UNIVERSITÄT DES SAARLANDES

Bachelor-Thesis

In order to obtain the academic degree

Bachelor of Science (B. Sc.)

at Saarland University

in the field of computer science

Analysis of Content Security Policy in different browsers settings, including Desktop and Mobile browsers.

By Ahmad Hajy Omar

Supervised by

Dr. Cristian-Alexandru Staicu

Dr. Dolière Francis Somé

Saarbruecken, 31. 5 .2023

Declaration of Independence

I assure that I have independently authored the present work and have not used any sources or aids other than those specified.

Furthermore, I hereby declare that the submitted work has not been previously submitted by me or any other person at this or any other university.

Moreover, I am aware that the falsity of this declaration may result in grading the work as insufficient and may lead to exclusion from further examination achievements.

Saarbruecken, 31. 5 .2023

Ahmad Hajy Omar

Contents

1 Introduction

Human society lives in an era that depends on technological development in all fields, and the most important of these areas is the use of the Internet, one of the most significant technological innovations whose impact was evident in changing the world and humanity. There are many areas in which the Internet is used, for example, social communication, education, entertainment, and other needs, so using the Internet to browse is one of the most common and essential things.
Browsing the World Wide Web is done by using popular browsers such as Chrome, Safari, and Firefox, by using mobile devices with various systems, such as Android and iOS, or using a browser on a computer, but still browsing through the Internet poses many challenges and risks, the most important of which is adequate protection Which provides safe browsing and protection against the usual attacks.
As a result, the founders of browsers seek to improve the security of the browser to avoid many attacks that occur through many factors, such as visiting unsafe sites, one of the most critical factors that help maintain the safety and security of browsing on the Internet is the security header Content Security Policy (CSP), that is included in the HTTP response and meta tags because CSP mainly included in the HTTP response, our work focuses only on the CSP in the response headers, furthermore, in this work, other security headers will be considered, such as Strict-Transport-Security, X-Frame-Options, and Set-Cookie that help to mitigate certain types of attacks, such as clickjacking.
Starting from the fact that the security policies of many websites can be inconsistent for the users visiting the same website according to many factors, such as different User Agents, different viewports, different accepted languages, and other factors cause different HTTP responses.
Many previous studies have analyzed and observed this problem and even tested it for the top websites (only home pages); in our work,

Goals 1: We will continue this analysis and go deeper into the sub-pages of these top homepages; for example, if the homepage of a website is secure against a specific type of attack, but a sub-page is not, then many risks of this attack to happen can be posed to the main page and the other sub-pages, as a simplified explanation of our daily life, going back to the days of the spread of the Coronavirus, if one of the family members in the house is infected, then the rest of the family members would be exposed to infection.

Goals 2: Back to the critical security header Content Security Policy (CSP), CSP acts like a traffic police officer who takes the responsibility to regulate the traffic flow and enforce the traffic laws; CSP is here to instruct the browser how to handle and enforce the execution of sources through the particular type of directives of the form type-src like script-src and values (Directive: Value), one of the most potent values to ensure the safety of the CSP against certain types of attacks is the Nonce-Value, it is a value that will be randomly generated for each request, and it is the key to allowing the execution of inline scripts, but the question what if there is a chance of nonce duplication even though nonce values are designed to be generated without facing the duplication issue for two different requests? This work will test if there is

a chance to get the same nonce value after visiting the website twice and observing when this problem happens and what factors can cause this problem.
Furthermore, the nonce values of each homepage and its sub-pages will be compared to find out if the nonce value can be duplicated for a homepage and one of its sub-pages.

Goals 3: Another question is discussed in this work: How would the websites react if the client sent a malformed User Agent or a User Agent that does not exist? Moreover, we want to observe the reaction of these websites when using unusual accepted language, such as malformed accepted language.

2 Background

2.1 Content-Security-Policy

Content Security Policy (CSP) is a security mechanism that aims to mitigate certain types of attacks such as cross-site scripting (XSS), CSP is an HTTP response header, which controls the allowed sources such as scripts and other resources; Furthermore, CSP gives the developers the ability to control and determine the set of trusted domain sources [13]. CSP consists of directives separated by semicolons. Each directive contains values; CSP has been designed to help the browser to know how to handle and execute the scripts and to define which resources can be loaded onto the website through the rules that have been defined by the values of CSP's directives (Some CSP's directives in Table 2.1) , most of these directives have the form of `type-src`, such as `script-src` , `default-src` , `style-src` , and `object-src` ; these directives are followed by values that determine the allowed sources from which the content can be loaded. Fetch directives are the essential directives of the Content Security Policy, such as `script-src` and `default-src`, these are the directives that we will pay attention to specify whether the CSP is secure or not; Furthermore, fetch directives fall back if it is not present, e.g., if `default-src` is present in a CSP policy and `script-src` is missing, then `script-src` falls back to `default-src` [12]. Web pages deploy different CSPs, and these differences lead to weaknesses against CSP violations, especially in the directive `script-src` that provides the required protection against XXS attacks, the reason behind these vulnerabilities is allowing insecure domains that affect the security of the other pages within the exact origin and make them vulnerable to security attacks. According to previous research, 72% of the web pages that deploy a CSP are weak against CSP violations [13].

2.1.1 Important directives

Table 2.1: CSP directives

directive	Defined resources
`script-src-elem`	inline scripts
`script-src`	scripst
`default-src`	all resources (fallback of fetch directives)
`frame-ancestors`	resources that can embed the page[5]
`object-src`	used resources in <object>, <embed> and <applet>
`font-src`	resources to load fonts
`img-src`	resources for loading images
`style-src`	resources for loading CSS stylesheets
`media-src`	resources for loading media content (video and audio)

2.1.2 Directive's values

Table 2.2: List of Possible CSP Directives and Values

Value	Description
'none'	No source is allowed.
'self'	Allow sources from the same origin.
'unsafe-inline'	Inline scripts are allowed.
'unsafe-eval'	Allow the execution of code contained in a string.
'nonce-value'	The nonce terms stands for **number used once**, it is a value is randomly generated for each request and allows the execution or loading of inline scripts and styles with a specific nonce value e.g., `nonce-We11Trust54YOu`.
'hash-value'	The hash value is generated by a cryptographic hash function (e.g., sha-256). Only inline scripts ans styles with a computed hash value matching the hash value in the CSP header are allowed to execute.
'strict-dynamic'	Allow the usage of inline event handlers and inline scripts.
Host-source	Allow sources from a specific URL (e.g., `https://allow.com`).
Wildcards '*'	Allow sources from all domains or subdomains.

Examples of nonces and hashes, usage in CSP:

1. *nonce-value*: A website deploys a CSP policy that contains the following directive:

   Listing 2.1: Content Security Policy

   ```
   Content-Security-Policy: script-src 'nonce-IamAllowed00';
   ```

   the following two scripts are included in the page:

   Listing 2.2: Inline script with nonce value to execute

   ```
   <script nonce="IamNotAllowed00">handleUserClick();</script>
   ```

   Listing 2.3: Inline script with nonce value to execute

   ```
   <script nonce="IamAllowed00">showImageGallery();</script>
   ```

   The second script, " **showImageGallery()**" will be executed due to the correct nonce value, while "**handleUserClick()**" is not.

2. *hash-value*: A Website deploys a CSP policy that contains the following directive.

Listing 2.4: Content Security Policy

```
Content-Security-Policy: script-src 'sha256-letMe9911Run';
```

the following two scripts are included in the page:

Listing 2.5: Inline script with hash value to execute

```
<script integrity="sha256-letMe9911Run">handleButtonClick()
   ;</script>
```

Listing 2.6: Inline script with hash value to execute

```
<script integrity="sha256-DontletMe9911Run">submitForm();
   </script>
```

The first script, " **handleButtonClick()**" will be executed due to the correct identical computed hash value, while "**submitForm()**" is not.

Note: The values unsafe-inline and unsafe-eval are not possible values for the directive frame-ancestors. Furthermore, there are more possible values and directives than the previous Tables **(2.1 and 2.2)**.

2.2 Playwright

Playwright is a cross-browser test automation framework, i.e., a Node.js library; it supports various programming languages like Javascript, Typescript, Python, and other languages and it also supports the most famous browsers, such as **Firefox, Chrome, and WebKit Safari**.
Playwright allows the developer to run the browser during the tests with or without user interface, referred to as headed and headless modes. Furthermore, playwright supports the emulation of the mobile viewport that allows the developers to test web applications using different devices, playwright also enables the usage of desktop and mobile browsers using different User Agents, different browser versions, different geolocations, and other options, and hence the developers can determine the required options for their tests and write the tests using their preferred programming language.
Playwright enables to carry out multiple actions, such as the interaction with webpages as a user and performing usual user tasks, monitoring the network to change a request or a response, scanning for security, web scraping, and other actions [4].

2.2.1 User Agent

The User-Agent (UA) is one of the critical HTTP request headers, and it plays a crucial role in elucidating which device has been used by the user to access a website. The User-Agent contains valuable information and details about the user's device type, version number, and operating system (OS). Additionally, it presents the browser used by the user.
For example, the following UA string identifes a Chrome browser on an Android device "Samsung Galaxy S9+":

```
'Mozilla/5.0 (Linux; Android 8.0.0; SM-G965U Build/R16NW) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/112.0.5615.29 Mobile Safari/537.36'
```

Analyzing the UA string:

- ***'Mozilla/5.0'*** clarifies that the browser is suitable for Mozilla.
- ***'(Linux; Android 8.0.0; SM-G965U Build/R16NW)'*** specifies the OS (Linux, Android 8.0.0) and the device model as SM—G965U.
- ***'AppleWebKit/537.36'*** illustrate the usage of WebKit engine, '(KHTML, like Gecko)' clarifies that the browser supports the KHTML engine.
- ***'Chrome/112.0.5615.29'*** represent the browser's name and version.
- ***'Mobile Safari/537.36'*** suggests a browser based on Mobile Safari, though not limited to iOS devices.

3 Motivation and Dataset

3.1 Motivation

In our current era, the world wide web and especially visiting websites have become essential thing in our daily lives, and since the protection of these sites requires adequate protection, many previous studies discussed web security inconsistencies and what causes these inconsistencies, and they focused on the inconsistency in the security header Content Security Policy and the factors that cause this inconsistency, such as the geographical vantage points, accept-language and the User Agent **[11]**, the User Agent play a very essential role in causing the inconsistency; for example, the presence of a security header can be inconsistent through the usage of two different User Agents that are well-formed and exist **[6]**, this inconsistency gives the attacker the ability to perform various types of attacks, such as MITM (man-in-the-middle) by modifying the User Agent from the request of the client to receive a response that does not include a specific security header, the attacker in this case, takes advantage of the different configurations of a specific security header between different User Agents. In conclusion, inconsistencies in the protection against specific types of attacks can rely on the User Agent header of the request. In contrast to the previous studies, this work analyzes and checks how the website will react if the used User Agent is malformed or does not exist. Simultaneously, nonce values have gained widespread popularity in web security, in view of the fact that nonces ensure protection against different types of attacks by providing a unique identification value to control the execution and loading of content on the websites. As their usage has spread, the question arises: **Can these unique values be duplicated or reused?** Moreover, **under which circumstances can these values reused or being duplicated? What impact does this have on web security?**; Our work aims to answer these questions by exploring the potential scenarios in which nonce value can be reused or duplicated. Moreover, the relationship between a homepage and its subpages is essential in web security since, to ensure the total security of a homepage, we have to pay attention to the security of its subpages since a weak subpage could be the entry point for an attacker to perform certain types of attacks; our work is focusing on this relationship to emphasize the importance of consistent security measures across all page, protecting the website as a whole.

3.2 Dataset

Homepages-Dataset:
The dataset contains the most visited 1992 websites (homepages) based on the Tranco list [7].
Subpages-Dataset:
The dataset contains between 1 to 5 subpages from different paths (in case there are multiple paths), which our script extracted from each homepage after visiting it using Playwright.
The elements of this dataset are URLs, for example, `https://www.facebook.com/#`.

4 Methodology and challenges

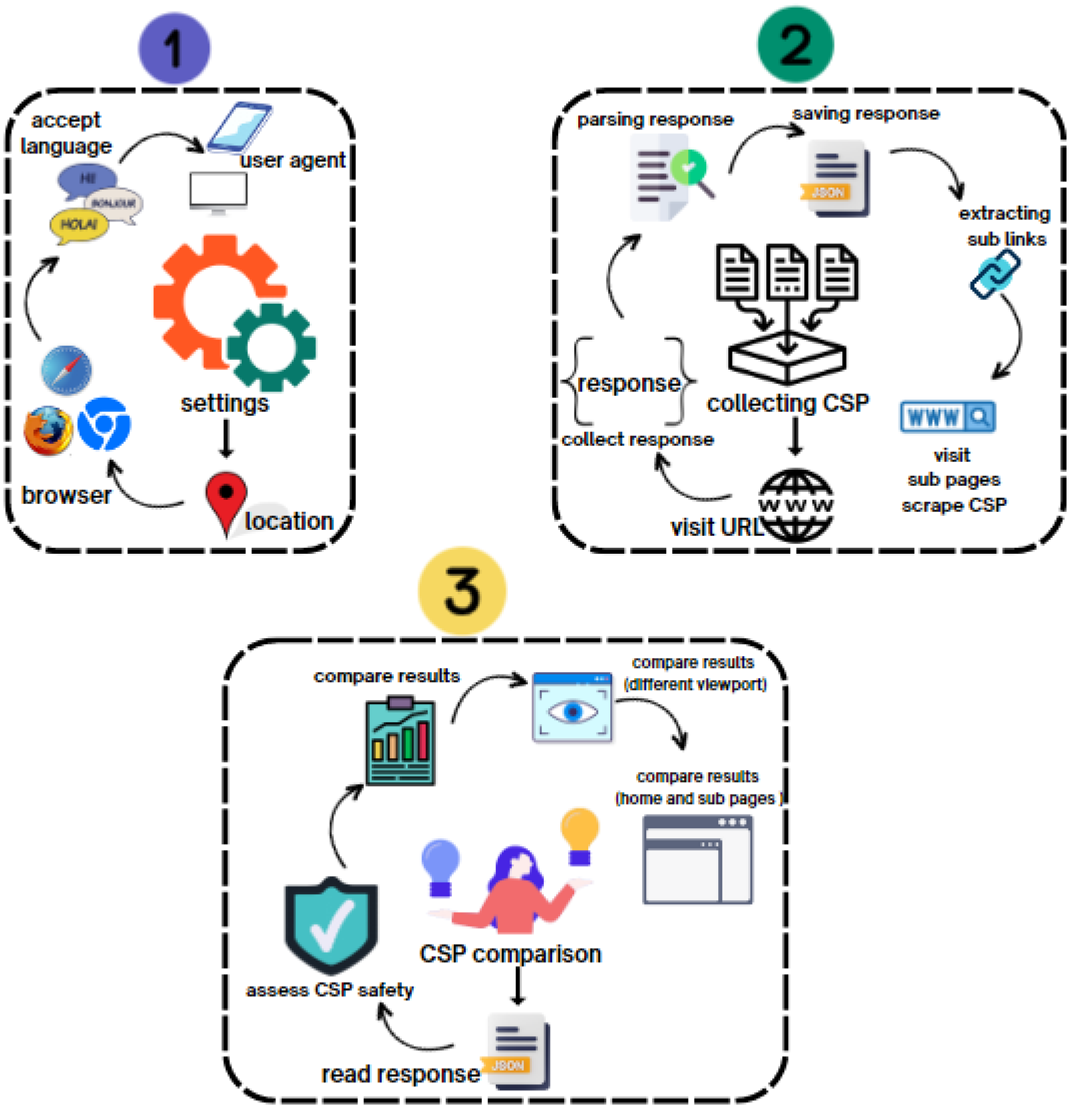

Figure 4.1: CSP Analysis Methodology: Step 1 - Selection of Parameters; Step 2 - Website Navigation and Response Collection; Step 3 - Response Header Analysis for CSP Safety; Includes Viewport and Home/Subpage Comparisons

4.1 Collecting CSP from sites

4.1.1 Important factors

To scrape the Content Security Policy from a website, we will consider some essential factors for this work; these factors must be considered before running the experiments (step 1 in Figure 4.1):

Table 4.1: Factors

Factor	Description
Geolocation Coordinates	latitude and longitude of the testing location
Selecting Browsers	chosen browser for the test
Accept Language	language of the client
User Agents	Specifications of User Agents used

- ***Geolocation Coordinates:***
  The first step is to determine the latitude and longitude; in this way, the experiments can be executed from different vantage points, and hence we can observe how the websites react due to the different vantage points.
  The importance of this comes from the result of prior work[11], which showed that visiting a website from two different vantage points can lead to different security policies.
  As an example for the geolocation coordinates, if we want to conduct the experiment from Rome in Italy, we can use the following coordinates ***{latitude: 41.8898855, longitude: 12.4913723}***.

- ***Selecting Browsers:***
  For each experiment, we must decide which desktop and mobile browsers we want to use and which mobile and desktop devices; In this work, we consider three popular browsers:

  Table 4.2: Browsers

Desktop Browsers	Mobile Browsers
Google Chrome	Mobile Browsers
WebKit (Safari)	Mobile Safari on iOS and Android
Mozilla Firefox	

  The work aims to analyze how CSPs can differ from each other due to using different desktop and mobile browsers to visit a website.
  Furthermore, selecting the browser will lead to selecting the User Agents that use this selected browser as a default browser.
  For example, if Chrome is the selected browser, the experiment will use the mobile device **"Galaxy S9+"** and other devices that use Chrome as a default browser.
  To specify which device uses Chrome, Firefox, and WebKit (Safari) as a default browser, we checked the Playwright's devices list; the list contains names of devices, and each device from this list has a specific JSON configuration file, and hence we built a small parser that takes each JSON configuration file for each device from the list

and checks whether the device uses Chrome, Firefox, or WebKit(Safari) as a default browser.
Example for a JSON configuration file of **"iPhone 13"**:

```
{
  "userAgent":"Mozilla/5.0 (iPhone; CPU iPhone OS 15_0 like Mac OS X)
      AppleWebKit/605.1.15 (KHTML, like Gecko) Version/16.4 Mobile/15
      E148 Safari/604.1",
  "viewport":{"width":390, "height":664},
  "screen:{"width":390, "height":844},
  "deviceScaleFactor":3,
  "isMobile":true,
  "hasTouch":true,
 "defaultBrowserType":"webki"
}
```

Table 4.3: Devices

Browser	Devices	OS Version
Google Chrome	Galaxy S9+	Android 8.0.0
	Galaxy S9+ landscape	Android 8.0.0
	Galaxy Tab S4	Android 8.1.0
	Galaxy Tab S4 landscape	Android 8.1.0
	Desktop Chrome	Windows 10 (64-bit)
WebKit (Safari)	iPhone 13	iOS 15.0
	iPhone 13 landscape	iOS 15.0
	Galaxy Note 3	Android 4.3
	Galaxy Note 3 landscape	Android 4.3
	Desktop Safari	macOS 10.15.7 (Catalina)
Mozilla Firefox	Desktop Firefox	Windows 10 (64-bit)

For Chrome we considered the following devices {**'Galaxy S9+', 'Galaxy S9+ landscape', 'Galaxy Tab S4', 'Galaxy Tab S4 landscape', 'Desktop Chrome'**}, for WebKit (Safari) {**'iPhone 13', 'iPhone 13 landscape', 'Galaxy Note 3', 'Galaxy Note 3 landscape', 'Desktop Safari'**}, and for Firfox {**'Desktop Firefox'**}.
Unexpectedly, the device "Galaxy Note 3"uses Mobile Safari as a default browser; the reason behind it appears in its User Agent string:

```
{
  "userAgent":\"Mozilla/5.0 (Linux; U; Android 4.3; en-us; SM-N900T
      Build/JSS15J) AppleWebKit/534.30 (KHTML, like Gecko) Version/16.4
       Mobile Safari/534.30\"
}
```

The User Agent string of this device contains "Mobile Safari"which is the Android version of Safari, which led us to the fact that Safari is also used as a default browser by some Android devices, such as "Galaxy Note 3".

Question: What is the difference between 'Galaxy S9+', ' Galaxy S9+ landscape'? The difference is just in the viewport of the device 'Galaxy S9+' since different viewports can affect the consistency of CSPs and hence lead to different CSPs [11]; we considered the case of using a device with two different viewports.

Challenge :
We observed that for the Firefox browser, we have only one desktop device, which is **'Desktop Firefox'**.
As a result, we added two devices running on the Android system **'Galaxy S9+',' Galaxy S9+ landscape'**, and two devices running on the iOS system **'iPhone 13',' iPhone 13 landscape'**, When the experiment uses Firefox browser for the devices that do not use it as a default browser, and then we face a specific issue while creating a new browser context, the received error is "`browser.new context: options.isMobile is not supported in Firefox` "; The cause of it is the property "isMobile"in the device Information, e.g. (device Information of iPhone 12 Pro landscape):

```
{
  userAgent: 'Mozilla/5.0 (iPhone; CPU iPhone OS 14_4 like Mac OS X)
      AppleWebKit/605.1.15 (KHTML, like Gecko) Version/16.4 Mobile/15
      E148 Safari/604.1',
  viewport: \{ width: 750, height: 340 \},
  screen: \{ width: 390, height: 844 \},
  deviceScaleFactor: 3,
  isMobile: true,
  hasTouch: true,
  defaultBrowserType: 'webkit'
}
```

The solution to this issue is just removing the isMobile property from the device information object, and this will not affect the experiment because the device information still contains essential properties such as `User Agent`, `viewport`, `defaultBrowserType`, and other properties after removing the `isMobile` property, the device information will look as follows:

```
{
  userAgent: 'Mozilla/5.0 (iPhone; CPU iPhone OS 14_4 like Mac OS X)
      AppleWebKit/605.1.15 (KHTML, like Gecko) Version/16.4 Mobile/15
      E148 Safari/604.1',
  viewport: { width: 750, height: 340 },
  screen: { width: 390, height: 844 },
  deviceScaleFactor: 3,
  hasTouch: true,
  defaultBrowserType: 'webkit'
}
```

As a result, this solution allows the experiment to be conducted using a Firefox browser with any device that uses one of the following OS: Android and iOS, and does not use Firefox as a default browser.

- ***Accept Language:***

  A prior work by *Sebastian Roth, Stefano Calzavara, Moritz Wilhelm, Alvise Rabitti, and Ben Stock* **[11]** indicates that the different languages of two users who want to visit the same website can affect the security policies and lead to having two different CSPs for both users, in this work, the factor accept language has been considered, accept language is an HTTP request header that describes the language of the user, who sent the request, the goal is to observe how the website responds to different users' language preferences
  Each language has the following form {language-COUNTRY CODE}, for example, en-US, en is for English, and the US specifies the United States of America. In this work, three types of accepted language have been considered, as shown below in Table **4.4**.

  Table 4.4: Types of accept language

Accept Language	**Status**
fr-FR, en-US, de-DE	Supported
ab-YZ	Unsupported
{AR;q=10}	Malformed

  The goal is to observe the visited website's reaction to an unsupported or malformed accepted language.

- ***User Agents:***

  Since the usage of different User Agents can influence the consistency of CSP, which mean if two users use two different User Agents to visit a website, each one of them can get different CSPs due to the different User Agents [11]. Furthermore, the difference in User Agents is significant for the attacker to perform certain types of attacks like **man-in-the-middle** (MITM) due to the inconsistency in security policies, such as ***Presence Inconsistency*** that different User Agents can cause [6];

  Will the websites reject or encounter these types of User Agents? To answer this question, three types of User Agents in this work have been considered **(Table 4.5)**:

  1. Existed (used and well-formed) User Agents.
  2. Non-existing User Agents, e.g., Mobile/6.0 (Mobile Android 27.6; FakeUA/6).
  3. Malformed User Agents, e.g., Mobile/91.0 (Windows; 14.1).
     Why the User agent "`Mobile/91.0 (Windows; 14.1)`" is malformed, or how can we define a User Agent as malformed?
     A well-formed User Agent string contains information and details about the user's device type **2.2.1**, such as OS and browser version; the provided User Agent is not in the usual format that we would expect for well-formed User Agents strings; Furthermore, some essential information is missing, such as browser name, and hence the provided User Agent is malformed.

  This work used four non-existing User Agents and two malformed User Agents, as mentioned above in Table **4.5**.

  The goal is to perform various experiments; each experiment considers one type of the previously mentioned above User Agents and analyze how websites react to these different types of User Agents and how this affects the consistency of CSPs.

Table 4.5: User Agents

Non-Existing User Agents	
Chrome and Firefox mobile browser	Mobile/6.0 (Mobile; Android 27.6; FakeUA/6)
Webkit (mobile safari) mobile browser	Mobile/6.0 (Mobile; iOS 14.1; FakeUA/2)
Chrome and Firefox desktop browser	Desktop/114.0 (Windows NT 13.9; Windows; x32; FakeUA/6)
Webkit (Safari) desktop browser	Desktop/114.0 (Windows NT 13.9; macOS; x32; FakeUA/2)
Malformed User Agents	
Chrome, Firefox and mobile safari	Mobile/1.0 (Android; 12.8)
Chrome, Firefox and Webkit	Chrome/91.0 (Windows; 14.1)

4.1.2 Scraping HTTP response headers and collecting CSPs:

The following tasks are executed to scrape the Content Security Policy (Step 2 in Figure 4.1).

4.1.2.1 Navigating to websites

The methodology deploys the library Playwright to scrape the Content Security Policy from the HTTP response headers.
The process begins with iterating through the homepages given as input as a text file or JSON file for the experiment.
For each website(Homepage), we check the availability of the site and verify if it is accessible or not, and split the websites into two groups:

1. Available online websites e.g., youtube.com is reachable.
2. Unavailable offline websites e.g., a-msedge.net is not reachable.

To specify whether a website is reachable or not, we are performing a small test using Chrome desktop browser on a desktop device that works on Windows; this test will run in a headed mode to determine which website is reachable and which website is not.
We create two arrays to store the available and unavailable websites; Next, we save these two lists as text files to file system. This process can be conducted as shown below **4.1**:

Listing 4.1: Checking reachable websites

```
(async () => {
    let browser = await playwright.chromium.launch({headless: false, timeout: 30
        * 1000});
    context = await browser.newContext({
        ...devices['Desktop Chrome'],
        premissions: ['geolocation'],
        geolocation: {latitude: 40.4081906, longitude: -3.6894398},
        locale: "de-DE",
        ignoreHTTPSErrors: true
    });
    let page = await context.newPage();
    // checking the availability of websites
    for(var w of websites){
        try{
            await page.goto(`https://www.${w}`, { waitUntil: "load"})
            reachableWebsites.push(w)
        }catch(e){
            notReachableWebsites.push(w)
        }
    }
    await context.close();
    await browser.close();
    // Saving these two arrays in file system
    fs.writeFileSync(`./ReachableWebsites.txt`, ...)
    fs.writeFileSync(`./NotReachableWebsites.txt`, ...)
})();
```

In our work, we only consider reachable websites for the CSP collection step.
Navigation to the homepage is conducted using Playwright, as mentioned previously in the Homepages-Dataset section **3.2**. Each element in this dataset is a domain name, such as facebook.com. Before navigating the website, the domain name is modified to `https://www.facebook.com`.
Then using the goto() function a navigation to the website is performed

```
    await page.goto(URL, options);
```

Before reaching this step, first of all, we create an instance of the browser used for the experiment; once the instance of the browser is created, the browser context must be defined to specify the device (Desktop computer or phone), the geolocation coordinates (latitude and longitude from where the experiment must be conducted), and the accept language, Next, a page object is defined to navigate the website.
As an example, navigating the homepage of Facebook is conducted using Chrome mobile browser is shown below **4.2**:

Listing 4.2: Navigation to a website label

```
(async () => {
    ....
    const browser = await playwright.chromium.launch({ headless: true, timeout:
        30 * 1000});
    const context = await browser.newContext({
                ...devices['Galaxy S9+'],
                premissions: ['geolocation'],
                geolocation: {latitude: 40.4081906, longitude: -3.6894398},
                locale: "de-DE",
                ignoreHTTPSErrors: true
              });
    const page = await context.newPage();
    const url = 'https://www.facebook.com';
    await page.goto(url);
    ....
    await browser.close();
})();
```

Challenge:
If the experiment should run using non-existing or malformed User Agents, an issue occurs in creating the browser context because Playwright does not support them; they are not registered in Playwright. The solution to this issue is done in two steps (before creating the browser context):

1. Define the device information for these User Agents such as name, userAgent, viewport and browser information **(Javascript Code 4.3)**.

2. Register the device in Playwright **(Javascript Code 4.4)**.

Listing 4.3: Defining device information

```
(async () => {
....
const browser = await playwright.chromium.launch({ headless: true, timeout: 30 * 1000});
// Device information
let device = {
             name: "random device",
             userAgent: "Mobile/6.0 (Mobile; Android 27.6; FakeUA/6)",
             viewport: {
               width: 1280,
               height: 720,
               deviceScaleFactor: 1,
               isMobile: true,
               hasTouch: true,
               isLandscape: false,
             },
             browserName: 'chromium',
             browserVersion: browser.version(),
           };
const context = await browser.newContext({
           ...devices["random device"],
           premissions: ['geolocation'],
           geolocation: {latitude: lat, longitude: lon},
           locale: `${acceptLanguage}`,
           ignoreHTTPSErrors: true
         });
....
})();
```

Listing 4.4: Register device information

```
(async () => {
    .
    .
    const browser = await playwright.chromium.launch({ headless: true, timeout: 30 * 1000});
    let device = {
                  name: "random device",
                  userAgent: "Mobile/6.0 (Mobile; Android 27.6; FakeUA/6)",
                  viewport: {
                    width: 1280,
                    height: 720,
                    deviceScaleFactor: 1,
                    isMobile: true,
                    hasTouch: true,
                    isLandscape: false,
                  },
                  browserName: 'chromium',
                  browserVersion: browser.version(),
                };
    // device registeration
    devices[deviceName] = device
    const context = await browser.newContext({
                ...devices["random device"],
                premissions: ['geolocation'],
                geolocation: {latitude: lat, longitude: lon},
                locale: `${acceptLanguage}`,
                ignoreHTTPSErrors: true
              });
    .
    .
})();
```

4.1.2.2 Scraping response headers

After navigating each accessible website, we start the process of scraping the response headers from the server, since the focus is only on the response security header, Content Security Policy, we will consider only that response headers that correspond to the main HTML document, because the main content of the webpage usually includes the CSP headers; this is done by checking if the resource type is "document" or not **(Javascript Code 4.5)**,

Listing 4.5: Checking resource type

```
    await page.on("response", async (response) => {
        if(response.request().resourceType() == 'document'){
            let allHeaders = await response.headers();
        }
    });
```

An example how the response headers looks like **4.6**:

Listing 4.6: Example response headers for the homepage of apple.com

```
{
    ..
  'content-security-policy': "default-src 'self' blob: data: *.akamaized.net *.
      apple.com *.apple-mapkit.com *.cdn-apple.com *.organicfruitapps.com; child-
      src blob: embed.music.apple.com embed.podcasts.apple.com https://
      recyclingprogram.apple.com swdlp.apple.com www.apple.com www.instagram.com
      platform.twitter.com www.youtube-nocookie.com; img-src 'unsafe-inline' blob:
       data: *.apple.com *.apple-mapkit.com *.cdn-apple.com *.mzstatic.com;
      script-src 'unsafe-inline' 'unsafe-eval' blob: *.apple.com *.apple-mapkit.
      com www.instagram.com platform.twitter.com; style-src 'unsafe-inline' *.
      apple.com",
  ..
}
```

Challenge:
What if the website sent multiple CSPs?
To answer this question, we ask the following question, **does the same website send all CSPs?**
For example, navigating through `https://www.youtube.com` using Galaxy S9+ and Chrome as a browser to observe whether the same website sends all CSPs **(Javascript Code 4.7)**:

Listing 4.7: Response headers collection

```
    await page.on("response", async (response) => {
        if(response.request().resourceType() == 'document'){
            let allHeaders = await response.headers();
            console.log(response.url())
            console.log(allHeaders)
        }
    });
```

While navigating through `https://www.youtube.com/`, which is the link to the homepage, we get the following response headers **4.8**:

Listing 4.8: Example response headers for youtube.com

```
{
  ...
  'content-security-policy-report-only': "require-trusted-types-for 'script';
      report-uri /cspreport",
  'content-type': 'application/binary',
  'cross-origin-opener-policy-report-only': 'same-origin-allow-popups; report-to=
      "youtube_main"',
  ...
  location: 'https://m.youtube.com/',
  ...
}
```

Only the last response from the same website must be considered in this work, and hence we filter the coming response headers and collect only the ones that come from the targeted website, as follows **(Javascript Code4.9)**:

Listing 4.9: filter response headers that comes while navigating the website

```
await page.on("response", async (response) => {
    let href = response.url();
    if(response.request().resourceType() == 'document'
        && (href === url || href === `${url}/`) ){
        let allHeaders = await response.headers();
        .
        // collect response headers
    }
});
```

4.1.2.3 Parsing response headers and collecting CSPs

For the process of parsing the response headers, we built a custom parser, which operates as follows:

Step 1: The parser takes the collected response headers as input and splits every header with its value.

Step 2: Then it generates an array that contains all the response headers; each element of this array contains two indexes:

- The first index includes the name of the header.
- The second index contains an Array, where the value of this header has been saved.

We assume we have the following response headers:

Listing 4.10: Example response headers

```
{
  ...
  "content-security-policy": "script-src 'nonce-vOpnOE538' 'unsafe-inline",
  ...
}
```

```
[
  [content-security-policy, ["script-src 'nonce-vOpnOE538' 'unsafe-inline']],
]
```

4.1.2.4 Saving parsed response headers

Once the response headers have been parsed, they will be stored in an array, and then a *JSON* file will be created to store all the elements of this array. This *JSON* file is used later for the comparison process; for each reachable website, a *JSON* file will be stored in

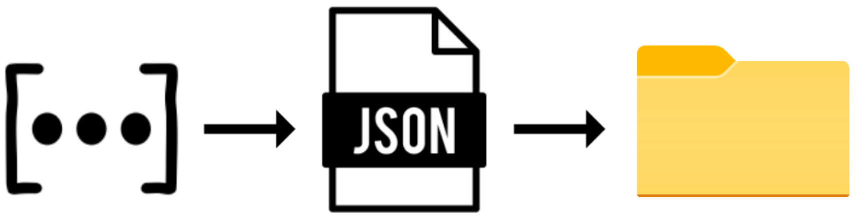

Figure 4.2: Saving the response headers to file system

a folder.
The User Agent headers have been saved with each website's URL to ensure that the results are not biased in the presence of bot detectors.
Additionally, in case of a website that is deemed unreachable, for each device that has been used to navigate to the website a JSON file containing the URL of the website will be saved in a separate folder.

4.1.2.5 Extracting sublinks

While navigating through the homepage, we begin the process of extracting the sublinks of the visited homepage:

Step 1: Collect all URLs of subpages.

Step 2: The paths of the extracted subpages are extracted and saved in an array to save one to five subpages with different paths (if it is possible).

Step 3: We iterate through both arrays, the one which contains the subpages and the array that contains the extracted paths to generate a dataset that encompasses a diverse set of sublinks.

Step 4: These sublinks are saved then in a text file, which represents the dataset of the subpages that we need for the experiments, each line of this text file contains two to six URLs separated by a space, the first one representing the URL of the homepage, while the remaining URLs are the links to the subpages.

For example, while navigating `https://www.apple.com`, we could extract 313 URLs and 191 paths, the algorithm that we built will save one to five subpages that have different paths.

4.1.2.6 Navigating through subpages and scrape CSPs

Before navigating through the subpages, an observation has been made regarding the order of the subpages in the dataset, the subpages are saved in an orderly manner, back to the experiment setup, each experiment often uses five different devices, and hence all subpages of the same homepage will be navigated by these devices sequentially.
However, this sequential navigation increments the risk of being perceived as a bot by the website server.
To avoid and decrease this risk, the following steps are taken before starting navigating and scraping CSPs:

1. Extracting all subpages from the dataset and saving them in an array.
2. Reordering the subpages inside the array by shuffling them to have these subpages in a randomized manner.

These two steps decrease the risk of being flagged as a bot. Once the dataset is shuffled, we start the process of navigating subpages and scraping CSP for each website, the process follows the same methodology for the homepages without extracting subpages from within the subpages themselves.

4.2 CSP comparison

First of all, the browsers that have been utilized for the experiment are extracted and stored in an array, we iterate through the browser array, and for each browser; The following tasks will be executed sequentially (Step 3 in Figure 4.1):

4.2.1 comparison process

Step 1: ***Reading JSON files***:
Reading the *JSON* files that are stored in a specific folder, in this step, only the *JSON* files that contain the response headers that have been collected while navigating the website using the current iterated browser and website.

Step 2: ***Collecting the name and the contents of JSON files in an Array***:
Since multiple devices have visited each website, the stored *JSON* files while navigating through the current website using these devices will be parsed and stored in an array.
For example, the current browser is Chrome, and the current website is `youtube.com`. The navigation `youtube.com` has been performed using the devices that use Chrome browser as a default browser **(Table 4.3).**
Moreover, the *JSON* files inside the targeted folder will be considered, such as :

```
csp_Desktop Chrome_Chrome_112.0.5615.29_Windows_720_1280_10_youtube-com.json
```

Each element in the resulting array is an array that contains both the JSON file's name and the response headers saved within that file.

Step 3: ***Assessing implemented CSP safety against XSS***: To specify whether a CSP is safe or not, the following algorithm has been created to check the CSP's safety.
Initially, the algorithm checks if the response headers include the security header content-security-policy; if this is the case:

a) **PHASE.1**:

Algorithm 1 Process CSP Directive

```
1: Input: CSP (Content Security Policy)
2: if CSP contains 'script-src-elem' directive then
3:     Return 'script-src-elem' and execute Algorithm 2
4: else if CSP contains 'script-src' directive then
5:     Return 'script-src' and execute Algorithm 2
6: else if CSP contains 'default-src' directive then
7:     Return 'default-src' and execute Algorithm 2
8: else
9:     CSP is not safe
10: end if
```

b) **PHASE.2**:

Algorithm 2 Specifying CSP safety against XSS

```
1: Input: D (Returned directive from Algorithm.1)
2: if D contains 'nonce-value' or 'hash-value' then
3:     Return CSP is safe
4: else if D contains '*' or 'full scheme' then
5:     Return CSP is not safe against XSS
6: else
7:     Return CSP is safe against XSS
8: end if
```

Assume we have the following two CSPs:

Listing 4.11: Content Security Policy (CSP-1 and CSP-2)

```
Content-Security-Policy: script-src 'nonce-sdrFDEr321FGWE'
    'unsafe-inline';
Content-Security-Policy: script-src 'https:' 'self';
```

Returning to the algorithm mentioned above, the first Content Security Policy header(CSP-1) is safe, while (CSP-2) is not.

Challenge: ***What if the same website send more than one CSP ?*** Solution is shown in the following Algorithm:

Algorithm 3 Checking Website's CSP Safety

```
1:  **Input:** List of CSPs from the same website
2:  Initialize variable *counter* as 0
3:  **for** each CSP collected from the same website **do**
4:      Execute **Algorithm 2** for the CSP
5:      **if Algorithm 2** returns a safe CSP **then**
6:          Increment *counter* by 1
7:      **end if**
8:  **end for**
9:  **if** $counter > 0$ **then**
10:     The website deploys a safe CSP against XSS
11: **else**
12:     The website does not deploy a safe CSP against XSS
13: **end if**
```

Step 4: ***Saving the results of the previous algorithms in* JSON *files***
After running the algorithms and collecting the results from each one, a JSON file is created to store these results in addition to the device and browser information and the URL of the website.

4.2.2 Comparing the results

To make a comparison between the results we had saved earlier, we organized the data into two separate arrays, one for desktop devices and another for mobile devices.
Then, we examined the results for each website related to all devices present in the respective arrays. This method allowed us to assess and compare the outcomes for desktop and mobile categories separately:

- **Did the website deploy a CSP for desktop or mobile devices? we calculate and save the following information:**
  1. The number of total websites.
  2. The number of total device.
  3. The number of websites that deployed CSP for all devices.
  4. The websites that deployed CSP for all devices.
  5. The number of websites that did not deploy CSP for all devices.
  6. The websites that did not deploy CSP for all devices.
  7. The number of websites that deployed CSP for some devices.
  8. The websites and the URLs of these websites that deployed CSP for some devices, along with the device information that has been used to visit the website, which deploy a CSP.
- **Did the website deploy a safe CSP for desktop or mobile devices? we calculate and save the following information:**
  1. The number of total websites.

2. The number of total devices.
3. The number of websites that deployed a safe CSP for all devices.
4. The websites that deployed a safe CSP for all devices.
5. The number of websites that did not deploy a safe CSP for all devices.
6. The websites that did not deploy a safe CSP for all devices.
7. The number of websites that deployed a safe CSP for some devices.
8. The websites and the URLs of these websites that deployed a safe CSP for some devices, along with the device information that has been used to visit the website, which deploy a safe CSP.

Examples:
Two examples to clarify how the last results look like:
Navigating five websites made by four mobile devices and one desktop device using Chrome.

Listing 4.12: Example results for deploying CSP using Chrome, one desktop device and four mobile devices (Andorid)

```
{
  "number of total websites": 5,
  "number of total devices": 5,
  "The number of websites that deployed CSP for all devices": 2,
  "The websites that deployed CSP for all devices": ["https://www.amazon.com
      ","https://www.dropbox.com"]
}
```

This example of comparison results shows that from 5 homepages, only the homepages of `amazon.com` and `dropbox.com` deployed a CSP only when we navigate through it using the desktop device and the four mobile devices that use Chrome as a default browser **(Table 4.3)**.

Listing 4.13: Example results for deploying a safe CSP using Chrome, one desktop device and four mobile devices (Andorid)

```
{
  "number of total websites": 5,
  "number of total devices": 5,
  "The number of websites that deployed a safe CSP for all devices": 1,
  "The websites that deployed a safe CSP for all devices":
  ["https://www.dropbox.com"],
}
```

The second example expresses that only the homepage of `dropbox.com` deployed a safe CSP for all devices.

4.2.3 Comparing mobile devices with two different viewports

The goal here is to investigate the impact of similar devices with different viewports on the consistency of CSP by iterating through each visited website and compare the results between the devices that only have one difference between themselves, which is the viewport, such as '`Galaxy S9+`', and '`Galaxy S9+ landscape`'.

In this section, we only focus on mobile devices; assume an experiment has been performed to visit ten websites using Firefox and four mobile devices to navigate through the website, and the results answer similar points as in the last subsection **(Subsection 4.2.2)** in order to measure how many websites deploy safe and unsafe CSP for similar devices with different viewports. The following example explains the results of this type of comparison:

Listing 4.14: Example results for deploying safe CSP using two identical mobile devices with different viewports

```
{
    "number of total pages":2,
    "total devices (each device with 2 different viewport)":
    ["Galaxy S9+","iPhone 13"],
    "number of pages (safe csp and not safe csp with diff view port)": 1,
    "pages (safe csp and not safe csp with diff view port)":[["https://policies.
        google.com/technologies/cookies", "Galaxy S9+ hXw:320X658, safeCSP: false
         ","Galaxy S9+ hXw:658X320, safeCSP: true "]]
}
```

We can observe that the website with the URL `https://policies.google.com/technologies/cookies` deployed a safe CSP using the device with the following viewport (height: 658, width: 320) while it deployed an unsafe CSP for the other device (height: 320, width: 658).

4.2.4 Comparing CSP safety between home and subpages

The purpose of this type of comparison is to observe the differences between CSPs of the homepage and their subpages; a prior work by *Marco Squarcina, Mauro Tempesta, Lorenzo Veronese, Stefano Calzavara, and Matteo Maffei* **[14]** clarifies that if the main domain deployed a safe CSP that provides the required protection against various types of attacks.
In contrast, one of the subdomains deploys a not well-configured CSP, and hence the provided protection for the main domain will not prevent the attacker from using the subdomain to attack the main domain; the idea of this comparison is to measure the number of homepages that deploy a safe CSP along with their subpages in case we navigate to these websites using all devices.
Assume we experimented with visiting the homepage of the website `zhihu.com` and two other subpages of this homepage; using Chrome as a desktop browser and a desktop device; after executing the comparison, the following result appears:

Listing 4.15: Example results for deploying a safe CSP by homepage along with their subpages

```
{
    "number of total homepages": 5,
    "number of Home and Subpages that deploy safe CSP for all user agents": 1,
    "Home and Subpages that deploy safe CSP for all user agents":
    [
     [ homepage :
       "https://www.zhihu.com" ],
     [ subpages :
       "https//www.zhihu.com/signinnext=https%3A%2F%2Fwww.zhihu.com%2F%3Futm_id%3
           D0]
    ]
}
```

In this small experiment, we could find out that using a desktop browser, the main page, along with the one mentioned subpages, deploy a safe CSP.

4.3 Nonce-Values comparison

Nonce values serve an essential purpose in web security **(Table 2.2)**.
These values are designed to be unique and never repeated within different responses for the same URL **[12]**.
This non-replication aspect is to o uphold the integrity and effectiveness of security policies.
In this context, our primary objective is to investigate whether nonce values could potentially be duplicated in different scenarios:

Scenario 1: Nonce Value duplication between the main page and its subpages.
To test this scenario, we are collecting each main page and its subpages and testing if one of the subpages uses the same nonce value as the homepage.
In this scenario, the subpage could use the same nonce value as the homepage; this duplication of the nonce value can cause weakness in the web security; hence, the attacker can use these duplicated nonce values.

Scenario 2: Nonce Value duplication between different websites.
In this scenario, we are collecting the websites along with their nonce values and testing if any websites share the same nonce value with other websites. This duplication scenario could compromise user privacy, facilitate unauthorized access, and affect data integrity.

Scenario 3: Nonce Value Duplication within the Same Website after Sending a New Request.
In this scenario, we send new requests to the website server and observe how nonce values are managed during subsequent interactions. Replicating nonce values presents a security concern within the internet environment and can significantly impact security policies and expose users to potential risks.

5 Results and Analysis

This section presents a detailed analysis of our findings from the different types of CSP comparison, We have conducted different experiments on the top 1992 websites (based on Tranco [7]), from these 1992 websites, 471 website were not reachable; We conduct navigations to 1521 homepages along with 1521 subpages to investigate whether specific factors such as user agents, geolocation coordinates, accepted language, and various types of browsers have an impact on deploying CSP by websites.
In these experiments we the three types of user agents (existing, non-existing, and malformed user gents), three types of browsers (Chrome, Firefox, Webkit), and five types of accept language (en-US, de-DE, ar-SA, AR;q=10, ab-YZ). We performed these experiment from 3 different geolocation coordinates (Madrid, Brazilia, Hyderabad).

5.1 CSP Deployment

CSP deployment plays a crucial role in enhancing the web security, it is a crucial security measure adopted by websites, in order to provdie the required protection against different web attacks.

5.1.1 Impact of User Agents on CSP Deployment

The deployment of Content Security Policy (CSP) can be affected by various types of factors, such as User Agents and other factors. We are focusing in our analysis on the effect of different types of these factors, such as different gelocations, different types of User Agents, and various types of accept language.

5.1.1.1 Malformed User Agents

This section presents the impact of malformed User Agents on deploying CSP by the websites; We want to observe how whether the malformed User Agent make the web applications struggle to enforce proper security policies by sending unusual User Agent data through HTTPS requests.

Different geolocations - Same accept language:

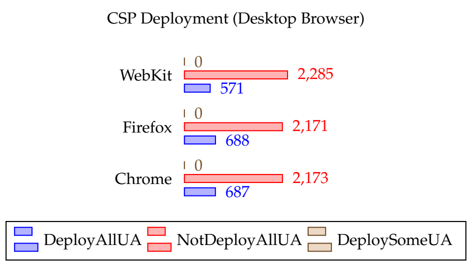

Figure 5.1: Desktop Browser

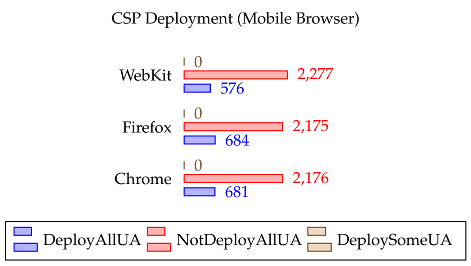

Figure 5.2: Mobile Browser

Figure 5.3: CSP Deployment - Malformed User Agents (geolocation: Brazilia, accept language: en-US

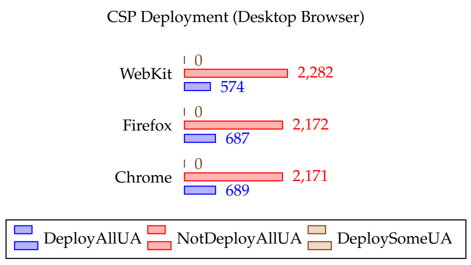

Figure 5.4: Desktop Browser

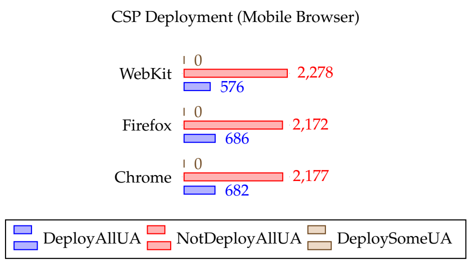

Figure 5.5: Mobile Browser

Figure 5.6: CSP Deployment - Malformed User Agents (geolocation: Hyderabad, accept language: en-US

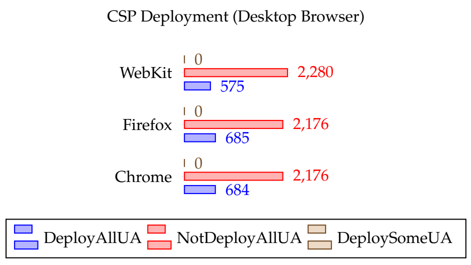

Figure 5.7: Desktop Browser

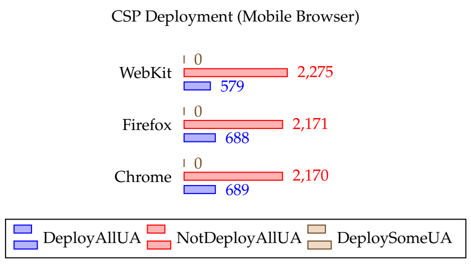

Figure 5.8: Mobile Browser

Figure 5.9: CSP Deployment - Malformed User Agents (geolocation: Madrid, accept language: en-US)

Same geolocation - different accept language:

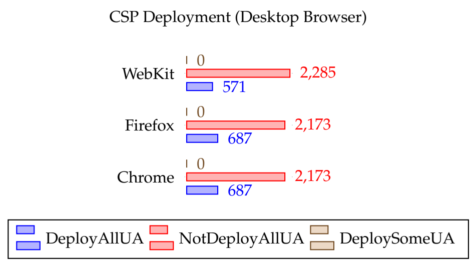

Figure 5.10: Desktop Browser

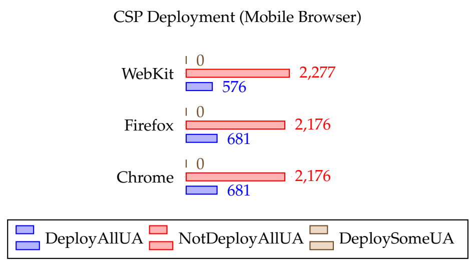

Figure 5.11: Mobile Browser

Figure 5.12: CSP Deployment - Malformed User Agents (geolocation: Madrid, accept language: en-US

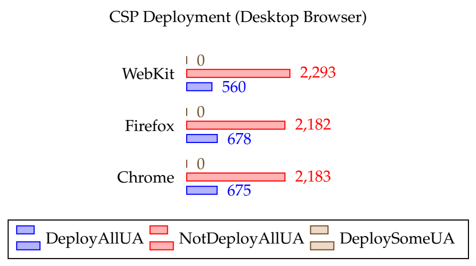

Figure 5.13: Desktop Browser

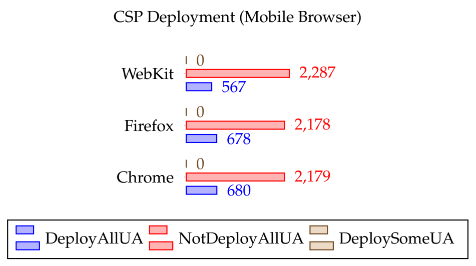

Figure 5.14: Mobile Browser

Figure 5.15: CSP Deployment - Malformed User Agents (geolocation: Madrid, accept language: de-DE

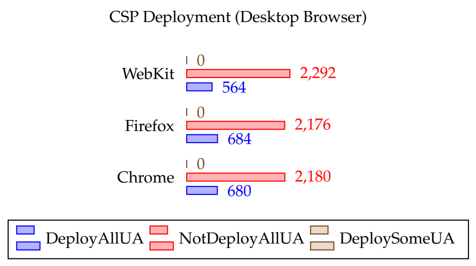

Figure 5.16: Desktop Browser

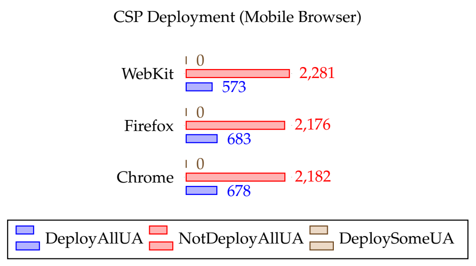

Figure 5.17: Mobile Browser

Figure 5.18: CSP Deployment - Malformed User Agents (geolocation: Madrid, accept language: ar-SA

Conclusion:
Based on the findings presented in the results, it can be observed for all types of used Browsers (Chrome, Firefox and WebkKit(Safari)) using malformed user agents from three different geolocations, the proportion of webpages deploying Content Security Policy (CSP) using both desktop and mobile browsers is consistently 22% for Chrome and Firefox, and 18% for WebKit (for all user agents).
For example, during the experiment, we could detect that the homepages of `apple.com`, `amazon.com`, and `whatsapp.com` deploy a CSP along with at least one subpage. Furthermore, the results illustrate that the difference between desktop browsers and mobile browsers in respect of the number of websites that deploy a CSP, is barely notable.

We only could observe, that the number of websites that deploy a CSP using WebKit (desktop and mobile browsers) along with malformed User Agents is small compared to Chrome and Firefox. In conclusion, the results from the three different locations *Brazilia, Hyderabad, and Madrid*, and three different accept languages (en-US, de-DE, and ar-SA) are similar to each other.

5.1.1.2 Non-Existing User Agents

This section demonstrate whether Non-Exisiting User Agents have impacts on CSP deployment; due to lack of the identification of User Agen's Information; We want to investigate if this type of User Agents can hinder the ability of the web server to enforce the deployment of CSP on the websites.
Different geolocations - Same accept language:

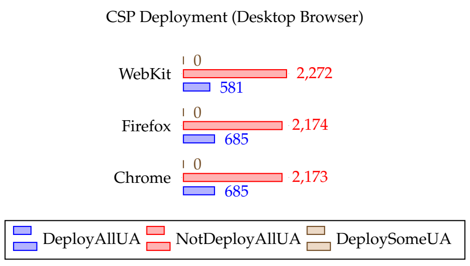

Figure 5.19: Desktop Browser

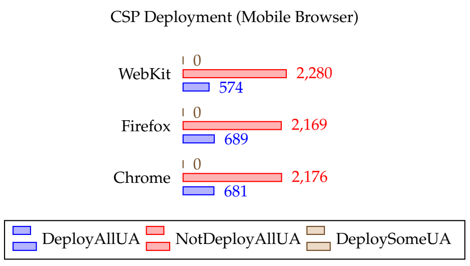

Figure 5.20: Mobile Browser

Figure 5.21: CSP Deployment - Non-Existing User Agents (geolocation: Madrid, accept language: en-US

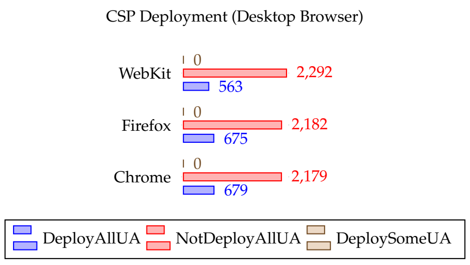

Figure 5.22: Desktop Browser

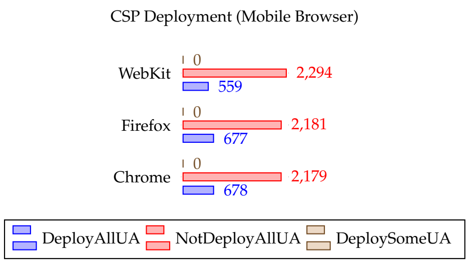

Figure 5.23: Mobile Browser

Figure 5.24: CSP Deployment - Non-Existing User Agents (geolocation: Madrid, accept language: de-DE

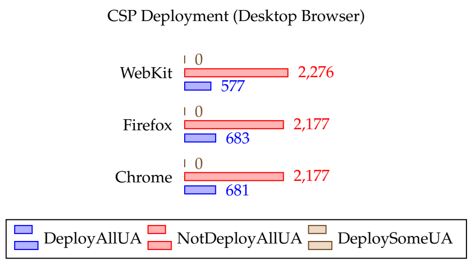

Figure 5.25: Desktop Browser

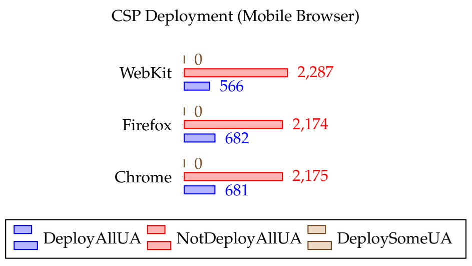

Figure 5.26: Mobile Browser

Figure 5.27: CSP Deployment - Non-Existing User Agents (geolocation: Madrid, accept language: ar-SA

Same geolocation - different accept language:

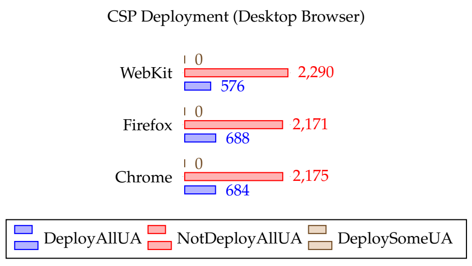

Figure 5.28: Desktop Browser

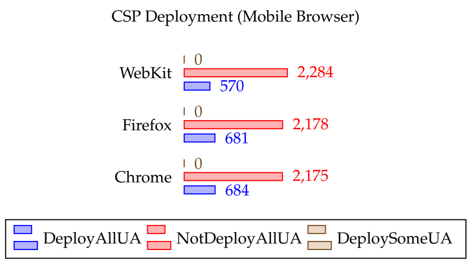

Figure 5.29: Mobile Browser

Figure 5.30: CSP Deployment - Non-Existing User Agents (geolocation: Madrid, accept language: en-US

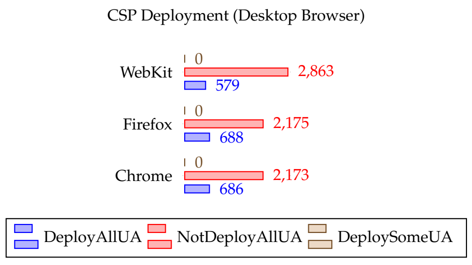

Figure 5.31: Desktop Browser

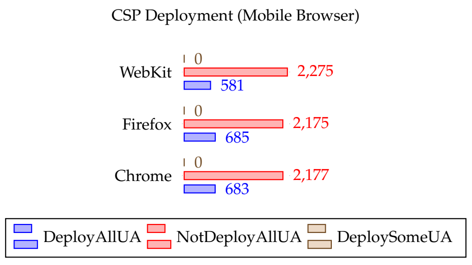

Figure 5.32: Mobile Browser

Figure 5.33: CSP Deployment - Non-Existing User Agents (geolocation: Madrid, accept language: de-DE

Conclusion:
We observed from the results that in case of using Chrome and Firefox (both desktop and mobile browsers) only 22% of the websites deploy a CSP, on the other hand 18% of websites deploy CSP when using WebKit browser. Moreover, the results for both experiments that included the usage of three different geolocations and three different accepted languages is not varied. As a conclusion, the number of websites that deploy CSP in case of using non-existing and malformed User Agents is barley identical. In conclusion, due to the previous results; The malformed and non-existing User agents have the same impact of CSP deployment on websites.

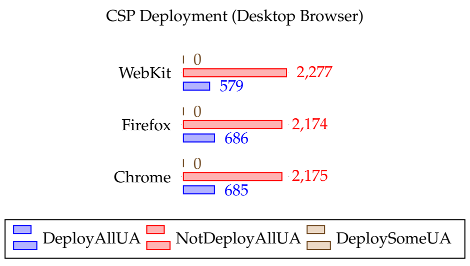

Figure 5.34: Desktop Browser

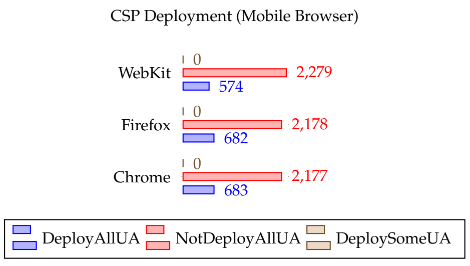

Figure 5.35: Mobile Browser

Figure 5.36: CSP Deployment - Non-Existing User Agents (geolocation: Madrid, accept language: ar-SA

5.2 CSP Safety against XSS

5.2.1 Impact of User Agents on CSP Safety

The Safety level of CSP can be affected by multiple types of factors, such as User Agents, geolocations, and accepted language **[11]**.

5.2.1.1 Malformed User Agents

This section explores how malformed User Agents impact secure CSP deployment on websites.
Different geolocations - Same accept language:
Page-Level:

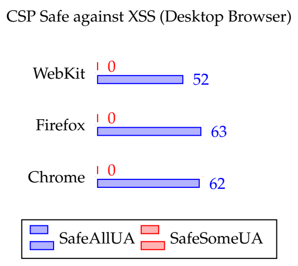

Figure 5.37: Desktop Browser

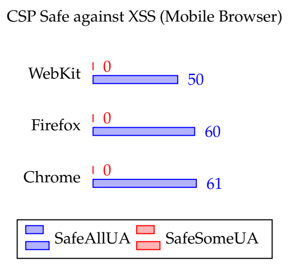

Figure 5.38: Mobile Browser

Figure 5.39: CSP Safety against XSS - Malformed User Agents (geolocation: Brazilia, accept language: en-US

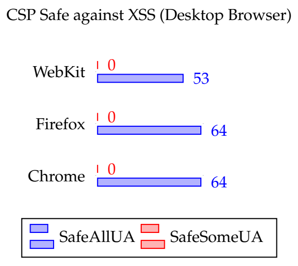

Figure 5.40: Desktop Browser

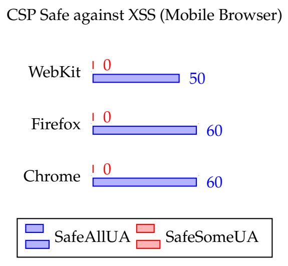

Figure 5.41: Mobile Browser

Figure 5.42: CSP Safety against XSS - Malformed User Agents (geolocation: Hyderabad, accept language: en-US

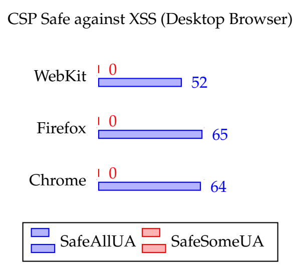

Figure 5.43: Desktop Browser

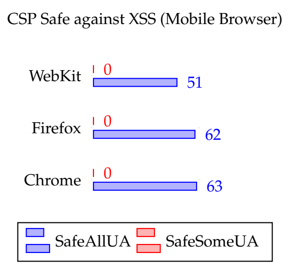

Figure 5.44: Mobile Browser

Figure 5.45: CSP Safety against XSS - Malformed User Agents (geolocation: Madrid, accept language: en-US)

Conclusion:
we observed consistent patterns across all three locations. The data revealed a noteworthy similarity in the percentage of websites employing a secure Content Security Policy (CSP). For Chrome and Firefox users, a substantial 60 to 65 percent of websites were seen to prioritize security by implementing CSPs. In contrast, Webkit-based browsers displayed a slightly lower range, with around 50 to 55 percent of websites utilizing secure CSPs.
Moreover, we took a closer look at how user agent differences between desktop and mobile browsers influenced the results. To our surprise, the disparities between the two platforms turned out to be minimal. This finding suggests that website administrators are equally committed to strengthening security through CSPs, regardless of the device users employ to access the sites.
In conclusion, our analysis points to Chrome and Firefox as the leading browsers in terms of websites adopting robust CSP deployments. While Webkit browsers also show considerable adoption of CSPs, they lag slightly behind their counterparts.

Website-Level:

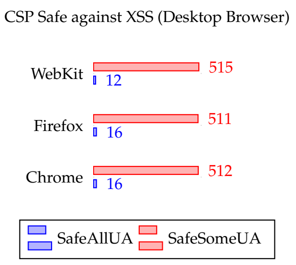

Figure 5.46: Desktop Browser

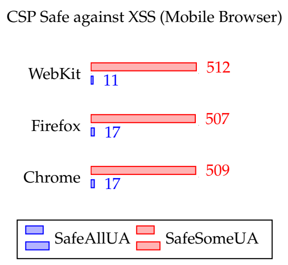

Figure 5.47: Mobile Browser

Figure 5.48: CSP Safety against XSS - Malformed User Agents (geolocation: Brazilia, accept language: en-US

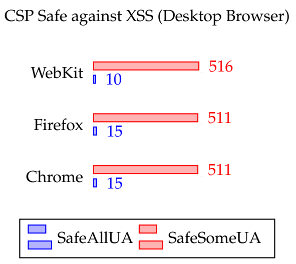

Figure 5.49: Desktop Browser

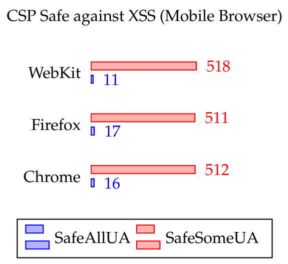

Figure 5.50: Mobile Browser

Figure 5.51: CSP Safety against XSS - Malformed User Agents (geolocation: Hyderabad, accept language: en-US

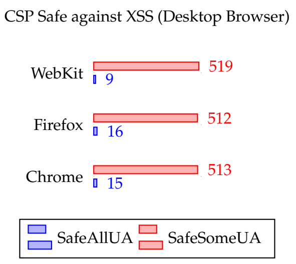

Figure 5.52: Desktop Browser

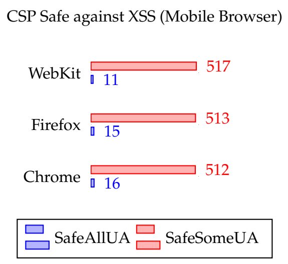

Figure 5.53: Mobile Browser

Figure 5.54: CSP Safety against XSS - Malformed User Agents (geolocation: Madrid, accept language: en-US

Conclusion: Based on our findings; when using Chrome and Firefox browsers, we observed that approximately 15 to 17 homepages that deployed a secure CSP along with a subpage, such as `snapchat.com`, `hubspot.com`, `tradingview.com` , and `zhihu.com`. On the other hand, when it came to Webkit-based browsers, the numbers were slightly lower, with about 9 to 11 websites implementing a secure CSP. Furthermore,the number of homepages with a secure CSP along with a subpage remained consistent when accessed through mobile browsers using Webkit browsers. In the context of Brazil's geolocation, we made an interesting observation. The number of homepages implementing CSP along with a subpage was identical for both Chrome and Firefox in that region. Moreover, we found that there was very small difference in the number of homepages with a secure CSP along with a subapge between desktop and mobile browsers.

5.2.1.2 Non-Existing User Agents

This section delves into the repercussions of non-existing User Agents on the implementation of a safe and robust Content Security Policy (CSP) by websites. **Different geolocations - Same accept language**:
Page-Level:

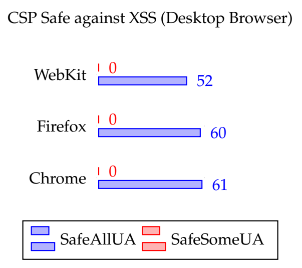

Figure 5.55: Desktop Browser

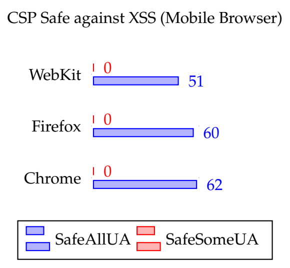

Figure 5.56: Mobile Browser

Figure 5.57: CSP Safety against XSS - Non-Existing User Agents (geolocation: Brazilia, accept language: en-US

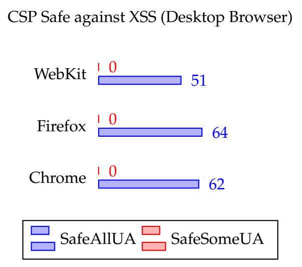

Figure 5.58: Desktop Browser

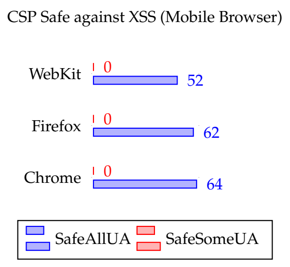

Figure 5.59: Mobile Browser

Figure 5.60: CSP Safety against XSS - Non-Existing User Agents (geolocation: Hyderabad, accept language: en-US

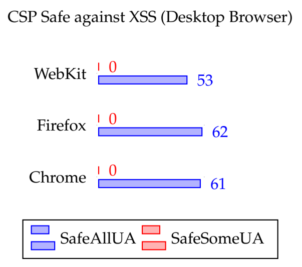

Figure 5.61: Desktop Browser

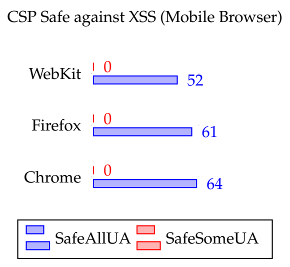

Figure 5.62: Mobile Browser

Figure 5.63: CSP Safety against XSS - Non-Existing User Agents (geolocation: Madrid, accept language: en-US)

Conclusion:
Just as we noticed with malformed User Agents, the impact of non-existing User Agents on websites' Content Security Policy (CSP) deployment was quite significant. Regardless of the specific geolocation, our research consistently revealed that a substantial number of websites have implemented secure CSP measures. Specifically, when looking at websites using Chrome and Firefox browsers, we found that approximately 60 to 65 of the webpages had a secure CSP. On the other hand, 50 to 55 webpages for Webkit browsers.

Website-Level:

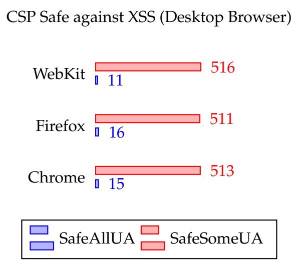

Figure 5.64: Desktop Browser

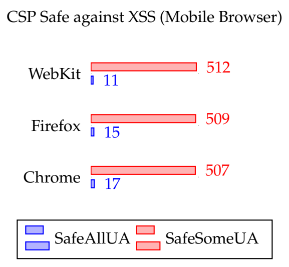

Figure 5.65: Mobile Browser

Figure 5.66: CSP Safety against XSS - Non-Existing User Agents (geolocation: Brazilia, accept language: en-US

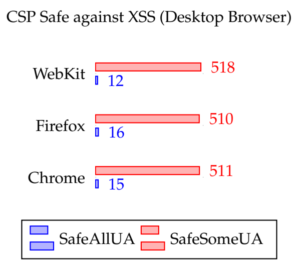

Figure 5.67: Desktop Browser

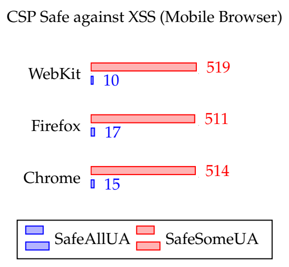

Figure 5.68: Mobile Browser

Figure 5.69: CSP Safety against XSS - Non-Existing User Agents (geolocation: Hyderabad, accept language: en-US

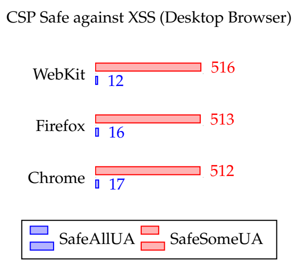

Figure 5.70: Desktop Browser

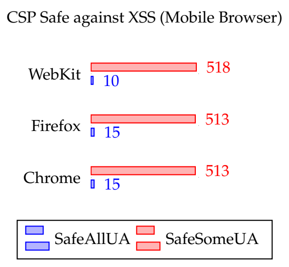

Figure 5.71: Mobile Browser

Figure 5.72: CSP Safety against XSS - Non-Existing User Agents (geolocation: Madrid, accept language: en-US

Conclusion:
We found that a significant number of homepages that deploy a safe CSP along with one of its subpages, about 15 to 17, implemented secure Content Security Policies (CSP) for Chrome and Firefox browsers. This reflects a strong dedication to web security among these websites. For Webkit-based browsers, the number was slightly lower, with around 10 to 12 websites adopting secure CSPs.
Notably, we observed minimal differences in implementing secure CSPs between desktop and mobile browsers, indicating a consistent approach to security measures across platforms. Additionally, in the specific geolocation of Madrid, the number of homepages and subpages that deploy together safe CSPs was identical for both Chrome and Firefox, signifying a shared emphasis on security in that region.

5.2.1.3 Existing User Agents

We conduct a thorough analysis to understand how existing well-formed User Agents can affect the safety and effectiveness of the deployed Content Security Policy (CSP). **Different geolocations - Same accept language**:
Page-Level:

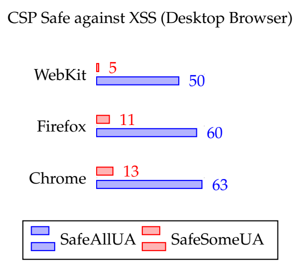

Figure 5.73: Desktop Browser

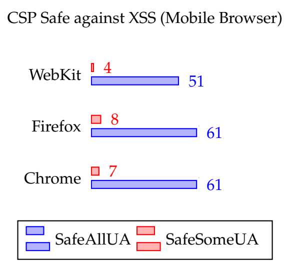

Figure 5.74: Mobile Browser

Figure 5.75: CSP Safety against XSS - Existing User Agents (geolocation: Brazilia, accept language: en-US

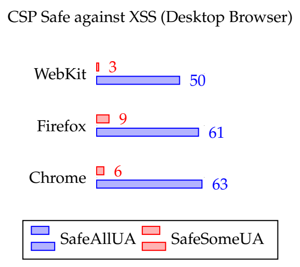

Figure 5.76: Desktop Browser

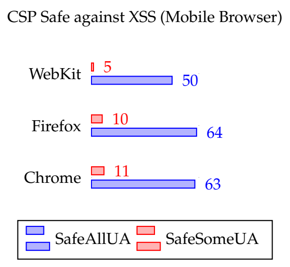

Figure 5.77: Mobile Browser

Figure 5.78: CSP Safety against XSS - Existing User Agents (geolocation: Hyderabad, accept language: en-US

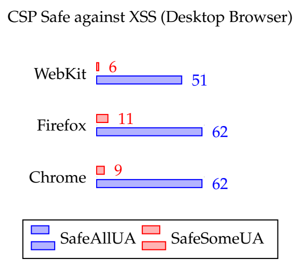

Figure 5.79: Desktop Browser

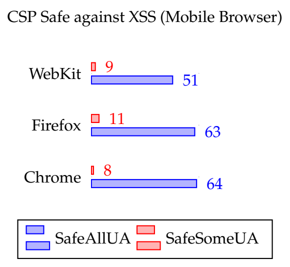

Figure 5.80: Mobile Browser

Figure 5.81: CSP Safety against XSS - Existing User Agents (geolocation: Madrid, accept language: en-US)

Conclusion:
Our findings indicate consistent implementation of safe Content Security Policies (CSP) across webpages for both malformed and non-existing user agents. However, variations were observed in webpages considered safe only for specific user agents across different geolocations. In Madrid, a constant number of webpages were safe, especially for Firefox, while in Brazilia, differences were found between desktop and mobile browsers, particularly with Chrome.

Website-Level:

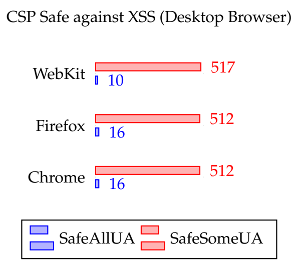

Figure 5.82: Desktop Browser

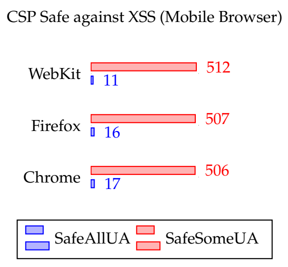

Figure 5.83: Mobile Browser

Figure 5.84: CSP Safety against XSS - Existing User Agents (geolocation: Brazilia, accept language: en-US

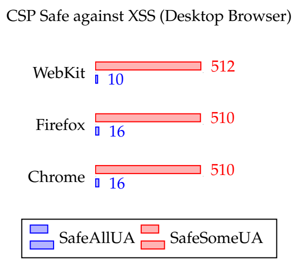

Figure 5.85: Desktop Browser

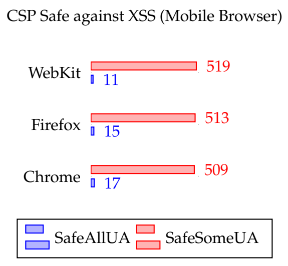

Figure 5.86: Mobile Browser

Figure 5.87: CSP Safety against XSS - Existing User Agents (geolocation: Hyderabad, accept language: en-US

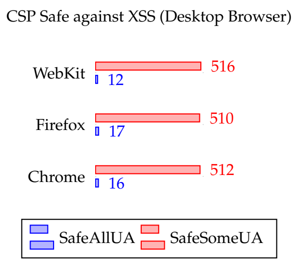

Figure 5.88: Desktop Browser

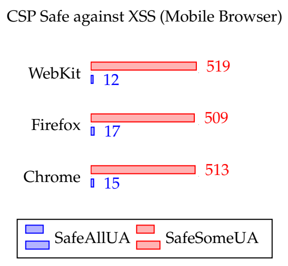

Figure 5.89: Mobile Browser

Figure 5.90: CSP Safety against XSS - Existing User Agents (geolocation: Madrid, accept language: en-US

Conclusion:
We found that the number of domains implementing safe Content Security Policies (CSP) remains steady, similar to what we observed with both malformed and non-existing user agents. This consistency highlights the persistent efforts of websites to prioritize web security and protect their users from potential threats.
A particularly interesting observation emerged when we analyzed the data for the geolocations of Madrid and Brazilia. We noticed that the number of homepages and subpages that deploy CSP remained constant between desktop and mobile browsers, specifically when using Firefox. This suggests that website administrators are maintaining a uniform and reliable approach to security measures, ensuring a secure browsing experience for users across different devices in these regions.

In the geolocation of Hyderabad, we made a remarkable discovery. The number of these main pages that implement safe CSP along with a subpage was exactly the same for both Chrome and Firefox when accessed through a desktop browser. This finding emphasizes the importance website administrators place on security, regardless of the specific browser being used, ensuring robust web security practices for visitors to their sites.

5.2.2 Impact of accept language on CSP Safety against XSS

In this section; We want to demonstrate whether different types of accept language (malformed, unsupported, and supported) can affect the deployment of CSP.

5.2.2.1 Malformed accept language

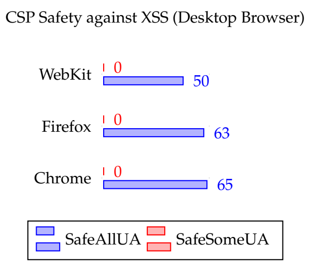

Figure 5.91: Desktop Browser

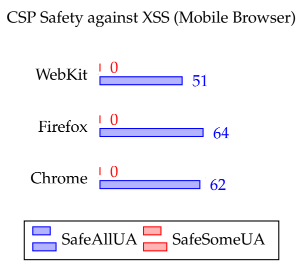

Figure 5.92: Mobile Browser

Figure 5.93: CSP Safety against XSS - Malformed User Agents (geolocation: Brazilia, accept language: ab-YZ

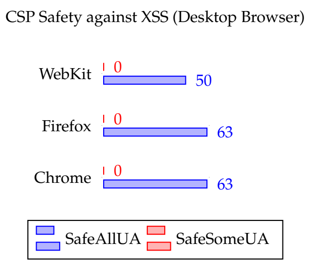

Figure 5.94: Desktop Browser

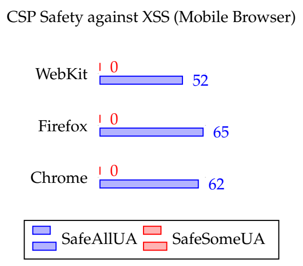

Figure 5.95: Mobile Browser

Figure 5.96: CSP Safety against XSS - Non-Existing User Agents (geolocation: Hyderabad, accept language: ab-YZ

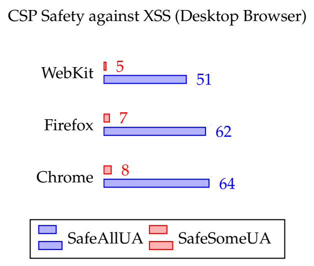

Figure 5.97: Desktop Browser

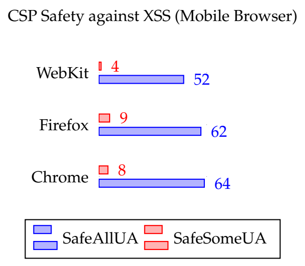

Figure 5.98: Mobile Browser

Figure 5.99: CSP Safety against XSS - Existing User Agents (geolocation: Madrid, accept language: ab-YZ)

5.2.2.2 Unsupported accept language

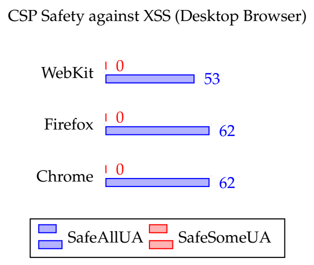

Figure 5.100: Desktop Browser

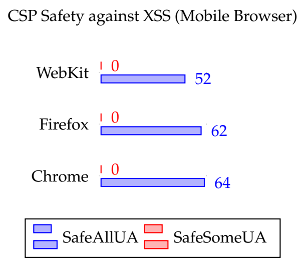

Figure 5.101: Mobile Browser

Figure 5.102: CSP Safety against XSS - Malformed User Agents (geolocation: Brazilia, accept language: ab-YZ

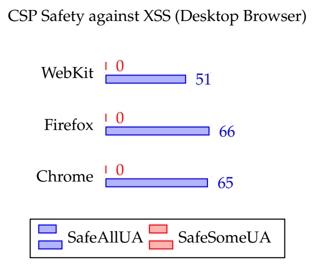

Figure 5.103: Desktop Browser

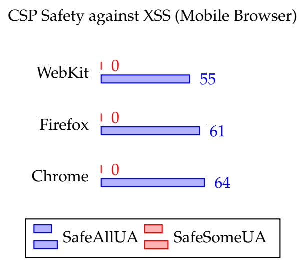

Figure 5.104: Mobile Browser

Figure 5.105: CSP Safety against XSS - Non-Existing User Agents (geolocation: Hyderabad, accept language: ab-YZ

Conclusion:
Our findings revealed consistent and reassuring results, mirroring our previous findings. The number of webpages implementing secure Content Security Policies (CSP) remained steady across various scenarios we explored. When we investigated the use of Chrome and Firefox on both desktop and mobile browsers, along with different user agents (malformed, non-existing, and existing), we observed that approximately 60 to 64 webpages had taken the proactive step of adopting a secure CSP. This observation reflects a strong commitment among websites to prioritize web security, ensuring a safe and protected browsing experience for users, regardless of the specific browsing platform or user agent used. While the number of webpages with a secure CSP was slightly lower for

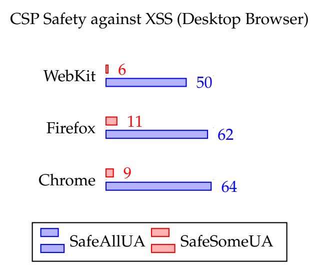

Figure 5.106: Desktop Browser

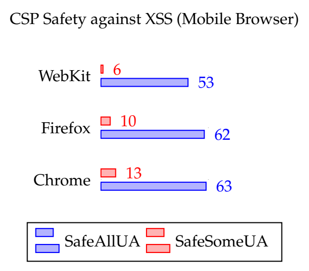

Figure 5.107: Mobile Browser

Figure 5.108: CSP Safety against XSS - Existing User Agents (geolocation: Madrid, accept language: ab-YZ)

Webkit-based browsers, ranging from 50 to 55, it still demonstrated a substantial portion of websites actively implementing security measures to safeguard against potential security threats.

5.3 Nonce Duplication

Our analysis involved examining the nonce values of each homepage and its corresponding subpages. To our surprise, during the first scenario **(4.3)**, we came across an unexpected finding. We discovered that the nonce value used by the homepage of `amocrm.ru` was also reused on one of its subpages, more specifically, on the webpage with the URL: `https://www.amocrm.ru/confidence`. This raised intriguing questions about how nonce values were managed on this particular webpage. To further explore the nonce behavior, we continued navigating through various webpages. For the third scenario **(4.3)**, we collected the URLs of each webpage along with their corresponding nonce values. We then revisited these websites to observe whether the previous nonce values would be reused for subsequent requests. Unexpectedly, we found that the only webpage that reused the same nonce value for the second request was amocrm.ru itself. This particular observation sparked curiosity about the nonce handling practices employed by this website. Additionally, we implemented an algorithm to check for any instances where different homepages shared the same nonce value. Remarkably, our analysis demonstrated that no homepage shared the same nonce value with any other homepages. This reassuring result provided evidence that nonce values were indeed unique for each individual homepage.

6 Discussion and Limitations

In our work, we want to highlight certain limitations that shaped the scope and depth of our work. These constraints are significant to consider when interpreting our findings and understanding the full extent of our investigation.

6.1 Limited Number of User Agents:

Since our work focuses on three types of User Agents, existing, non-existing, and malformed, the number of User Agents is limited due to the short time of work, and it could have been extended to a large set of User Agents.

6.2 Restricted Usage of various supported Accept Language:

The investigation considered three types of accepted language and focused on the effect of types of them unsupported and malformed accept language on the deployment and safety of Content Security Policy, and hence this work did not compass all potential supported accepted language.

6.3 Limited Types of Browsers:

This work focuses on using the widely used browsers: Chrome, Firefox, and Webkit, and overlooks other types of browsers, such as Opera, that also have a different impact on implementing CSP.

6.4 Limited Dataset:

Due to time constraints and limited time to execute experiments with many websites, the number of homepages and subpages has been minimized to 1992 homepages and 1521 subpages; undoubtedly, a large number of websites would enhance the results and the analysis.

6.5 Focusing on CSP from response headers:

Since CSPs from response headers are commonly used in practice [3], the work ignored the CSPs defined in meta tags.

6.6 Evaluation of Other Security Headers:

In addition to Content Security Policy (CSP), our framework cover a valuable analysis of other critical response headers: `'set-cookie'`, `'strict-transport-security'`, and `'x-frame-options'`. These headers play an essential role in protecting against various web attacks.

- `'set-cookie' :`
  This header is essential for protecting against certain web attacks, such as session *hijacking* and *stealing cookies*. It allows the developer to implement secure practices for handling cookies, making it tough to steal crucial information by attackers through performing *XSS* and *Cross-Site Request Forgery (CSRF)* attacks.

- `'strict-transport-security' (HSTS):`
  This header helps to provide a secure and encrypted connection (HTTPS) between the browser of the client and the web server. The enforcement of HTTPS mitigates *SSL stripping* attacks; By preventing the downgrading from HTTPS connections to HTTP connections.

- `'x-frame-options' :`
  This header plays a crucial role in protecting against *Clickjacking* attacks by ensuring that the web application cannot be maliciously embedded within an invisible frame or iframe on another website.

Our framework's evaluation is strengthened by incorporating these response headers by examining them **(Sections 8.1, 8.2, 8.3, and 8.4)** alongside CSP; we can identify potential vulnerabilities and fortify web applications against various cyber attacks.

7 Related Work

In this section, we take a closer look at related works of other researchers. We analyzed their methodologies and findings to have a priceless insights for our work. By comparing our work to other existing related works, we could specify similarities and differences, By comparing our work to existing research, we uncover similarities and differences, in order to gain a deeper understanding of our work's distinctive contributions.

7.1 Addressing Inconsistencies in Web Security and Policy Enforcement

7.1.1 The Security Lottery: Measuring Client-Side Web Security Inconsistencies [11]

This paper discusses the client-side security policies in modern web browsers, it takes a closer look at the consequences of the inconsistencies in the enforcement of client-side security policies and it shows how a security lottery can be introduced by these inconsistencies.
In general, the attackers aim primarily at performing attacks on web applications, and the protection against these attacks will be provided by client-side web security. HTTPS response headers such as Content Security Policy and HTTP Strict Transport Security are the needed security headers to provide the required protection against web attacks; in order to have optimal protection, these security headers must not be inconsistently for all different client characteristics.
The paper illustrates the factors that can affect web application security:

- User-Agent: This is one of the essential headers in HTTP request; it describes the device or the technology that the client use to access the website user agent contains many details like device type, version number, OS (operating system) numbers, the operating systems and the browser that has been used. Using an unprotected user agent will lead to a specific type of attack; for example, the Firefox desktop browser is safer than the Firefox mobile browser. Based on the user agent header websites can set up different configurations; this is called User-Agent sniffing, which leaves the user-agent unprotected and enables attacks.

- Vantage Point: Accessing the website from different geographical locations, which can affect the dynamically loaded advertisements and thus require a different server-side configuration of CSP. Moreover, security differences can occur based on the different responses through Onion network (For browsers).

- Client Configuration: Configuration settings can affect the user's Interaction with the website (e.g., the language of the user), this can lead to different security configurations.

Both our work and this paper discussed enhancing web application security through response headers, with a common focus on client-side security policies in modern web browsers.

- **Similarities between my work and this paper:**
  My work and the paper highlighted the essential roles of specific response headers, such as CSP and HSTS, in protecting against various types of attacks. They both recognize the crucial role of User-Agent headers and it implication on web application security. These use agent headers provide information about the user's browser and device's details, and depending on the details provided, different security configurations can be applied to ensure optimal protection; Furthermore, these works considering other factors that have effect on web application security, such as the geographical location of the user and the specific configurations of their client device.
- **Differences between my work and this paper:**
  - The paper go deeply into the results of inconsistent enforcement of client-side security policies and introduces the concept of a "security lottery". On the other hand, my work provides an in-depth analysis of CSP's importance in web development and considers more types of user agents, such as malformed and non-existing user agents, and malformed and unsupported accepted languages.
  - My work acknowledges specific limitations, including a limited dataset of URLs and a limited number of accepted languages and browsers. The paper does not mention these constraints.
  - Our work takes a more comprehensive approach by considering both homepages and subpages in our analysis. While the paper focuses only on homepages, we acknowledge the importance of examining subpages as well to gain a more thorough understanding of web application security.

7.1.2 Who's Hosting the Block Party? [15]

This paper discusses CSP implementation and Subresource Integrity (SRI) in order to promote web security and restrict third parties involvement, it illustrates an experiment on the top 10.000 websites to observe the behavior of different parties, disclosing the hindrances caused by third parties to perform security measures, the paper illustrates how third parties force first parties to include the entire domain in their security policies through the new hosts that have been introduced by third parties.
The paper clarified that CSP helps the vendors of the websites to specify the list of sources of the content that are allowed to be loaded by the website, and hence this mechanism plays the role of preventing cross-site scripting (XSS) attacks or any other attack that aim to load untrusted sources on the website; on the other hand, SRI helps the vendors of the websites to ensure the integrity of the content that is fetched from third-party sources.
Furthermore, the paper demonstrates the concept of the same party that describes the relation between the first party and its subdomains, it also illustrates that third parties involvement led to the requirement of the usage of the following directive values unsafe-inline and unsafe-eval, which deduct the web security; additionally, third parties provide code that made the hist-based CSP complicated.
To tackle these challenges, the paper elucidates multiple solutions for the first-party sites that face limitations in fully deploying CSP and SRI due to the lack of cooperation from third parties; one of these solutions is to provide a fallback policy that allows for loading

third-party scripts over HTTPS by providing the following directive values unsafe-inline and unsafe-eval to the fallback policy; another solution is also presented in the paper through the adoption of a hybrid policy that approves unsafe-inline and unsafe-eval for loading third-party scripts from trusted domains.

7.1.3 Uncovering HTTP Header Inconsistencies and the Impact on Desktop/Mobile Websites [6]

This paper presents the security implications coming from the inconsistency of the implementations between desktop and mobile versions of websites; the paper demonstrates some attack scenarios when the attacker takes advantage of these inconsistencies to perform certain types of attacks, the impact of these inconsistencies has been illustrated in this paper, by providing real-world studies using the most visited websites, the paper highlights the vulnerabilities caused by these inconsistencies and avouch the need to enhance consistency in the implementations of websites.

7.2 Analyzing and testing Content Security Policy (CSP) implementation.

7.2.1 DiffCSP: Finding Browser Bugs in Content Security Policy Enforcement through Differential Testing [17]

This paper introduces the framework DiffCSP, which is a framework that has been designed to identify and address security vulnerabilities in commonly used web browsers in respect of the implementation of Content Security Policy (CSP) that explicitly regulates the execution of JavaScript (JS).
Furthermore, this research avouches the significance of CSP regarding *XXS* mitigation, it also avouches the obligation of browser vendors to ensure the proper compliance with CSP standard during JS execution.
The paper introduced DiffCSP as the solution for this issue; DiffCSP generates inputs for testing and observes whether any unexpected behaviors can be found during the enforcement of CSP; these inputs are generated to trigger inherent browser bugs that lead to erroneous enforcement of CSP.

7.3 Deployment and Analysis of Content Security Policy (CSP)

7.3.1 Coverage and Secure Use Analysis of Content Security Policies via Clustering [8]

The paper demonstrates an approach for assessing the security effectiveness of the deployed Content Security Policy (CSP) by measuring the coverage of the directives and secure use, Content Security Policy allow the developers to define the sources of the content that are allowed to be loaded on the website through this definition CSP can prevent certain types of attack such as cross-site scripting (XSS).
Furthermore, the paper put forward a methodology that includes generating a policy feature based on the latest CSP Level 3 specification, a clustering algorithm has been developed and used to categorize CSPs; through a study involving 100.000 websites, the CSPs, which have been deployed on 13.317 homepages, have been categorized to 16 clusters,

each characterized by its unique attributes; the security levels of CSPs that included in each cluster are then analyzed to promote proper CSP deployment.

7.3.2 On the Content Security Policy Violations due to the Same-Origin Policy [13]

The paper delves into the significance of establishing a robust and uniform Content Security Policy (CSP) to uphold the security of web pages. CSP serves as a security feature that permits developers to define trusted domain sources from which content, including JavaScript, can be fetched. Its primary objective is to mitigate cross-site scripting attacks (XSS), data leak attacks, and other forms of security breaches.
A fundamental issue with CSP is highlighted in the paper, namely the presence of multiple or inconsistent CSPs across different pages within the same origin, which can potentially give rise to security vulnerabilities. The paper explains how CSP violations can occur when a page lacks CSP or possesses a weak and insecure CSP, thereby compromising the security of other pages within the same origin.
To assess the frequency of such violations, the paper conducts a large-scale analysis encompassing over 1 million pages from 10,000 of the top Alexa sites. The findings indicate that 5.29% of the examined sites contain pages with CSPs, a higher proportion compared to previous studies focused solely on home pages (which had a 2% occurrence rate). The paper also identifies three scenarios where CSP violations may transpire, elucidating how same-origin pages can bypass page-specific policies, including CSP.

7.3.3 Content Security Problems? Evaluating the Effectiveness of Content Security Policy in the Wild [2]

Content Security Policy (CSP) is a security mechanism that help the developers to define a set of content sources that are allowed to be loaded on the website, it helps by preventing the cross-site scripting (XXS) attack and any attack that aims to inject malicious scripts, this protection is determined by the enforced rules that restrict the execution of untrusted scripts; moreover, the degree to which CSP effectively enhances website security is contingent upon different factors, including the level of browser support, website adoption, configuration, and regular maintenance.
This paper demonstrates an analysis of CSP and its effect on website security through using the top 1 million Alexa websites to extract and collect CSP headers and policy violations, the paper observed websites that deploy or not deploy CSP and how policies changed over time. Based on the results in this paper, 92.4% of the website deploy a Content Security Policy that is weak in respect of protecting against attacks that perform code injection; furthermore, most of the deployed CSPs have errors and weaknesses that lead to defective security.

7.4 Effectiveness of Web Application Security and Policy Implementation

7.4.1 12 Angry Developers: A Qualitative Study on Developers' Struggles with CSP [12]

This paper introduces a large explanation of Content Security Policy (CSP) and their functionality, it demonstrates how CSP allow developers to specify the resources to load trusted content on the website such as images, stylesheets and more contents; moreover, this specification mitigates cross-site scripting (XSS) attacks and any attack that execute malicious scripts.
The paper clarified the challenges that the developer face during the process of CSP implementation such as considering the correctness of the syntax and semantics of CSP directives; furthermore, the paper discussed the problem of balancing between functionality and security, which each web developer face.
To detect the challenges that the developer face during the development process, the paper demonstrates a study that involved 12 professional web developers in CSPs, each of whom has been interviewed to collect data from their experiences in implementing CSPs; Next the collected data from the interviews have been analyzed to specify the shared challenges and difficulties between the developers.

7.4.2 Complex Security Policy? A Longitudinal Analysis of Deployed Content Security Policies [10]

This paper introduces CSP and its primary role in web security, which is preventing the execution of malicious scripts by specifying the trusted content that are allowed to be loaded on the website; however, the paper demonstrates an extensive study of CSP using data from the Internet Archive for the period from 2012 to 2018, the paper shows how the evolution of CSP and its use cases has been documented, the study present that CSP has gradually transitioned from primarily enforcing content restrictions to encompassing other security objectives such as TLS enforcement and framing control. However, despite its potential benefits, wide-scale adoption of CSP remains limited.
The paper highlights several challenges associated with the adoption and implementation of CSP. These challenges include slow deployment, vulnerabilities allowing for trivial bypasses, the existence of insecure CSP variants, compatibility issues, and the overall complexity of CSP. The authors provide detailed insights into each challenge and illustrate how they can be exploited by malicious actors. For instance, they demonstrate how CSP can be circumvented through the use of expired or typo domains, and they identify insecure CSP variants that can be easily bypassed.
To enhance the adoption and effectiveness of CSP, the authors propose potential solutions. They suggest simplifying the implementation process for website operators, offering improved support and warnings for CSP-related issues. The authors emphasize the significance of third parties being explicit about their dependencies and their impact on CSP, suggesting the incorporation of this information into Integrated Development Environments (IDEs) to alert developers about potential CSP roadblocks.

7.5 Web Application Security: Measures and Attack Patterns

7.5.1 Can I Take Your Subdomain? Exploring Same-Site Attacks in the Modern Web [14]

The paper discusses the threat posed by related-domain attackers to the security of Content Security Policy (CSP), a client-side defense mechanism designed to mitigate content injection and other threats. Related-domain attackers are attackers who share the same registrable domain as the target web application, and they represent a significant security threat to CSP, as they can exploit the ability to create arbitrary subdomains and bypass the restrictions imposed by CSP.
The paper argues that related-domain attackers have become more relevant and realistic in recent years, as major browsers are changing their behavior so that cookies are only attached to same-site requests by default. This further differentiates related-domain attackers from web attackers and highlights the need for site operators to carefully vet and ensure that all subdomains included in their CSP whitelists adhere to the same security standards as the main domain.
The paper also discusses the importance of the "site"concept in web security, which has become more prominent over the years and goes beyond cookie confidentiality and integrity issues. The notion of site has become more and more important for web security over the years, going well beyond cookie confidentiality and integrity issues. For example, the Site Isolation mechanism of Chromium ensures that pages from different sites are always put into different processes, so as to offer better security guarantees even in the presence of bugs in the browser.

7.5.2 CSP Is Dead, Long Live CSP! On the Insecurity of Whitelists and the Future of Content Security Policy [16]

Content Security Policy (CSP) is a security mechanism that allows web developers to specify which sources of content are allowed to be loaded on their web pages. CSP is designed to mitigate a range of attacks, including cross-site scripting (XSS) and data injection attacks. However, despite its potential benefits, CSP has not been widely adopted by web developers, and its effectiveness in practice has been called into question.
The paper presents an assessment of the practical security benefits of adopting CSP in real-world applications, based on a large-scale empirical study. The study analyzes a corpus of approximately 106 billion pages, of which 3.9 billion are protected with CSP. It identifies a significant number of policies and finds that many of them are ineffective in mitigating XSS attacks due to policy misconfigurations and insecure whitelist entries.
The study proposes a change to how CSP is deployed, advocating for a nonce-based approach instead of whitelisting. It introduces a new feature called 'strict-dynamic' in the CSP3 specification, which is implemented in the Chromium browser. The benefits of this approach are discussed, and a case study of deploying a policy based on nonces and strict-dynamic in a popular web application is presented.

7.5.3 Assessing the Impact of Script Gadgets on CSP at Scale [9]

The paper explores the challenges of mitigating Cross-Site Scripting (XSS) attacks through the use of Content Security Policy (CSP) and the potential vulnerabilities posed by script gadgets.

The paper begins by discussing the importance of the web in today's society and the increasing threat of XSS vulnerabilities. These vulnerabilities allow an attacker to execute JS code within the context of a flawed website, essentially enabling the attacker's code to conduct any action the site's own JS could. This enables an attacker to steal a victim's credentials, leak sensitive information, or perform actions on behalf of a victim. Thus, XSS can cause severe damage, especially if present in security-critical applications.
To mitigate the threat of XSS, web developers can use CSP, which is a security standard that allows web developers to specify which sources of content are allowed to be loaded on a web page. However, CSP is not foolproof, and attackers can use script gadgets to bypass CSP and execute malicious code.
Script gadgets are small pieces of code that can be used to bypass CSP by exploiting the way that CSP policies are evaluated. Script gadgets can be used to load malicious scripts from otherwise trusted sources, such as Google Analytics or Facebook, and execute them on the target website. This can allow an attacker to bypass CSP and execute malicious code, even if the website has a CSP policy in place.
The paper analyzes the real-world deployment of CSP and discovered gadgets to show that 10% of otherwise secure real-world CSPs can be bypassed through script gadget sideloading.

7.6 HTTPS Security: Issues and Inconsistencies

7.6.1 Exploring HTTPS Security Inconsistencies: A Cross-Regional Perspective [1]

The paper explores the security aspects of HTTPS connections from a cross-regional perspective. To conduct the study, the researchers scan the top 250,000 most visited domains on the Internet using clients located in five different regions: North America, Europe, Asia, South America, and Africa. They employ a combination of automated and manual techniques to gather data from both the application and transport layers. The investigation focuses on three key security vectors: URLs security, security headers, and TLS security, examining specific security properties deemed significant.
The findings of the paper reveal inconsistencies in the HTTPS security guarantees provided by servers when responding to requests from different regions for the same domain. The paper establish a correlation between HTTPS security inconsistencies and factors such as URL diversity, IP diversity, and the presence of redirections. The paper also introduces a novel attack scenario called the "region confusionättack, where an attacker exploits HTTPS security inconsistencies to redirect a client's request from a region with robust security to another region with weaker security, thereby exploiting vulnerabilities in the weaker region.

7.7 Mitigating Inconsistencies in Web Security Policies

7.7.1 Reining in the Web's Inconsistencies with Site Policy [3]

The paper focuses on addressing the issue of inconsistencies in web application security mechanisms that can lead to security vulnerabilities. It highlights the current state of client-side security measures, emphasizing their fine-grained nature and susceptibility to inconsistencies. These inconsistencies pose a significant risk as they can potentially enable

attackers to bypass security measures and gain unauthorized access to sensitive data.
To demonstrate the prevalence and impact of inconsistencies, the paper formalize the concept across three prominent security mechanisms: cookie security attributes, Content Security Policy (CSP), and HTTP Strict Transport Security (HSTS).
The analysis reveals the widespread nature of inconsistencies and their potential for significant security implications.
The paper also examines the effectiveness of the recent Origin Policy proposal, which aims to tackle the problem of inconsistencies.
To overcome these challenges, the paper propose a novel security mechanism called Site Policy (SP). SP is specifically designed to address the shortcomings of Origin Policy and offers a solution to rectify the defined inconsistencies observed in real-world scenarios.
The paper emphasize the need for explicit identification of any potential inconsistencies within the SP manifest, enabling researchers to measure the effectiveness of SP and allowing operators to understand the security guarantees it offers in the worst-case scenarios.

8 Appendix

8.1 *Assessing safety level against stealing the cookies (XXS attacks)*

The algorithm search for the header ´set-cookie′:

1. **PHASE.1**

Algorithm 4 Searching for ′set-cookie′ Header

```
 1: Input: HTTP response headers
 2: PHASE.1
 3: if ′set-cookie′ header is present then
 4:     Execute Algorithm 5
 5: else
 6:     Return No protection against stealing cookies and against CSRF attacks
 7: end if
 8: Algorithm 5 {Perform actions for ′set-cookie′}
```

2. **PHASE.2**

Algorithm 5 Searching for specific values in ′set-cookie′ Header

```
 1: Input: set-cookie
 2: PHASE.2
 3: if ′set-cookie′ header contains ′HttpOnly′ then
 4:     Return Secure against stealing cookies
 5: else if ′set-cookie′ header contains ′Strict′ then
 6:     if ′Strict′ is present then
 7:         Return Secure against CSRF attacks (safety level 2)
 8:     else if ′Lax′ is present then
 9:         Return Secure against CSRF attacks (safety level 1)
10:     end if
11: else
12:     Return No protection against stealing cookies and against CSRF attacks
13: end if
```

Note: The higher the level of safety, the better the security.

8.2 *Assessing safety against hijacking*

In this case, a website can be secure against hijacking if 'set-cookie' contains both 'Secure' and 'HttpOnly'.

8.3 *Assessing safety against clickjacking*

1. **PHASE.1**

Algorithm 6 Checking for Clickjacking Protection

```
Input: HTTP response headers
PHASE.1
if 'content-security-policy' header is present and contains 'frame-ancestors' then
    Execute Algorithm 7
else if 'x-frame-options' header is present then
    Execute Algorithm 8
else
    Return Not secure against clickjacking
end if
Algorithm 7 {Perform actions for 'frame-ancestors'}
Algorithm 8 {Perform actions for 'x-frame-options'}
```

2. **PHASE.2**

Algorithm 7 Checking for Clickjacking Protection frame-ancestors

```
Input: Content Security Policy
PHASE.2
if 'frame-ancestors' contains only 'none' then
    Return Secure against clickjacking (safety level 3)
else if 'frame-ancestors' contains only 'self' then
    Return Secure against clickjacking (safety level 2)
else if 'frame-ancestors' contains other values then
    Return Secure against clickjacking (safety level 1)
else
    Return Not secure against clickjacking (safety level 0)
end if
```

3. **PHASE.3**

Algorithm 8 Checking for Clickjacking Protection x-frame-options

```
Input: x-frame-options
PHASE.3
if 'x-frame-options' contains 'DENY' then
    Return Secure against clickjacking (safety level 3)
else if 'x-frame-options' contains 'SAMEORIGIN' then
    Return Secure against clickjacking (safety level 2)
else
    Return Not secure against clickjacking (safety level 0)
end if
```

8.4 *Assessing safety against ssl stripping*

1. **PHASE.1**

Algorithm 9 Checking for SSL Stripping Protection

```
Input: HTTP response headers
PHASE.1
if 'strict-transport-security' header is present then
    Execute Algorithm 10
else
    Return Not secure against SSL stripping
end if
Algorithm 10 {Perform additional actions for strict-transport-security}
```

2. **PHASE.2**

Algorithm 10 Checking for SSL Stripping Protection

```
Input: strict-transport-security
PHASE.2
if 'preload' is present then
    Return Secure against SSL stripping
else if 'includeSubDomains' is present then
    Return Secure against SSL stripping
else if 'max-age' is present then
    Return Secure against SSL stripping
else
    Return Not secure against SSL stripping
end if
```

Literatur

[1] Eman Salem Alashwali, Pawel Szalachowski und Andrew Martin. „Exploring HTTPS security inconsistencies: A cross-regional perspective". In: *Comput. Secur.* 97 (2020), S. 101975. DOI: `10.1016/j.cose.2020.101975`. URL: `https://doi.org/10.1016/j.cose.2020.101975`.

[2] Stefano Calzavara, Alvise Rabitti und Michele Bugliesi. „Content Security Problems?: Evaluating the Effectiveness of Content Security Policy in the Wild". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*. Hrsg. von Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers und Shai Halevi. ACM, 2016, S. 1365–1375. DOI: `10.1145/2976749.2978338`. URL: `https://doi.org/10.1145/2976749.2978338`.

[3] Stefano Calzavara, Tobias Urban, Dennis Tatang, Marius Steffens und Ben Stock. „Reining in the Web's Inconsistencies with Site Policy". In: *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*. The Internet Society, 2021. URL: `https://www.ndss-symposium.org/ndss-paper/reining-in-the-webs-inconsistencies-with-site-policy/`.

[4] Eran Kinsbruner und Gleb Bahmutov. *A Frontend Web Developer's Guide to Testing(Chapter 3: Top Web Test Automation Frameworks)*. `https://learning.oreilly.com/library/view/a-frontend-web/9781803238319/`.

[5] *Mdn Web Docs*. `https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/frame-ancestors`.

[6] Abner Mendoza, Phakpoom Chinprutthiwong und Guofei Gu. „Uncovering HTTP Header Inconsistencies and the Impact on Desktop/Mobile Websites". In: *Proceedings of the 2018 World Wide Web Conference on World Wide Web, WWW 2018, Lyon, France, April 23-27, 2018*. Hrsg. von Pierre-Antoine Champin, Fabien Gandon, Mounia Lalmas und Panagiotis G. Ipeirotis. ACM, 2018, S. 247–256. DOI: `10.1145/3178876.3186091`. URL: `https://doi.org/10.1145/3178876.3186091`.

[7] Victor Le Pochat, Tom van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczynski und Wouter Joosen. „Tranco: A Research-Oriented Top Sites Ranking Hardened Against Manipulation". In: *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019. URL: `https://www.ndss-symposium.org/ndss-paper/tranco-a-research-oriented-top-sites-ranking-hardened-against-manipulation/`.

[8] Mengxia Ren und Chuan Yue. *Coverage and Secure Use Analysis of Content Security Policies via Clustering*.

[9] Sebastian Roth, Michael Backes und Ben Stock. „Assessing the Impact of Script Gadgets on CSP at Scale". In: *ASIA CCS '20: The 15th ACM Asia Conference on Computer and Communications Security, Taipei, Taiwan, October 5-9, 2020*. Hrsg. von Hung-Min Sun, Shiuh-Pyng Shieh, Guofei Gu und Giuseppe Ateniese. ACM, 2020, S. 420–431. DOI: `10.1145/3320269.3372201`. URL: `https://doi.org/10.1145/3320269.3372201`.

[10] Sebastian Roth, Timothy Barron, Stefano Calzavara, Nick Nikiforakis und Ben Stock. „Complex Security Policy? A Longitudinal Analysis of Deployed Content Security Policies“. In: *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society, 2020. URL: `https://www.ndss-symposium.org/ndss-paper/complex-security-policy-a-longitudinal-analysis-of-deployed-content-security-policies/`.

[11] Sebastian Roth, Stefano Calzavara, Moritz Wilhelm, Alvise Rabitti und Ben Stock. „The Security Lottery: Measuring Client-Side Web Security Inconsistencies“. In: *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*. Hrsg. von Kevin R. B. Butler und Kurt Thomas. USENIX Association, 2022, S. 2047–2064. URL: `https://www.usenix.org/conference/usenixsecurity22/presentation/roth`.

[12] Sebastian Roth, Lea Gröber, Michael Backes, Katharina Krombholz und Ben Stock. „12 Angry Developers - A Qualitative Study on Developers’ Struggles with CSP“. In: *CCS ’21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*. Hrsg. von Yongdae Kim, Jong Kim, Giovanni Vigna und Elaine Shi. ACM, 2021, S. 3085–3103. DOI: `10.1145/3460120.3484780`. URL: `https://doi.org/10.1145/3460120.3484780`.

[13] Dolière Francis Somé, Nataliia Bielova und Tamara Rezk. „On the Content Security Policy Violations due to the Same-Origin Policy“. In: *Proceedings of the 26th International Conference on World Wide Web, WWW 2017, Perth, Australia, April 3-7, 2017*. Hrsg. von Rick Barrett, Rick Cummings, Eugene Agichtein und Evgeniy Gabrilovich. ACM, 2017, S. 877–886. DOI: `10.1145/3038912.3052634`. URL: `https://doi.org/10.1145/3038912.3052634`.

[14] Marco Squarcina, Mauro Tempesta, Lorenzo Veronese, Stefano Calzavara und Matteo Maffei. „Can I Take Your Subdomain? Exploring Same-Site Attacks in the Modern Web“. In: *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*. Hrsg. von Michael Bailey und Rachel Greenstadt. USENIX Association, 2021, S. 2917–2934. URL: `https://www.usenix.org/conference/usenixsecurity21/presentation/squarcina`.

[15] Marius Steffens, Marius Musch, Martin Johns und Ben Stock. „Who’s Hosting the Block Party? Studying Third-Party Blockage of CSP and SRI“. In: *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*. The Internet Society, 2021. URL: `https://www.ndss-symposium.org/ndss-paper/whos-hosting-the-block-party-studying-third-party-blockage-of-csp-and-sri/`.

[16] Lukas Weichselbaum, Michele Spagnuolo, Sebastian Lekies und Artur 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, Vienna, Austria, October 24-28, 2016*. Hrsg. von Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers und Shai Halevi. ACM, 2016, S. 1376–1387. DOI: `10.1145/2976749.2978363`. URL: `https://doi.org/10.1145/2976749.2978363`.

[17] Seongil Wi, Trung Tin Nguyen, Jihwan Kim, Ben Stock und Sooel Son. „DiffCSP: Finding Browser Bugs in Content Security Policy Enforcement through Differential Testing“. In: *30th Annual Network and Distributed System Security Symposium, NDSS 2023, San Diego, California, USA, February 27 - March 3, 2023*. The Internet Society, 2023. URL: `https://www.ndss-symposium.org/ndss-paper/diffcsp-`

finding-browser-bugs-in-content-security-policy-enforcement-through-differential-testing/.