage did not settle, but continued to grow. In both sites, professormesser.com and libertatea.ro, the reports generate much more traffic than what was needed to load the site. Both of those hosts use dynamically loaded scripts which are impossible to be fetched by the oracle, resulting in many duplicate reports. For the Policy Maker to be applicable, the sites would need significant restructuring.



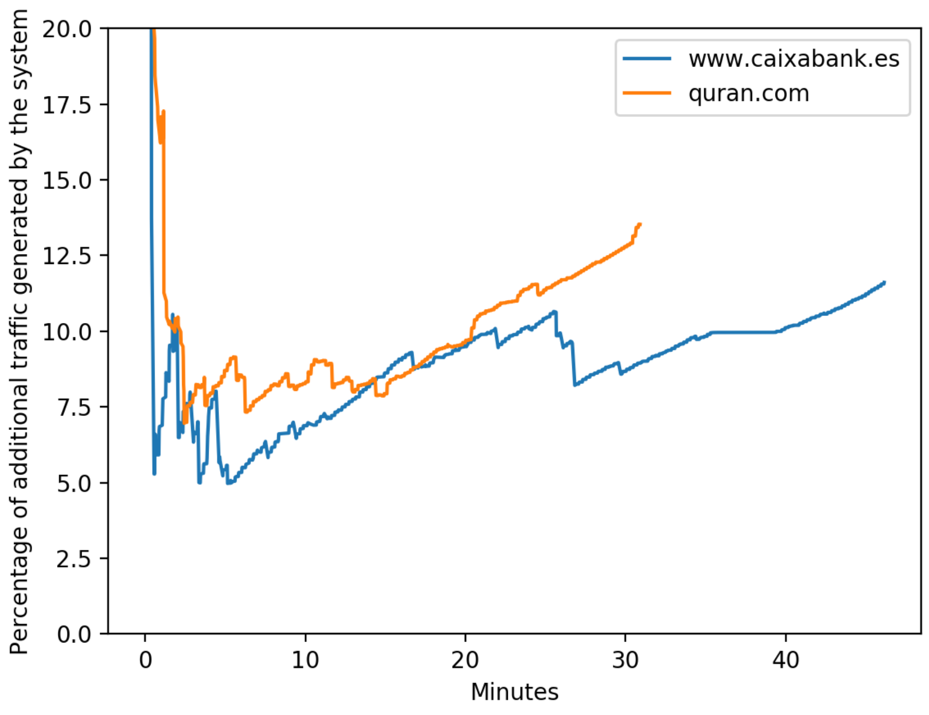


Figure 4.1: Percentage of bandwidth dedicated to additional traffic generated by the Policy Maker using default settings

Table 4.2 shows report and policy related statistics. Percentage traffic added to header signifies the unavoidable increase in data transferred which is on the critical path to the user. As the policies, sizes of which are also displayed in this table, need to be sent in the header of the response. This means that they will be received by the user before any other content, increasing the time before the site is responsive.

Host	Biggest Policy	Percentage traffic added to header
www.caixabank.es	15.1 KB	2.7 %
quran.com	3.4 KB	4.5 %
www.professormesser.com	5.7 KB	36.1 %
www.libertatea.ro	41.4 KB	41.6 %

Table 4.2: Statistics of traffic on critical path to the user

The Policy Maker was also tested with the host added to the CSP from the start. This scenario ignores scripts loaded from the host, but still checks all inline and external scripts. When adding this source, the site is no longer protected from insider threats or reflected attack where a malicious script is saved to the host. The server still protects from many common DOM injections and from depended upon hosts turning malicious. The results of this test are shown in table 4.3. As shown the results are much better where additional traffic to Caixabank has reduced from 12% to 8%, making the Policy Maker even more viable. Quran.com uses no sources outside of itself and in this example the 4% is the minimal load increase that is required for the system to function.

Host	Total Host Traffic	Additial Traffic	Percentage Added
www.caixabank.es	378.6 MB	31.9 MB	8%
quran.com	18.8 MB	0.7 MB	4%
www.professormesser.com	3.1 MB	8.0 MB	261%
www.libertatea.ro	8.0 MB	30.4 MB	382%

Table 4.3: Final network usage statistics when scripts from the host are allowed

The costs detailed in both tables should be carefully examined by the potential users of the Policy Maker. The automated tool may provide a significant boost to security by reducing the time to detect any attacks. It also has not insignificant monetary and experience costs associated to running the server.

4.3 Things to improve

The approach shown in this paper proves to be a new and promising way to improve the security of particular websites. Main applicability issues stem from the quickly expanding and vastly untamed environment of front end development. Websites may quickly grow to sizes that prove to be unmanageable by the Policy Maker.

4.3.1 Content-Security-Policy Specification Changes

During the development of the Policy Maker, it had to comply with the Content-Security-Policy specification. This results in issues and small insecurities, which could be solved if the specification was more robust.

Variable reports

Things that could dramatically improve the performance of the Policy Maker would require changing the CSP standard. Allowing for more control over the report template could reduce the data transferred back to the CSP reporting endpoint by about 90%. This comes from each report containing the current policy used when loading the page which is responsible for most of the traffic. For each host this percentage is displayed in table 4.4. With significant reduction in report sizes, the project would operate much lower overheads. This idea could be implemented by creating a new directive `report-fields` where each source of type `’no-original-policy’` or `’no-violated-directive’` would prevent the specific field from being included in the report. This solution will work with CSP idea of backwards compatibility as older browser will ignore this directive and send full reports.

Host	Bytes used by Original Policy in Reports
www.caixabank.es	95 %
quran.com	85 %
www.professormesser.com	90 %
www.libertatea.ro	95 %

Table 4.4: How much of each report is the current policy

Script hash in reports

Another useful addition to the CSP reports specification would be the possibility to ask the browser to pass the hash of the loaded scripts. It would allow the oracle to check whether the retrieved script is identical to the one that the user has loaded. Currently, even if the oracle successfully retrieves a scripts there is no such guarantee and the script may have changed in between. The loss of privacy incurred by adding hash is either very negligible or non existent as reports are already passing full URL of blocked resources and sites on which said violation happened. This feature could be implemented in a similar way to `report-sample` source, where `report-hash` would send the hash of the script as received by the browser.

Looser integration checks

The project initially intended to use hashes for each type of script. This would potentially allow for scripts to be checked before timeout by receiving a significant amount of reports when it no longer has the previously calculated hash. Unfortunately, the CSP standard required integration attribute for `<script>` elements which are loaded from a file [3]. The integration attribute needs to contain the expected hash, which also needs to match a hash in the CSP for the script to be loaded. Adding such attribute nullifies the idea of an automatic tool, that could work independently from the rest of the codebase. Due to this reason the server uses a mixture of URL and hash based allowlists, lowering the security slightly.

To make the described approach work, the specification would need to make a way to ignore the integration attribute and check the hashes after the script is loaded. This would not change the behaviour of the browser when using the report only policy as the non enforcing nature allows all scripts to be loaded. When used with the enforcing policy it would allow for data to be sent out of and received by the browser, before potentially blocking the script. As CSP is not particularly good at preventing exfiltration attacks [4], such addition would allow to balance security and usability of CSPs.

Unfortunately, this specification change would break the backwards compatibility the most out of all proposed changes within this paper. When used in enforcing mode, older browsers which are not updated to use this feature, would block otherwise allowed sources from executing. In report only mode, it would result in many bogus reports, which would either be ignored or significantly strain the system. Although this change would severely increase the security of the Policy Maker, it is unlikely to be added, due to issues described above.

4.3.2 Other improvements

Real life deployment

Within the impact estimation, the server is tested on existing code bases of real hosts. This is the closest this set of tests can go, without being actually integrated into the server itself. The next logical step for the Policy Maker is to be tested alongside a willing host. This would allow to calculate the real impact of the server by receiving reports from the website's users. It would also show any issues with the Policy Maker logic, when reports received are old, invalid or from a varying set of browsers.

Evaluator Improvements

The evaluator in this project is a rather simple machine learning model which functions great for testing and development purposes. It could be expanded by using more data when deciding the maliciousness of the script. It could include host names and IP addresses and compare those against a set of known compromised endpoints. It could also benefit from more training data or from more elaborate linguistic models. The possibilities to improve the evaluator are many and they should be seriously considered when trying to run the Policy Maker on a real host.

Securing other Frames and Styles

This work focuses on protecting against JavaScript based attacks, while giving some insight into iframes that are embedded within the site. Protecting against attacks using either of those elements is much harder because of practical reasons. As sites are moving away from using inline attribute scripts, styles can still be commonly found in attributes of many HTML elements. Attacks using styles are also much harder to detect as they often retrieve single characters at a time. This would result in big policies with potentially little benefit to them.

IFrames on the other hand require more context to be considered malicious. A payment IFrame is to be expected on a shopping site, but the same iframe may be considered malicious when injected into the landing page of a IT solutions provider, which does not work with individual clients. For this reason the server implemented in this work displays the iframes encountered as warnings, as each frame should be reviewed manually.

Chapter 5

Conclusion

Cross Site Scripting is a very old and still relatively common vulnerability that affects many sites. It has very serious consequences to the privacy of users and the reputation of web hosts. Among many tools that help detect vulnerable pieces of code there are Content Security Policy headers which may protect and even prevent the attacks before they even happen. Unfortunately, due to high barrier of entry and the consistently increasing size of websites, most of the hosts use those headers incorrectly. This leads to policies that are easily bypassable and as a result do not give any protection to the users.

This paper presents a new way of using the Content-Security-Policy-Report-Only header to monitor the site for JavaScript based attacks. It avoid pitfalls related to deploying policies in enforcing mode by abusing the liberties given to it by running exclusively in reporting mode. It can create the strictest possible policy, which would be impractical otherwise, to cast an unbypassable net for each new and possibly malicious script to fall into. In this way the newly created server can report on any breaches very quickly or even instantly as they happen.

The work is also evaluated in tests that simulate real world deployment as closely as possible. The results of said evaluation may prove to be satisfactory for the Policy Maker to be deployed on real hosts which are already putting in significant effort to secure their applications. As the server is reliant on the Content Security Policy Standard, its performance may significantly increase as new features are added.

We hope that the ideas presented in this paper inspire others to create or improve other policy creating servers. This report shows that there is still work to be done in the Content Security Policy domain to improve the security of users on the web.

Bibliography

[1] 2023. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/script-src (visited on 08/31/2023).

[2] 2023. URL: https://content-security-policy.com/unsafe-inline/ (visited on 08/31/2023).

[3] 2023. URL: https://www.w3.org/TR/CSP3/#external-hash (visited on 09/09/2023).

[4] Steven Acker, Daniel Hausknecht, and Andrei Sabelfeld. "Data Exfiltration in the Face of CSP". In: May 2016, pp. 853–864. DOI: 10.1145/2897845.2897899.

[5] Devdatta Akhawe. *[CSP] On Reporting and Filtering*. 2015. URL: https://dropbox.tech/security/on-csp-reporting-and-filtering (visited on 06/06/2023).

[6] Julie Bernard, Deborah Golden, and Mark Nicholson. *Reshaping the cybersecurity landscape*. 2020. URL: https://www2.deloitte.com/us/en/insights/industry/financial-services/cybersecurity-maturity-financial-institutions-cyber-risk.html (visited on 07/24/2020).

[7] World Wide Web Consortium. *Content Security Policy Level 3*. 2023. URL: https://www.w3.org/TR/CSP3/ (visited on 06/06/2023).

[8] *CSP: require-trusted-types-for*. 2023. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/require-trusted-types-for (visited on 06/06/2023).

[9] The OWASP Foundation. *OWASP Top 10 Application Security Risks - 2017*. 2017. URL: https://owasp.org/www-project-top-ten/2017/Top_10 (visited on 09/06/2023).

[10] OWASP Fundation. *DOM Based XSS*. 2023. URL: https://owasp.org/www-community/attacks/DOM_Based_XSS (visited on 09/10/2023).

[11] Mike Gualtieri. *Stealing Data With CSS: Attack and Defense*. 2018. URL: https://www.mike-gualtieri.com/posts/stealing-data-with-css-attack-and-defense (visited on 06/06/2023).

[12] Brij B Gupta, Shashank Gupta, and Sumit Gangwar. “Cross-Site Scripting (XSS) Abuse and Defense: Exploitation on Several Testing Bed Environments and Its Defense”. In: *Journal of Information Privacy and Security* 11 (July 2015), pp. 118–136. DOI: `10.1080/15536548.2015.1044865`.

[13] Jastra$_ilic$. *Cross-Site Scripting (XSS) Makes Nearly 40% of All Cyber Attacks in 2019*. 2020. URL: `https://www.precisesecurity.com/articles/cross-site-scripting-xss-makes-nearly-40-of-all-cyber-attacks-in-2019/` (visited on 02/19/2020).

[14] XSS Jigsaw. *CSP 2015*. 2015. URL: `https://blog.innerht.ml/csp-2015/` (visited on 06/06/2023).

[15] Krzysztof Kotowicz. *Prevent DOM-based cross-site scripting vulnerabilities with Trusted Types*. 2023. URL: `https://web.dev/trusted-types/` (visited on 06/06/2023).

[16] Inc OWASP Foundation. *Testing for CSS Injection*. 2023. URL: `https://owasp.org/www-project-web-security-testing-guide/stable/4-Web_Application_Security_Testing/11-Client-side_Testing/05-Testing_for_CSS_Injection` (visited on 06/06/2023).

[17] Sebastian Roth et al. “Complex Security Policy? A Longitudinal Analysis of Deployed Content Security Policies”. In: *Proceedings of the 27th Network and Distributed System Security Symposium (NDSS)* (). DOI: `10.14722/ndss.2020.23046`. URL: `https://par.nsf.gov/biblio/10173479`.

[18] OWASP Top 10 team. *OWASP Top 10:2021*. 2021. URL: `https://owasp.org/Top10/` (visited on 09/05/2023).

[19] *The HackerOne Top 10 Most Impactful and Rewarded Vulnerability Types – 2020 Edition*. 2020. URL: `https://www.hackerone.com/top-ten-vulnerabilities` (visited on 09/10/2023).

[20] Lukas Weichselbaum et al. “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*. 2016, pp. 1376–1387.

[21] Maria Zorkaltseva. *Detect Malicious JavaScript Code Using Machine Learning*. 2022. URL: `https://pub.towardsai.net/detect-malicious-javascript-code-using-machine-learning-7709f9cdb7e7` (visited on 09/05/2023).