Ca' Foscari
University
of Venice

Master's Degree programme
in Computer Science

Final Thesis

Automated verification of the Mixed Content policy by using Web Platform Tests

Supervisor
Prof. Stefano Calzavara

Graduand
Valentino Dalla Valle
Matriculation Number 874210

Academic Year
2022/2023

To my best friend Andrea (Miche)

Abstract

As the reliance on web applications for critical tasks such as banking and shopping grows, ensuring user data and privacy protection becomes imperative. This thesis delves into the intricacies of web application security, emphasizing the pivotal role of browser client-side security mechanisms, particularly the Mixed Content policy. Managed by the World Wide Web Consortium (W3C), this policy addresses vulnerabilities introduced when HTTPS-loaded webpages request insecure resources, which can lead to exploitable attacks. The study proposes an automated methodology to verify the Mixed Content policy's implementation in web browsers using the Web Platform Test suite. The results show that the policy's implementation is not always compliant with the specification. In particular, exploitable vulnerabilities were found in two major web browsers. The vulnerabilities have been disclosed to the vendors and have been fixed, and one CVE was assigned with a base score of 8.8. To understand the presence of mixed content in the wild, a large-scale analysis of the top 100K websites was conducted, comparing the data obtained with information from 2015. The results show that despite the community effort to reduce the presence of mixed content, the issue is still present in a non-negligible number of websites.

Acknowledgements

Many thanks are due to my supervisor, Prof. Stefano Calzavara. His support and guidance have been fundamental for the completion of this work. The opportunities he provided have been invaluable for my personal and professional growth.

A special thanks also goes to the members of the Security and Privacy research group at TU Wien. The feedback, discussions, and ideas they provided made this work possible. In particular, I would like to thank Lorenzo Veronese, Pedro Bernardo, Marco Squarcina, and Matteo Maffei, whom I worked with during my time at TU Wien. The time spent with them helped me to understand the opportunities and challenges of research, and it has been a great source of inspiration for my future career.

Words are not enough to tank my family, who supported me during my studies, as well as my girlfriend Gaia, a constant source of love and encouragement. My mom Marta and my sister Veronica have always been there for me, and I am so grateful to them.

Finally, I would like to thank my best friend Andrea, who has been with me at the start of this journey. His support and friendship have been fundamental for so many years and his concrete help during the first challenging years of university has been invaluable. He suddenly passed away in November 2020. Here I would like to remember him as a hard worker, an excellent student, and a wonderful person. A special thought goes to his family: Luigina, Giovanni, Irene, and Chiara

Contents

List of Figures

List of Tables

List of source codes

1 Introduction

The work I present in this thesis is based on my internship at the Research Unit Security and Privacy of TU Wien from April to July 2023. There, I worked on a project that aims at automating the verification of security policy implementations in web browsers. The project is ideated by Lorenzo Veronese, Pedro Bernardo, Marco Squarcina, and Matteo Maffei from TU Wien, together with Stefano Calzavara from Ca' Foscari University and Pedro Adão from the University of Lisbon.

My contribution to this project was the study and modeling of the Mixed Content policy.

As users increasingly rely on web applications for sensitive tasks like banking, shopping, and communication, it becomes vital to ensure that their data and privacy are protected from potential breaches. The security of web applications is a complex problem: it involves multiple layers of software and a shared responsibility between browser vendors and web developers.

Browser client-side security mechanisms are fundamental as they are the first line of defense against attacks and can mitigate the effects of vulnerabilities in web applications.

Security mechanisms are regulated via specifications developed by various standardization bodies. These specifications define platform-independent rules that browsers must follow.

However, the implementation of these specifications is not always straightforward as often may require changes to existing browser components which were not developed with such integration in mind.

One of the most relevant client-side security mechanisms is the Mixed Content policy, which regulates the inclusion of insecure resources in HTTPS-delivered web pages. The specification is managed by the World Wide Web Consortium (W3C) and has gone through multiple revisions in the last few years.

When a webpage loaded over HTTPS requests resources over an insecure connection, it introduces vulnerabilities that can be exploited by attackers.

These vulnerabilities can lead to man-in-the-middle attacks, where attackers can intercept, alter, or inject malicious content into the insecure requests.

The Mixed Content policy is designed to mitigate these attacks by blocking the insecure requests. Its correct implementation is therefore fundamental to ensure the security of users.

In this thesis is proposed a methodology to automate the verification of the Mixed Content policy in web browsers. To do so, we will leverage the Web Platform Test (WPT) suite, a collection of tests that are maintained by browser vendors, specification authors, and web developers.

After having analyzed the implementation of the browser security mechanism, we will move on to study the presence of mixed content in web applications. Our attention will be focused on the top 100K websites, trying to compare up-to-date information with data collected in 2015. This will allow us to understand the effectiveness of the Mixed Content security mechanisms as well as evaluate the web developers' efforts in securing their applications.

The thesis is structured in the following parts:

- **Chapter 2**: provides the background knowledge needed to understand the thesis. It starts with a brief introduction to the Web Platform, its architecture, and its main technologies. The focus is on web standards and respective browser implementations. A brief look at the Same Origin Policy will follow, in order to understand its role in securing the web and understand its limitations.

  Then the HTTPS protocol is introduced, with a focus on the security properties it provides and the risks that could arise with a wrong deployment.

  Finally, an overview of mixed content and the W3C specification that regulates its inclusion in web pages. A brief look at an example of the implementation of mixed content filtering is considered, to better understand the current state of the art.

- **Chapter 3**: provides the methodology used to automate the verification of the Mixed Content policy in web browsers. The Web Platform Test project is briefly presented, with an overview of the test suite structure and the role it has in the web ecosystem.

  A framework that leverages WPT to automate the verification of security policies is then presented. An overview of the pipeline used by the framework is provided, with a focus on the design choices and the implementation details required to perform a large-scale formal analysis of data collected from the execution of WPT tests in web browsers.

  Finally, the Mixed Content specification is studied in detail, to extract a model of the policy that can be used within the framework. The challenges encountered during this process are presented, with a focus on the choices made to obtain a platform-independent, efficient, and precise model that can be deployed on a large-scale analysis. Each step of the model extraction is presented with a direct reference to the corresponding component of the specification.

- **Chapter 4**: the chapter is divided into two parts. First, the results of the analysis on the three major web browsers (Chromium, Firefox, and Safari) are presented. The security vulnerabilities found are discussed, with a focus on the impact they have on the security of users. A brief look at the responsible disclosure process is provided, looking at the reports that were sent to browser vendors. The fix implemented by the vendors is discussed, with a look at the corrections made to the codebase.

  After having analyzed the deployment of mixed content policy in web browsers, a study on the presence of mixed content on the web is conducted. The data collected from crawling the top 100K websites is presented, with an analysis of mixed content at different levels of granularity (per category and request type). The analysis is performed in two steps: the top 1000 websites are studied first, then the whole dataset is considered.

Finally, a comparison of the results with data from a study conducted in 2015 will provide the basis for a discussion on the effectiveness of the different filtering approaches proposed in the last years.

Numerous code snippets are included in the thesis, and various types of programming languages are considered. To easily identify them, the background color of the snippet encodes the language used. In the following, the languages used and the color encoding.

- **HTML** : webpages source code.
- **JavaScript** : webpages source code.
- **C++** : web browsers source code.
- **SMT-LIB** : implementation of web invariants.
- **Pseudocode** : algorithms defined in specifications.
- **Text** : generic text (e.g. vulnerabilities disclosure reports, commmit messages, etc.)

2 Background

2.1 The Web Platform

The term "web platform" refers to the collection of tools, languages, and protocols that enable the creation, operation, and experience of websites and web applications.

The web platform is continually evolving and most of its components are open and standardized, ensuring that web content can be accessed and experienced consistently across different devices and platforms.

Several organizations play pivotal roles in the creation, management, and evolution of web standards. These organizations collaborate with industry experts, browser vendors, and developers to ensure the web remains open, interoperable, and user-friendly.

Among these organizations, there are:

- **World Wide Web Consortium (W3C)**: Founded in 1994 by Tim Berners-Lee, develops open standards following a well-defined process which includes public drafts, feedback collection, candidate recommendations, and final recommendations.

- **Web Hypertext Application Technology Working Group (WHATWG)**: Founded in 2004 by individuals from Apple, Mozilla, and Opera. The primary focus is on HTML and related web technologies by maintaining "living standards": continuously updated document that don't have version numbers.

- **Internet Engineering Task Force (IETF)**: Founded in 1986, develops and promotes Internet standards, in particular the standards that comprise the Internet protocol suite (TCP/IP). The IETF's standardization process involves drafting documents called "Request for Comments" (RFCs). These documents are reviewed, discussed, and iterated upon by the community before becoming standards.

2.1.1 HTML

HTML is the standard markup language used to create the structural framework of web pages. The markup components of HTML are called tags and are usually expressed as a pair of opening and closing elements used to delimit the content of the tag. The content of a tag can be either text or other HTML tags, and the nesting of tags defines the structure of the page. Tags are enclosed in angle brackets, and the opening tag can contain attributes that provide additional information about the element. The HTML specification [41] defines a set of standard tags that can be used to create the structure of a web page. The `<html>` tag is the root element of an HTML page and is used to define the document type. The `<head>` tag is used to define the header of the page and contains metadata about the document, while the `<body>` tag is used to define the body of the page. Upon receiving an HTML document, the browser parses the document and renders the page accordingly.

DOM tree

The structure provided by the HTML tags is used by browsers to build the Document Object Model (DOM) tree. The DOM tree is a representation of the HTML document as a tree of nodes, where each node represents an HTML object. It defines the logical structure of the document, providing a reference for the methods that access and manipulate it. All the HTML components have a representation in the DOM tree: HTML elements are element nodes, HTML attributes are attribute nodes and the text contained in the HTML elements forms text nodes. The `<html>` tag is the root element of the DOM tree, and it is the parent of the `<head>` and `<body>` elements.

The DOM tree implementation is regulated by the WHATWG DOM Living standard [42] that proposes a platform-neutral model for the tree, as well as the interfaces that can be used to access and manipulate it.

Subresources

Within a single HTML document, it is possible to include references to other resources such as images, stylesheets, and scripts. As the browser parses the HTML document, it may encounter a subresource, when this happens, the requested content will be fetched by the browser via a new separate HTTP request. When the content is retrieved, the browser can use it to render the page. Depending on the type of resource requested, the browser can decide to fetch and execute the content immediately or to defer the request. As an example, if a subresource consists of a script requested via the `<script>` tag, the browser will halt the rendering of the page until the script is fetched and executed. However, attributes like `async` and `defer` can modify this behavior, allowing the browser to continue parsing the HTML while the script is being fetched or executed.

Once a subresource is fetched, it can be cached by the browser. This means that on subsequent visits or page loads, the browser can use the cached version instead of fetching it again from the server, speeding up the page load time.

For certain subresources, like images, browsers can opt for "lazy loading": that is loading the resource only when it's about to be displayed on the screen (e.g. when a user scrolls to it). Web developers can enable lazy loading of subresources by using the `loading="lazy"` attribute. Lazy loading can improve the initial page load time, especially for pages with a lot of images. Another benefit of lazy loading subresources is data and battery saving: in a recent blog article by the Chromium project, experiments showed a $\sim 10\%$ reduction in bytes downloaded per page with lazy loading enabled in Chromium [1].

Frames

The HTML tag `<iframe>`, short for "inline frame", defines a type of subresource used to allow an external webpage to be embedded within the current page. It consists of a window within a webpage that displays another separate web document. Inline frames are used for embedding content like advertisements, maps, videos, or even entire web pages within a parent page. The source of the content to be embedded is specified using the `src` attribute. An example of an inline frame is `<iframe src="https://www.example.com"></iframe>`. When the content is fetched, it will be rendered within the boundaries of the iframe on the parent page. This content is independent of the parent page, meaning it will have its own DOM tree and code execution context.

2.1.2 JavaScript as client-side language

JavaScript is a high-level prototype-based object-oriented programming language used on the web to add interactivity, dynamic behavior, and complex functionalities to websites. The language conforms with the ECMAScript specification [17] that ensures the interoperability of web pages across different browsers.

When JavaScript code is embedded in websites it is executed by the client's browser using an interpreter (usually referred to as JavaScript engine).

JavaScript code can be embedded in web pages in different ways, the most common being:

- **Inline scripts**: code can be embedded directly in the document using the `<script>` tag.
- **External scripts**: code can be included as a subresource via `<script src="">` to specify the URL of the script.
- **Event handlers**: code can be embedded in HTML objects to be executed in response to events triggered by the user or the browser, such as clicks, mouse movements, or keyboard input. An example of an event handler is `<button onclick="alert('Button clicked')">Click</button>`. When the button is clicked, the code specified in the `onclick` attribute is executed.

Manipulate the DOM tree via JavaScript

JavaScript, when used in the context of web browsers, provides the ability to interact with and manipulate the DOM tree of a document. This capability is fundamental to creating dynamic, interactive, and responsive web applications. Scripts embedded in the page can access and modify elements' attributes as well as add or remove nodes from the DOM tree.

Such functionalities are enabled by a series of methods and properties that are part of the WHATWG DOM living standard. As an example, the method `document.getElementById()` can be used to retrieve a reference to an element in the DOM tree.

The method `document.createElement('tagname')` can be used to create a new element node and `parent.appendChild(childNode)` can be used to append the element as a child of a node.

XMLHttpRequest

The XMLHttpRequest (XHR) API is a JavaScript API to asynchronously send HTTP requests from a web browser to a web server. It is a component of AJAX programming (Asynchronous

JavaScript and XML), a set of web development techniques that allow web pages to be updated by exchanging data with a web server behind the scenes.

The XMLHttpRequest API is defined by the WHATWG XMLHttpRequest living standard [45]. The API is event-driven and exposes a series of methods and properties that can be used to retrieve data from a server. The most relevant methods of the XHR API are:

- `open(method, URL, [async, user, password])` : initializes a request with the given parameters.
- `send([body])` : sends the request to the server with the optional body.
- `abort()` : aborts the request if it has already been sent.

Fetch

The Fetch API is a JavaScript API that provides an interface for fetching resources over the network. It is the successor of the XMLHttpRequest API and is regulated by the WHATWG Fetch living standard [43].

The Fetch API is promise-based and exposes a series of methods and properties that can be used to create and manage requests. The main method of this API is `fetch(resource)` which creates a request for the resource provided (usually a URL) and returns a promise that resolves to the response.

WebSockets

The WebSocket protocol provides full-duplex communication channels over a single TCP connection, ensuring lower overhead than half-duplex alternatives such as HTTP polling. With a WebSocket connection, the server can send content to the browser without the need for a request to be made by the client first, allowing messages to be passed back and forth while keeping the connection open. The connection is in cleartext by default but can be encrypted via the Secure WebSocket protocol (WSS), built on top of TLS.

The WebSocket API is a JavaScript API that provides an interface for creating and managing WebSocket connections. The API is regulated by the Whatwg HTML living standard [41]. The WebSocket API is event-driven and exposes a series of methods and properties that can be used to interact with the server. The most relevant methods of the WebSockets API are:

- `new WebSocket(url)` : the constructor used to create a new WebSocket connection.
- `send(data)` : sends data to the server. The data can be a string, a Blob, an ArrayBuffer, or an ArrayBufferView.
- `close()` : closes the connection.

Being event-driven means that the WebSocket API exposes a series of events that can be used to handle the different stages of the connection. For example, when a message is received from the server, the browser dispatches a `message` event.

To react when the browser receives data from the server, one can register a listener for the `message` event: `socket.onmessage = (event) => alert(event.data)`.

2.1.3 Identifying resources on the web

A Uniform Resource Identifier (URI) is a unique sequence of characters that identifies a resource used in the web.

URL

A URL (Uniform Resource Locator) is a subset of URIs that contains information about the location of a resource on the web. The URL specification is defined by the WHATWG URL living standard [44]. A URL consists of the following components

- **Scheme**: the protocol used to access the resource.
- **Authority**: optional component to provide authentication information.
- **Host**: the domain name or IP address of the server where the resource is located.
- **Port**: the port number on which the server is listening for requests.
- **Path**: the location of the resource on the server.
- **Query**: a set of key-value pairs that are appended to the URL and are used to pass additional information to the server.
- **Fragment**: a string of characters that identifies a secondary resource within the primary resource.

An example of a URL is the following string:
https :// user:pass @ example.com : 8080 /path/to/resource ? key1=value1&key2=value2 # fragment

Main URI schemes

The scheme is the initial part of a URI, preceding the colon (:) and indicates how the identifier should be interpreted or accessed. It defines the namespace and the protocol for the rest of the URI. The most common URI schemes are:

- **`http:`** and **`https:`** used in URLs to access resources on the web via the HTTP and HTTPS protocols.
- **`ws:`** and **`wss:`** used in URLs to access resources on the web via the WebSocket and Secure WebSocket protocols.
- **`about:`** used in browsers to reveal internal state and built-in functions. Two examples of resources using this scheme are `about:blank` and `about:srcdoc`. The first is used to create empty documents. The second is the fixed URI of iframes whose content comes from the `srcdoc` attribute.

- `data:` used to include data in-line in web pages as if they were external resources.
- `blob:` used to create URLs that reference binary data as if they were external resources.
- `file:` used to reference files on the user's computer.

2.1.4 Classifying resources based on the content types

Multipurpose Internet Mail Extensions (MIME) types are a standard way of classifying files on the Internet according to their nature and format. They are defined and standardized by the RFC 6838 [23]. MIME types are composed of two parts: a type and a subtype, separated by a slash (/). The type represents the general category of the data, while the subtype specifies the exact kind of data. For example, the MIME type `text/html` indicates that the document is a text file and its content is HTML. A MIME type always has both a type and a subtype, never just one or the other.

An optional parameter can be added to provide additional details, such as the character encoding used. The parameter is defined with the format: `type/subtype;parameter=value`

MIME types are case-insensitive but are traditionally written in lowercase. The parameter values can be case-sensitive.

Some relevant MIME types are:

- `text/html` : represents an HTML document.
- `text/css` : represents a CSS stylesheet.
- `text/javascript` : represent JavaScript code.
  Previously, the MIME type `application/javascript` was also used but is now deprecated.
- `image/png` : represents a PNG image.
- `image/jpeg` : represents a JPEG image.
- `application/json` : represents a JSON document.
- `application/octet-stream` : used to represent arbitrary binary data.

2.1.5 HTTP

The Hypertext Transfer Protocol (HTTP) is a stateless application-level protocol for distributed, collaborative, hypertext information systems. The protocol is currently regulated by the RFC 9112 [24]. The protocol operates on a request-response model: a client sends a request to the server which responds with the requested data and/or an appropriate status code. In HTTP each request from a client to a server is treated in isolation and without reference to any previous requests.

The default port for HTTP communications is 80.

HTTP request

An HTTP request is made by a client to a server with the aim of accessing a resource hosted by the server. To make the request, the client uses a URL that identifies the location of the resource to be fetched. The HTTP request is composed of a series of parameters:

- **Method**: indicates the desired action to be performed. The most common methods are `GET`, `POST`, `DELETE` and `OPTIONS`.
- **URL**: the URL of the resource to be fetched.
- **Version**: the version of the HTTP protocol used to perform the request.
- **Headers**: a set of key-value pairs that provide additional information about the request.
- **Body**: the body of the request, used to send data to the server.

Request headers

HTTP request headers provide meta-information about the resource to be fetched or about the client itself in order to allow the server to tailor the response accordingly. For this thesis, the following request headers will be considered:

- **User-Agent**: a string that identifies the client software.
- **Host**: the domain name of the server (and optionally the port number).
- **Accept**: a list of acceptable content types for the response, expressed using the MIME type format.

An empty line is used to separate the headers from the body of the request.

HTTP response

An HTTP response is made by a server to a client to provide the resource that was requested, or inform the client of the success of the action requested. The HTTP response is composed of a series of parameters:

- **Version**: the version of the HTTP protocol used to perform the request.
- **Status code** and **Status text**: a numeric code and textual equivalent that indicates the result of the request. Examples of response status are `200 OK`, `301 Moved Permanently`, `404 Not Found`, and `500 Internal Server Error`.
- **Headers**: a set of key-value pairs that provide additional information about the response.
- **Body**: the body of the response, used to send data to the client.

Response headers

HTTP response headers provide meta-information about the response to allow the client to interpret it accordingly. For this work, we will focus on the following response headers:

- **Content-Type**: the MIME type of the response body.
- **Location**: the URL of the resource to be fetched in case of a redirection.

2.1.6 Same Origin Policy

The SOP is a fundamental security mechanism of web browsers and consists of restricting the interaction between documents and resources having different origins. It prevents malicious websites from accessing sensitive information or performing unauthorized actions on behalf of users by isolating different origins, disallowing one from interacting with other's resources such as cookies, DOM elements, web requests, and web storage.

SOP applies to various resources, but the mechanism lacks a formal specification. As a consequence, different browsers may implement it in slightly different ways. [30].

Definition of origin

The SOP mechanism is based on the well-standardized concept of origin. The concept of origin is regulated by the RFC 6454 [22]. Two URLs have the same origin if the protocol, port (if specified), and host are the same for both. When the port is omitted, the default port of the scheme is implicitly assumed. The RFC defines an algorithm (Snippet 2.1.1) to evaluate whether two URLs are same-origin.

```
   Two origins are "the same" if, and only if, they are identical.  In
   particular:

   -  If the two origins are scheme/host/port triples, the two origins
      are the same if, and only if, they have identical schemes, hosts,
      and ports.

   -  An origin that is a globally unique identifier cannot be the same
      as an origin that is a scheme/host/port triple.

   Two URIs are same-origin if their origins are the same.
```

Snippet 2.1.1: RFC 6454 Algorithm to evaluate whether two URLs are same-origin

According to the RFC, if we consider the URL `https://www.example.com/path`, then:

- `https://www.example.com/other` will be same-origin.
- `https://www.example.com:443` will also be same-origin as omitting the port evaluates to the default one.

- `http://www.example.com` will not be same-origin as the schemes are different.
- `https://other.example.com` will not be same-origin as the hosts are different.

SOP and cross-origin resources

Interaction between two resources having different origins is regulated by SOP as follows:

- **Cross-origin writes**: Examples are links, redirects, and form submissions. Typically allowed by SOP.
- **Cross-origin embeddings**: Examples are scripts, iframes, stylesheets, images, and videos. Typically allowed by SOP (but potentially regulated by other policies, such as Mixed Content or X-FrameOptions)
- **Cross-origin reads**: Examples are XHR and Fetch requests. Typically disallowed by SOP (but potentially allowed by CORS). WebSocket cross-origin communication is not disallowed by SOP.

Risks of cross-origin embeddings

Cross-origin embedding refers to the practice of including subresources from a different origin than that of the embedding document. Subresources that can be subject to cross-origin embedding are:

- **Scripts**: included in a page via the `<script>` tag.
- **Stylesheets**: included in a page via the `<link>` tag.
- **Resource Media**: included in a page via `<img>`, `<video>` and `<audio>` tags.
- **Frames**: included in a page via the `<iframe>` tag.
- **Fonts**: included in a page via the `@font-face` CSS rule.
- **Objects**: included in a page via the `<object>` tag.

There are risks linked with cross-origin embedding: we previously considered how scripts can potentially access the full DOM of the embedding page. This may lead to the disclosure of sensitive information, as the script will have control over the resources of the embedding page.

Therefore, we say that cross-origin embeddings do enable third-party dependencies: relying on resources provided by a third party (like scripts from CDNs) means that if the third party is compromised the embedding page will be exposed.

In general, if an attacker has control over the embedded content, it can leverage it to perform malicious actions on the embedding page.

2.2 Securing communications with HTTPS

When using HTTP, the data exchanged between the client and the server is sent in cleartext. This enables a man-in-the-middle attacker to intercept the transmitted data and read or modify the content.

HTTPS (Hypertext Transfer Protocol Secure) is an extension of the HTTP protocol, designed to ensure secure communication over the web. HTTPS leverages the TLS (Transport Layer Security) cryptographic protocol that provides encrypted communication, ensuring both the authenticity and integrity of data exchanged between a user's browser and the web server. HTTPS piggybacks HTTP entirely on top of TLS, this way the entirety of the underlying HTTP protocol can be encrypted, that is request's URL, query parameters, headers, and cookies. The default port for HTTPS communications is 443.

2.2.1 Security of HTTPS

When using HTTP, the data exchanged between the client and the server is sent in cleartext. Moreover, the client cannot be sure that the server it is communicating with is the one it expects.

HTTPS is designed to solve these problems by providing three main security properties:

- **Authentication**: ensuring that the data is coming from the expected server.
- **Confidentiality**: ensuring that the data is not readable by third parties.
- **Integrity**: ensuring that the data is not altered during transmission.

To obtain these goals HTTPS relies upon a combination of three key ingredients: symmetric encryption, asymmetric encryption, and certificates.

Authentication

When a website decides to use HTTPS, it requests a digital certificate from a third party called Certification Authority (CA). This certificate contains the website's public key and is digitally signed by the CA. When a user connects to the website, the server presents this certificate. The user's browser contains a list of trusted CAs and can verify the certificate's authenticity by checking the CA's digital signature. If the certificate is valid and matches the server's domain, it confirms the server's authenticity.

HTTPS does not require authenticating the client: it is possible to use client certificates, but this is rarely done in practice.

Confidentiality

The TLS protocol provides confidentiality by encrypting the data exchanged between the client and the server. The TLS handshake is preliminary to the actual data exchange and is used to establish a secure connection. During the TLS handshake, the client and server negotiate a shared secret key that is used to encrypt and decrypt data. To do so they rely on asymmetric encryption: the client obtains the server public key from the server certificate and uses it to encrypt its own public key, generated on the fly. They negotiate a shared secret key by exchanging messages encrypted with their own private key and decrypted with the other's public key. Once the symmetric key is exchanged, they can use it to encrypt and decrypt data with a symmetric encryption algorithm.

Integrity

When data is sent over HTTPS, a hash of the data is generated using a cryptographic hash function. This hash is then encrypted with the sender's private key, creating a digital signature. Upon receiving the data, the recipient can decrypt the digital signature using the sender's public key, obtaining the original hash. The recipient then computes the hash of the received data and compares it to the received value. If they match, it confirms that the data has not been tampered with during transmission.

2.2.2 Improper HTTPS deployment

Deploying HTTPS is a challenging task due to the technical complexity of its underlying protocols (i.e., HTTP, TLS) as well as the complexity of the TLS certificate ecosystem and this of popular client applications such as web browsers [32].

Many websites still avoid ubiquitous encryption and force only critical functionality and sensitive data access over encrypted connections while allowing more innocuous functionality to be accessed over HTTP. In such contexts, the bad TLS configuration may provide a false sense of security.

The following is a quick overview of the most common mistakes in HTTPS deployment.

Using invalid, revoked or expired certificates

If the server is using an invalid certificate, the client will not be able to verify its authenticity. This scenario may occur when the server is using a self-signed certificate or a certificate issued by an untrusted CA.

Another problem with HTTPS certificates comes with the expiration date. Certificates have a limited validity period, after which they are no longer considered valid. Certificate renewal must be performed before the expiration date or the certificate is considered invalid.

Revoking certificates is a mechanism used to invalidate a certificate before its expiration date. This is necessary when the certificate's private key is compromised, the certificate was issued erroneously, or for other reasons that might undermine the certificate's trustworthiness. There are several mechanisms in place to handle certificate revocation, such as Certificate Revocation Lists (CRLs) and the Online Certificate Status Protocol (OCSP).

Modern web browsers do present a warning message to the user when they detect an invalid, expired, or revoked certificate, giving the user the option to proceed to the website anyway but suggesting to avoid it.

Using weak cipher suites

The cipher suite is a combination of authentication, encryption, message authentication code (MAC), and hashing algorithms used. During the handshake, the client and the server negotiate a cipher suite to use for the connection. This choice is based on the order of preference of the client among the ciphers supported by both.

Many common HTTPS misconfigurations are caused by choosing the wrong cipher suites. Old or outdated cipher suites are often vulnerable to attacks. It's essential to disable weak or deprecated ciphers and ensure that the server is configured to use strong, modern cipher suites.

Both SSLv2 and SSLv3 are deprecated and should not be used. Cipher suites of such versions are in the form `SSLv2_*` and `SSLv3_*`.

Similarly, suites using weak hashing algorithms such as MD5 and SHA1 are deprecated and should not be used. Cipher suites of such versions are in the form `*_MD5` and `*_SHA`.

Redirect from HTTP

A user may attempt to navigate to a website that supports HTTPS via plain HTTP. This could happen when he manually enters the URL with the `http` scheme, or he may be following an outdated link. It is important that in this case, the web server redirects the user to the HTTPS version of the website.

When the server receives an HTTP request, it should respond with a `301 Moved Permanently` status code and a `Location` header containing the HTTPS URL. This way the user's browser will automatically redirect the user to the HTTPS version of the website for the current connection as well as future ones.

SSL stripping

Even if the webserver supports HTTPS redirect, an attacker can still intercept the initial HTTP request and tamper with it. This scenario enables the SSL stripping attack [25] where the attacker can prevent the redirect from happening.

The man-in-the-middle attacker modifies the initial response to remove the redirect to HTTPS. The user's browser will then load the HTTP version of the website, allowing the attacker to intercept all the following traffic. To enable a transparent attack, the attacker will set up an HTTPS connection to the server, encrypting and forwarding the traffic he received in cleartext from the client. When the server sends the encrypted reply, the attacker will decrypt it (removing, or "striping" the SSL encapsulation) and forward it to the client, who will be unaware of the fact that the server is sending encrypted traffic.

2.2.3 Current HTTPS deployment on the web

HTTPS is the de facto standard for secure communication over the web and nowadays, it is used by most of the websites. Google's HTTPS encryption on the web - Transparency Report [19] is a reliable source of up-to-date statistics on the percentage of pages loaded over

HTTPS. The report presents data collected via two different sources: the Chrome browser (via the usage statistics collected by default) and the Google search engine.

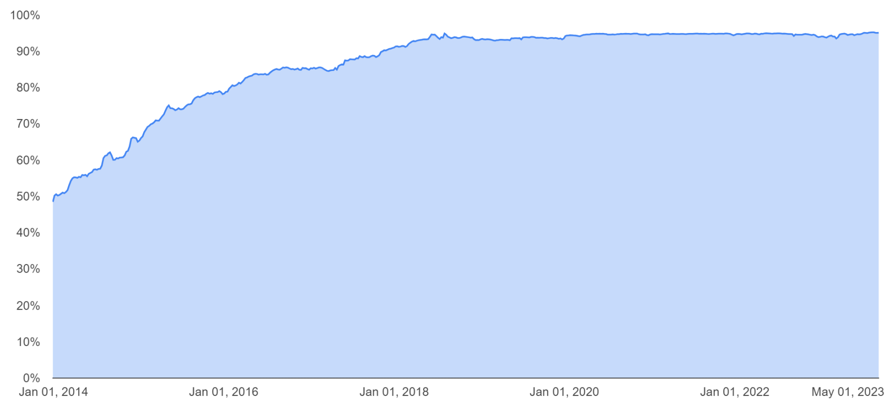

Figure 2.1: Websites using HTTPS. Data from [19]

The report provides information at different levels of granularity: country, device type (mobile or desktop), and platform (OS where the Chrome browser is installed). From the report, it is possible to observe how the percentage of pages loaded over HTTPS has been steadily increasing over the years (Figure 2.1). Upon desktop platforms, the report presents higher percentages of pages loaded over HTTPS under Windows compared to macOS and Linux (Figure 2.2).

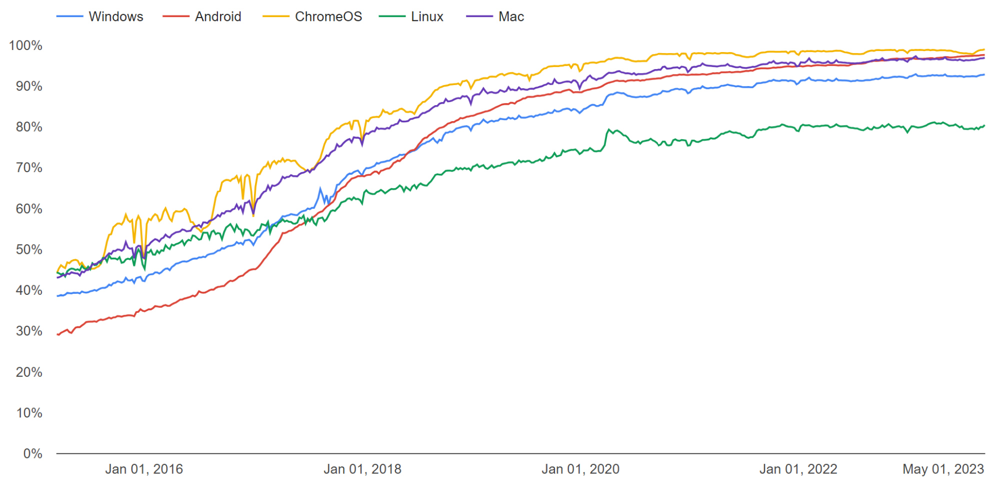

Figure 2.2: Encrypted traffic of Chrome users by OS platform. Data from [19]

The data provided by the Google report are in line with the results presented by the 2022 Web Almanac. The Web Almanac presents data collected via the HTTP Archive [31], a project that crawls the web and collects information on publicly available websites.

According to Web Almanac, in 2022 the percentage of requests that used HTTPS was 94% on desktop and 93% on mobile.

2.3 Mixed Content

In July 2014, W3C's *Web Application Security Working Group* published the *First Public Working Draft* of the Mixed Content Specification. The draft was intended as a reference for an open discussion on how user agents should handle the fetching of insecure resources in the context of secure documents. The specification went through a series of revisions and currently, it is a *W3C's Candidate Recommendation Draft* [34]. As stated by the document, the aim of the specification is that of "describing how a user agent should handle fetching of content over unencrypted or unauthenticated connections in the context of an encrypted and authenticated document".

The Mixed Content specification has to cover a relevant security aspect: when a user loads a document through a secure channel (such as HTTPS), the user agent can assert the authentication of the connection, the establishment of encrypted communication, and guarantee data integrity. However, the strength of such assertions is considerably diminished when subresources (such as frames, scripts, and images) are loaded via an insecure channel. As requests for such subresources are wide open for man-in-the-middle attacks, there could be a significant impact on user privacy and security if such are loaded into the document. The status of such content is defined as mixed, given that security assertions can no longer be expressed.

The specification details how user agents can mitigate these risks by limiting a resource's ability to communicate in the clear, filtering requests and responses based on secure characteristics of the current document.

Different types of mixed content are defined, based on the type of subresource being fetched. It is meaningful to perform such distinction as there is a difference in the risk of including active elements such as scripts or stylesheets compared to passive elements such as images, audio, and videos. For example, an insecurely loaded script allows the attacker arbitrary code execution in the context of the page. This can result in destructive consequences as other security mechanisms won't help in protecting the user (as observed in 2.1.6, SOP won't block the content).

On the other hand, an image loaded as mixed content will not expose the user to the same risk. Images do not contain any type of executable code or command, and their inclusion in the page will result in the simple visualization of the data fetched. The risks in this case are related to the context where the image is used: if we consider for example a web e-mail client that uses images to populate the two buttons "Send" and "Delete draft", a man-in-the-middle attacker can swap the content of the two network responses and phish the user. Even if this represents a clear security problem, the risk is much smaller compared to the previous case.

When we considered the various HTTPS misconfigurations (such as certificate problems or bad cipher suites), we noted how web browsers present warning messages to the user if any of those problems are detected. This happens during the TLS handshake: before any sensitive content is exchanged.

As mixed content exposes the user to similar risks as HTTPS misconfigurations, if there was no regulation put in place against mixed content presence in webpages, then users would be silently exposed. This is exactly what happened before the specification was proposed: no warning message was presented, and content was silently loaded on the page, exposing the unaware users.

2.3.1 Upgradeable content

The W3C Mixed Content specification defines mixed content as *upgradeable* when the risk of allowing its usage as mixed content is outweighed by the risk of breaking significant portions of the web. This category includes:

1. Requests whose destination is "image". This corresponds to most images loaded via `<img>`, with the exception of the ones that do use the `srcset` attribute or are embedded in a `<picture>` tag. The CSS proprieties `background-image` and `border-image` also belong to this category.

2. Requests whose destination is "video". This corresponds to elements loaded via `<video>` and `<source>`.

3. Requests whose destination is "audio". This corresponds to elements loaded via `<audio>` and `<source>`.

2.3.2 Blockable content

Any mixed content that is not upgradeable is considered to be *blockable*. Some elements that do belong in this category are:

1. JavaScript code included in the page via `<script>`.

2. CSS code included in the page via `<style>`.

3. Frames included in the page via `<iframe>`.

4. WebSocket endpoints contacted via the `WebSocket` API.

5. Requests made via the `XMLHttpRequest` and `fetch` APIs.

6. Fonts loaded via the CSS rule `@font-face`.

7. External resources included via `<object>` and `<embed>`.

8. Forms actions defined via `<form action="...">`.

9. Web application manifest included via `<link rel="manifest" href="...">`.

10. Anchors presenting an hyperlink auditing endpoint via `<a ping="...">`.

2.3.3 Overview on Mixed Content Filtering

The Mixed Content specification proposes a series of algorithms to determine if a certain resource is mixed content or not. The first set of algorithms is defined to determine if the context of a request is a relevant one for filtering mixed content. If a document is loaded over HTTP, then there is no need to perform any filtering as all the requests will be insecure. Mixed content is only relevant when the document is loaded over a secure protocol, such as HTTPS.

To reason about the safeness of including a resource over a specific environment, the specification adopts the concept of *potential trustworthiness*. If the environment is potentially trustworthy, then resources loaded over it cannot be mixed content.

As a consequence, resources loaded in a potentially trustworthy environment are checked to see if they are potentially trustworthy as well. If the checks report that the resource is not potentially trustworthy, then it is mixed content and the respective network request cannot proceed.

Sometimes, reasoning about trustworthiness can be difficult. For example, framed pages may inherit the trustworthiness of the parent page. This is the case of a framed page loaded via a `data:` URI. For this reason, the specification defines that the ancestors' chain should be considered when determining if mixed content should be blocked or not in a particular environment. Performing the checks may therefore require recursively exploring the ancestor chain, going up different levels (and eventually considering the top level).

When a particular resource is categorized as mixed content, dealing with it is a matter of deciding whether to block the request or to upgrade it to a secure one. The specification defines that the request should be blocked if the content is categorized as blockable. If the content is upgradeable, an auto-upgrade process is attempted. Autoupgrading the request is done by changing the scheme of the request's URL to `https`. This way the request will be performed over a secure channel opening the possibility to load the resource securely. The resource will not be loaded on the page if the auto-upgrade fails (i.e. if the server cannot provide the resource securely).

It's possible to understand how the approach proposed by the specification is about strictly blocking mixed content. This means that all the resources loaded by the page will be retrieved over a secure connection. This is the safest approach as no possibility is left for a network attacker to tamper with any of the content of the page. However, this approach may lead to breakages in the page.

2.3.4 Obsolescences

Earlier versions of the specification proposed a different filtering approach. Blocking the loading of insecure images, audio, and video was not enforced: the content was loaded and an in-between security indicator was shown (generally the removal of the padlock icon in the address bar). Eventually, this method did not provide a clear indicator of risk to users and as a consequence, there was little to no incentive for developers to avoid mixed content in webpages.

This pushed W3C to come up with the latest revision of the specification, with the current approach commonly referenced as “strict mode” or “strict blocking”, where the webpage is loaded entirely over a secure transport. With the current approach, the auto-upgrading of content comes as a mitigation of potential breakages (other than an approach to increase security), as well as a solution to minimize the amount of developer effort.

Previous namings

The approach attempted with previous versions of the specification explains the former naming of upgradeable content, that was *optionally blockable* as the decision to block it was delegated to browser vendors, with the majority opting for allowing the content and informing the user with a small UX change (the aforementioned padlock icon removal).

Another naming is commonly adopted with *active content* used to reference blockable mixed content and *passive content* to reference upgradeable mixed content. Although this naming is very popular, it is not officially adopted by the current version of the specification.

2.3.5 Integrations of Mixed Content

The specification describes how the Fetch specification should hook into the algorithms for mixed content filtering before performing the network request.

In Snippet 2.3.1, the first steps of the Main Fetch algorithm implementation (defined in the WHATWG Fetch specification [43]) with the integration of the auto-upgrade and Mixed Content filtering at line 6 and line 7.

```
  1   Let request be fetchParams's request.
  2   Let response be null.
  3   If request's local-URLs-only flag is set and request's current URL is not local,
      ↪  then set response to a network error.
  4   Run report Content Security Policy violations for request.
  5   Upgrade request to a potentially trustworthy URL, if appropriate.
  6   Upgrade a mixed content request to a potentially trustworthy URL, if
      ↪  appropriate.
  7   If should request be blocked due to a bad port, should fetching request be
      ↪  blocked as mixed content , or should request be blocked by Content Security
      ↪  Policy returns blocked, then set response to a network error.
  8   If request's referrer policy is the empty string, then set request's referrer
      ↪  policy to request's policy container's referrer policy.
...   ...
```

Snippet 2.3.1: Algorithm "*Main Fetch*"

Overview on Chromium implementation of Mixed Content checks

In the following, a concrete implementation of the Mixed Content specification with Chromium source code. The code is taken from the Chromium GitHub repository [9].

The method `MixedContentChecker::ShouldBlockFetch` is called when a request is about to be performed. It checks if the request can go through or if it should be blocked as mixed content. The decision is based on the final value of the variable `bool allowed` defined at line 446.

First, the method checks if the environment does prohibit mixed content via the `InWhichFrameIsContentMixed` method call. This call will return true if the environment is trustworthy (that is, if any of the ancestors were loaded via `https`). If the call returns false, the method exits immediately as the content should not be blocked. If the call returns true then the checks continue.

The content of the request type is then categorized via the `MixedContent::ContextTypeFromRequestContext` method call.

If the request is blockable then `allowed = false`. If the request is upgradeable (here referenced via the obsolete terminology *optionally blockable*) then the filtering will depend on whether the strict mode is enabled (the current default is `true`).

Snippet 2.3.2: Chromium implementation of "*Should fetching request be blocked as mixed content?*", file `blink/renderer/core/loader/mixed_content_checker.cc` [9]

```
  1   // Copyright 2019 The Chromium Authors
...
402   bool MixedContentChecker::ShouldBlockFetch(LocalFrame* frame,
...       ...
411       mojom::blink::ContentSecurityNotifier& notifier) {
412   Frame* mixed_frame = InWhichFrameIsContentMixed(frame, url);
413   if (!mixed_frame)
414       return false;
...   ...
440   Settings* settings = mixed_frame->GetSettings();
441   auto& local_frame_host = frame->GetLocalFrameHostRemote();
442   WebContentSettingsClient* content_settings_client =
443       frame->GetContentSettingsClient();
444   const SecurityOrigin* security_origin =
445       mixed_frame->GetSecurityContext()->GetSecurityOrigin();
446   bool allowed = false;
...   ...
457   mojom::blink::MixedContentContextType context_type =
458       MixedContent::ContextTypeFromRequestContext(
459           request_context, DecideCheckModeForPlugin(settings));
460
461   switch (context_type) {
462       case mojom::blink::MixedContentContextType::kOptionallyBlockable:
463           allowed = !strict_mode;
...           ...
485       case mojom::blink::MixedContentContextType::kBlockable:
...           ...
526           allowed = false;
527       case mojom::blink::MixedContentContextType::kShouldBeBlockable:
528           allowed = !strict_mode;
...           ...
532       case mojom::blink::MixedContentContextType::kNotMixedContent:
```

```
            NOTREACHED();
            break;
        };
    ...
```

3 Methodology

3.1 Overview on the WPT project

The Web Platform Tests (WPT) project [16] is a cross-browser test suite for the web platform stack started in 2014. Its main goal is to give browser vendors confidence that they are shipping software that is compliant with specifications and compatible with other implementations. The majority of the test suite consists of HTML pages that can be loaded in a browser and automatically provide a result. In general, the tests are cross-platform, short, and self-contained and can be easily run in any browser. The WPT test suite is integrated into the CI/CD pipeline of all major browsers [27, 8, 13], with Firefox and Chromium officially including it in 2014 [28, 20]. All the major browser vendors do contribute to the test suite and depend on its comprehensive set of checks to confirm compliance with web standards [18]. According to Firefox, Chromium, and WebKit contribution policies, each revision or patch should pass all WPT regression tests before it is approved [26, 2, 12].

3.1.1 Test Layout

Most of the repository top-level directories hold tests for specific web standards. In the case of W3C specifications, the directories are named after the short name of the spec (as an example, there are top-level folders named `/mixed-content`, `/service-workers` and `/secure-contexts`). Within the specification-specific directory, the test is laid out either with a flat structure or with a nested structure. The first is used with short or simple specifications, the latter presents subdirectories corresponding to the topic within the specification and is adopted with larger specs. For example, WPT tests on mixed content auto-upgrading are present in `/mixed-content/tentative/autoupgrade`.

Test filenames should be descriptive of what is tested, one option could be `test-topic-001.html` where `test-topic` is a short identifier that describes the test.

The test filename is significant in enabling specific features. For example, by default, all tests are served over plain HTTP, so if a test requires HTTPS it must be given a filename containing `.https` e.g. `test-topic-001.https.html`. In Table 3.1 a list of WPT filename flags and the respective feature.

Tests are generally defined as HTML (including XHTML) or XML (including SVG). It is also possible to define a test as a single JavaScript file (e.g. when including any JavaScript flag specified in Table 3.1, the WPT server will automatically generate the respective HTML file).

Test Flag	Feature
`.https`	Test loaded via HTTPS
`.h2`	Test loaded via HTTP/2
`.www`	Test is run on the `www` subdomain
`.sub`	Test uses server side substitution feature
`.window`	Generates a test run in a Window environment
`.worker`	Generates a test run in a dedicated worker environment
`.any`	Generates a test run in multiple scopes
`.tentative`	Test that makes assertions not yet required by any specification, or in contradiction to some specification

Table 3.1: WPT Test flags and respective feature. ▢: JavaScript files only

Server Features

Having just one or more HTML files may not be sufficient to cover specific test cases. More advanced tests may require support for extra features, such as:

- Cross-domain access.
- Setting specific headers or status codes.
- Require state to be stored on the server.
- Inspect the browser-sent request.

The WPT project includes server-side components that add support to these extra features. In particular:

- A custom Python HTTP server: *wptserve*.
- A websocket server: *pywebsocket*.

3.1.2 Javascript Tests

If a WPT test does not involve rendering and does not require user interaction, it can be defined as a Javascript test. The WPT project includes a framework for defining Javascript tests called `testharness.js` [15]. The framework provides APIs to test both synchronous and asynchronous DOM features. To use the framework, the test must include the `testharness.js` and `testharnessreport.js` scripts as shown in Snippet 3.1.1. Within a single file, it is possible to define multiple tests, each providing a single result as (`PASS/FAIL/TIMEOUT/NOTRUN`).

The final result of a test is in `PASS/FAIL` based on the result of checking for one or more assertions. The test fails at the first failing assert, and the remainder of the test is not executed. Functions for making assertions have a signature like
`assert_something(actual, expected, description)`.

An example of assertion function is `assert_true(actual, description)`, that fails if the expression `actual` does not evaluate to true. The `description` parameter is optional and is used to provide a more detailed description of the assertion in the result of the test.

```html
<!DOCTYPE html>
<meta charset="utf-8">
<title>${1:Test title}</title>
<script src="/resources/testharness.js"></script>
<script src="/resources/testharnessreport.js"></script>
<script>
test(function() {
  assert_true(document.implementation.hasFeature());
}, "test that never fails")
</script>
```

Snippet 3.1.1: Example of `testharness.js`

The body of the test is defined as a function passed to the `test(func, name)` function together with a name for the test.

Execution of tests on a page is subject to a global timeout. By default, this is 10 seconds, but a test runner may extend the timeout using a `meta` tag: `<meta name="timeout" content="long">`, this will extend the timeout to 60 seconds.

3.1.3 Practical considerations on the WPT test suite

There is a series of aspects to be considered with the adoption of the WPT test suite in browser development pipelines. Browser vendors leverage the results of the tests to confirm compliance with web standards, using it to check a given revision or patch before it is approved.

However, there is a series of practical constraints and practices that limit the effectiveness of the results of tests to identify security issues.

Flaky tests

One aspect is the contribution that browser vendors offer to the test suite.

It is common practice for developers of the principal web browsers (Chrome, Firefox, and Safari) to contribute to the test suite by adding new tests or updating existing ones. For example, when they implement a particular feature or fix a specific bug, they often write a set of WPT tests that are added to the suite. This practice helps with increasing the coverage of the test suite but often introduces a bias in the results of the tests. When a specific browser vendor does write a test for a given mechanism, it could be that the assertions he proposes are biased towards his implementation.

This means that running that test on other platforms leads to misleading results, where failures are reported even when no security violation happened. As a consequence, it is a common practice for vendors to identify and mark sets of tests as *flaky* and ignore the results in their pipelines.

In this case, however, the problem is with the assertions of the test, not with the test itself. It could be that in the future, the execution of the test may lead to a security violation, but if the test is marked as flaky, this will go unnoticed by browser vendors.

Outdated tests

Another aspect to consider is the maintenance of the test suite. When a new specification is released, a series of tests are written to cover the new features. Often the tests are published while the work on the specification is still in progress.

Such a set of tests has assertions that are based on the current state of the specification, but the specification itself may change and as a consequence results of the tests may become outdated.

Sometimes it is enough just to change the assertions of the test, keeping the test itself unchanged. For example, when the mixed content specification was updated in February 2023 to introduce the strict blocking of mixed content, many tests already available in the suite were still meaningful, it was only required to change the assertions to match the new behavior.

In practice, however, such maintenance is not always done, and tests are left with outdated assertions. A consequence of this is that browser vendors may mark those tests as outdated, ignoring them for future evaluations.

Tests that are too specific

When a security violation is reported to a browser vendor, it is common practice that a WPT test is written to reproduce the bug.

When this happens, the test is often written to reproduce the exact conditions that lead to the bug. This may include a specific sequence of actions, with a specific timing, and a specific set of parameters. It could be that to cover the violation it was possible to define a more generic test that would cover a wider range of conditions. This may require a more detailed reasoning on the nature of the problem and a given vendor may not be willing to invest the time to do so.

The problem with this practice is that when specific tests are reporting a failure on some other platform, the developers may assign a priority to the bug that is solely based on the specific conditions of the test, while in practice the violation may be more severe as present in a wider range of conditions.

Tests that do not assert on all the components tested

A single test may involve a set of different mechanisms and features. Sometimes, however, the assertions of the test are only defined for a subset of the components involved. This is the case of tests created to cover a given specification, where the interest is on the compliance of the browser to the specification itself, and not on the effects of the other components involved.

It could be that a test produces a security violation on a component that is not covered by the assertions of the test. In this case, the violation will go unnoticed by the browser vendor, as the test will be marked as passing.

3.2 Leverage WPT for automatic verification of web security mechanisms

The WPT project represents the largest benchmark of intended browser behavior to date.

This chapter presents the idea of leveraging the WPT test suite to automatically verify the intended behavior of web security mechanisms. The framework proposed collects information about the execution of the tests, and uses it to formally and automatically perform a verification against a model of the security mechanisms.

A set of execution traces (i.e., sequences of relevant browser events) is collected from test runs. Traces are then matched against web security invariants, which are intended security properties expressed in first-order logic. The invariants are crafted from specifications of web security mechanisms.

This way, it is possible to automatically identify flaws in the implementation of security mechanisms as there will be certain traces violating the invariants.

The analysis will focus on WPT JavaScript tests. The invariants defined do cover both JavaScript and browser internals behavior, while UI behavior is out of the scope of this work. In Table 3.2 are presented the tests evaluated, the respective folders, and the number of tests per folder.

In the previous section, we considered a set of practical problems in the evaluation of the results of WPT test executions. The proposed framework allows to overcome such problems, in particular:

- **Flaky tests**: the framework does not rely on potentially biased assertions of the tests. The invariants defined are independent of the browser implementation.
- **Outdated tests**: when a specification is updated, only the respective invariant has to be updated, there is no need to update single tests.
- **Tests that are too specific**: feedback on the gravity of a security violation comes from the invariant, not from the test result. The risk of underestimating the severity of a violation is reduced.
- **Tests that do not assert on all the components tested**: All the invariants are verified against each execution trace. If a violation is present, it will be identified, regardless of the assertions of the test.

3.3 Execution Traces

A trace is a list of browser events collected during the execution of a test. Among the information collected, there are:

- **JavaScript API calls**: calls to JavaScript APIs, e.g., Fetch, XHR.
- **Network requests and responses**: for both HTTP and HTTPS, as well as WS and WSS. Each is identified by a unique ID.
- **Cookie events**: information when cookies are stored in the cookie jar.

Each execution trace is stored in JSON format: an example can be seen in Snippet 3.3.1.

```
"network":{
    "requests":[
        {"requestId": "10",
            "url":{"href": "https://web-platform.test:8443/fetch/api/resources/cors-t⌋
            ↪  op.txt",...},
            "originUrl":{"href": 'data:text/html,<!DOCTYPE html><script>...',...},
            ...,
            "tabId": 2,
            "frameId": 8589934593,
            "method": "GET",
            "type": "xmlhttprequest",
            ...
            "requestHeaders":[...]
            ...
            "frameAncestors":[...],
            ...
        },...]
    "responses":[{...},...]
}
"events":[...],
"cookies": {},
"tests":{"HTTP fetch": { "start": 1694918528955, "end": 1694918529569 },...},
"proxy":{...}
```

Snippet 3.3.1: Example of the execution trace from the test `/mixed-content/nested-iframes.window.html`. Some data has been removed for compactness.

3.3.1 Browser instrumentation

Browser events are collected by instrumenting the browser with a custom extension. The code of the extension leverages the Extension API, a cross-browser framework that provides interfaces to access the content of open web pages and browser internals.

Network events monitoring

Information on network events is provided via callback functions executed when a request is sent and when a request is deemed completed (either a response was received or it was dropped). This approach provides the full content of a request, plus additional information provided by the browser, like the tab and frame IDs of the request initiator. The callbacks are added to the following events of the `browserwebRequest` object:

- `onSendHeaders` : emitted when a request is about to be sent.

- `onCompleted` : emitted when a request is completed.

- `onBeforeRedirect` : emitted when a request is about to be redirected.

JavaScript API calls monitoring

The WebExtension API allows injecting JavaScript code into the context of a web page. This feature was used to proxy the relevant JavaScript functions, by using Proxy objects and method overriding, enabling the collection of all the data associated with each API call, such as its arguments and the respective browsing context. Some APIs monitored are:

- **Fetch API**: the `fetch` function is overridden to log the parameters of web requests.

- **XHR API**: the `XMLHttpRequest` constructor is overridden to log the parameters of web requests.

- `window.open` : the function is overridden to log when a new window is opened.

- **PostMessage API**: the `postMessage` function is overridden to log when messages are posted between windows or frames.

As the WebExtension API is a cross-browser framework, most of the code of the extension is cross-compatible with the web browsers considered by the project (Chromium, Firefox, and Safari). Despite that, little changes were still required due to slight differences in the browser implementation of certain JavaScript APIs: one example is with the assignment of frame IDs in Safari, where there is a difference in the frame ID of children and the parent ID of nephews of the top level compared with Firefox and Chromium.

3.4 Web Invariants

Invariants represent models of security proprieties expressed as functions mapping traces to propositions, defined in quantified first-order logic using the theories of uninterpreted functions, integer arithmetic, algebraic datatypes, and strings.

On Snippet 3.4.1 an example of invariant to model a simplified version of mixed content filtering: if an HTTPS origin makes a network request, then the content cannot be fetched over HTTP. The invariant is defined as an implication, requiring the URL of the request to not have the `http://` scheme if the origin URL does present the `https://` scheme.

$$\begin{array}{l}
\textsc{simple-mixed-content-blocking}(tr) := \\
\quad \forall \mathit{t1} \forall \mathit{url} \forall \mathit{method} \forall \mathit{type} \forall \mathit{origin\text{-}url} \forall \mathit{doc\text{-}url} \forall \mathit{ancestors} \forall \mathit{r\text{-}hds} \forall \mathit{r\text{-}bdy} \forall \mathit{id}, \\
\quad \forall \mathit{origin\text{-}host}, \forall \mathit{url\text{-}host} \\
\quad (\textbf{\textit{net-request}}(\mathit{id}, \mathit{url}, \mathit{method}, \mathit{type}, \mathit{origin\text{-}url}, \mathit{doc\text{-}url}, \mathit{ancestors}, \mathit{r\text{-}hds}, \mathit{r\text{-}bdy}) \\
\qquad @_{tr}\mathit{t1} \wedge \\
\qquad \mathit{origin\text{-}url} = \text{"}https://\text{"} +\!\!+ \mathit{origin\text{-}host} \\
\quad \Rightarrow \neg(\mathit{url} = \text{"}http://\text{"} +\!\!+ \mathit{url\text{-}host})))
\end{array}$$

Snippet 3.4.1: Example of an invariant to check that an HTTPS origin does not fetch content over HTTP

Events (e.g. `net-request`) are defined as a datatype. The $@_{tr}$ predicate is implemented as a recursive function and represents the realization of an event at a specific time (e.g. t_1). Auxiliary predicates can be defined as either macros or functions. For example, the function $+\!\!+$ is used to concatenate strings.

On Table 3.3 the list of invariants defined for the project. Later in the chapter, the mixed content invariants will be studied in detail.

Name	Invariant	References
WS.I1	Integrity of Secure cookies	[36]
WS.I2	Confidentiality of HttpOnly cookies	[36]
WS.I3	Integrity of __Host- cookies	[36]
WS.I8	Authenticity of req. initiator	[36]
WPT.1	Integrity of SameSite cookies	
WPT.2	Cookie serialization collision resistance	[33]
WPT.3	Blockable mixed-content filterig	
WPT.4	Upgradeable mixed-content filtering	
WPT.5	Mixed-content filtering in nested contexts	

Table 3.3: Web Invariants.
⬜: Cookies, ⬜: Mixed-Content, ⬜: SOP/Origin

3.5 Trace Verification Pipeline

The WPT tests are run on the three major browsers that are instrumented by installing the extension. One execution trace is produced per each WPT test, and all the traces are collected into a database.

The traces are then post-processed, translated to SMT-LIB 2, and checked against the invariants using the Z3 theorem prover. This results in a formula that is a negation of each invariant applied to the events of a single trace. Results on the satisfiability of the formula

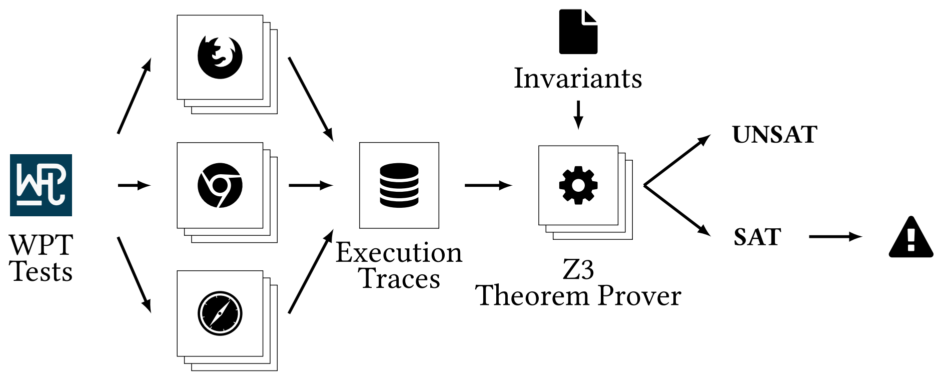

Figure 3.1: Trace Verification Pipeline.

give information on effective violations of the proposition that characterize a given invariant (i.e. a violation of a specific security property). Such violations are collected to be manually inspected.

The analysis pipeline is based on the Kubernetes container orchestration platform, allowing to execute multiple instrumented browsers and Z3 analysis in parallel.

A schematic representation of the pipeline can be found in Figure 3.1. A more detailed discussion of the pipeline follows.

3.5.1 Trace collection

Each WPT test is run in an isolated container.

Each container includes the WPT repository and a specific version of the tested browser with the extension installed.

For this project, the browsers analyzed are Chromium 118.0.5961.0, Firefox 116.0.3, and Safari 16.4.

Once a WPT test is executed, the execution trace is downloaded in JSON format and stored in a Redis database.

3.5.2 Trace verification

Upon completion of each single test, the JSON file is post-processed and translated to SMT-LIB format. The resulting code is combined with the SMT-LIB encoding of the invariants (an example of an encoded invariant is in Snippet 3.5.1). To prove that no event among all the ones collected during the test can disprove the invariant, the satisfiability is checked on the negation of the invariants.

Checking satisfiability with Z3 may have three possible outcomes:

- **UNSAT**: the invariant is true for the current trace.

- **SAT**: the invariant is not valid and the current trace is a counter-example for the invariant.

- **UNKNOWN**: Z3 was not able to prove or disprove the invariant (e.g. due to timeout).

```
(define-fun simple-mixed-content-blocking ((tr (List Action))) Bool
(forall ((t1 Int) (url String) (method RequestMethod) (type ResourceType)
↪  (origin-url String) (document-url (Option String)) frame-ancestor (redirs
↪  (List String)) (request-headers RequestHeaders) (request-body (Option
↪  String)) (id String))
(=>
    (and
        (at t1 tr (net-request id url method type origin-url document-url
        ↪  frame-ancestors redirs request-headers request-body))
        (= origin-url (str.++  "https://" cvalue)))
    (not (= url (str.++  "http://" cvalue))))))
```

Snippet 3.5.1: SMT-LIB encoding of the invariant presented in Snippet 3.4.1

In Snippet 3.5.2 a minimal example of the SMT-LIB file produced by merging the example invariant (line 6) with the encoding of the example trace (line 8). Note how in line 11 the satisfiability check is performed on the negation of the invariant.

```
1     (declare-datatypes ((RequestMethod 0)) (((M-GET) (M-POST) (M-PUT) (M-DELETE)
      ↪  (M-OPTIONS) (M-PATCH) (M-OTHER))))
2     ...
3     (define-fun-rec in (l (List Action) ...)
4     (define-fun-rec at ((n Int) ... )
5     ...
6     (define-fun simple-mixed-content-blocking ((tr (List Action))) ... )
7
8     (define-fun trace1 () (List Action) (insert (net-request "10"
      ↪  "https://web-platform.test:8443/fetch/api/resources/cors-top.txt" M-GET
      ↪  type-sub_frame ... )))
9     (echo "===== trace1 =====")
10    (echo "simple-mixed-content-blocking")
11    (assert (not (simple-mixed-content-blocking trace1)))
12    (check-sat)
```

Snippet 3.5.2: Final SMT-LIB code with the invariant from Snippet 3.5.1 and the encoded trace from Snippet 3.3.1

3.6 Encoding mixed content invariants

For the remainder of this chapter, we will study the W3C's Mixed Content specification, to extract a set of web invariants that can be used in our framework to automatically verify the implementation of mixed content filtering in web browsers.

The Mixed Content specification defines a series of algorithms to filter requests and responses. The attributes that are used by the algorithms to make the filtering decisions are:

1. Document's origin.
2. Request's URL.
3. Request's destination.
4. Request's initiator.
5. Ancestor navigables.
6. User agent configuration.

All these attributes are collected by the instrumented browsers and stored in the execution trace.

3.7 The notion of potential trustworthiness

In order to reason about the filtering algorithms it is relevant to define what the user agent should consider as secure content. The concept of mixed content is relevant only in the presence of a securely loaded document: it does not apply when the embedder page is not delivered securely (i.e. via HTTP).
The W3C Secure contexts specification [40] identifies certain hosts, schemes and origins as *potentially trustworthy*, even if they might not be authenticated and encrypted in the traditional sense. These elements are identified via the algorithms "*Is origin potentially trustworthy?*" and "*Is URL potentially trustworthy?*"

3.7.1 *Is origin potentially trustworthy?*

```
If origin is an opaque origin, return "Not Trustworthy".
Assert: origin is a tuple origin.

If origin's scheme is either "https" or "wss", return "Potentially Trustworthy".
If origin's host matches one of the CIDR notations 127.0.0.0/8 or ::1/128,
↪  return "Potentially Trustworthy".
If the user agent conforms to the name resolution rules in
↪  (let-localhost-be-localhost) and one of the following is true:
    origin's host is "localhost" or "localhost."
    origin's host ends with ".localhost" or ".localhost."
    then return "Potentially Trustworthy".
```

```
10    If origin’s scheme is "file", return "Potentially Trustworthy".
11    If origin’s scheme component is one which the user agent considers to be
      ↪  authenticated, return "Potentially Trustworthy".
12    If origin has been configured as a trustworthy origin, return "Potentially
      ↪  Trustworthy".
13    Else return "Not Trustworthy".
```

Snippet 3.7.1: Algorithm "*Is origin potentially trustworthy?*"

3.7.2 Modeling the concept of potentially trustworthy origin

The algorithm 3.7.1 is essentially a list of schemes and hosts that are considered trustworthy origins. It could be possible to model the algorithm with a regex that matches those elements. The regex can be easily defined with the schemes that are considered potentially trustworthy origins (`https://`, `wss://` and `file:`), since those are prefixes in the URL. On the other hand, checking if the host evaluates to localhost or a loopback address is a more complicated task and a bigger regex is required.

The function presented in Snippet 3.7.1 is an SMT-LIB encoding that models origin trustworthiness, it leverages the theory of character strings and regular expressions over an alphabet consisting of Unicode characters.

The specification considers the possibility of having a vendor-defined list of whitelisted origins (line 11 of the algorithm). Such a feature is used in practice by browser vendors to extend the concept of trustworthiness to certain customs origin e.g. chromium whitelisted the origins `chrome-extension://`, `chrome://` and `chrome-native://`, while firefox whitelisted `moz-extension://`. For this project, we decided to ignore such a possibility to maximize platform independence and reduce the maintenance task of periodically having to verify if some browser vendor updated its list.

```
(define-fun is-origin-potentially-trustworthy ((URI String)) Bool
(str.in.re URI
(re.union
    (re.++  ;1a. URL's protocol is https
        (str.to.re "https://")
        (re.* re.allchar))
    (re.++  ;1b. URL's protocol is wss
        (str.to.re "wss://")
        (re.* re.allchar))
        (re.++  ;2. URL's protocol is file.
        (str.to.re "file://")
        (re.* re.allchar))
    (re.++  ;3. URL's host is localhost as a domain (*.localhost or localhost)
        (re.* (re.range "a" "z"))    ;protocol
        (str.to.re "://")
        (re.opt (re.* (re.++ (re.* (re.range "a" "z")) (str.to.re "."))))
        (str.to.re "localhost")
        (re.opt (re.++ (str.to.re ":")((_ re.loop 1 5)(re.range "0" "9"))))
        (str.to.re "/")
        (re.opt (re.* re.allchar)))
    ;Loopback addresses are considered secure.
    (re.++  ;4a. URL's host is a loopback IPv4 address
        (re.* (re.range "a" "z"))
        (str.to.re "://127")
        ((_ re.loop 1 3) (re.++ (str.to.re ".")(re.opt ((_ re.loop 1
        ↪  3)(re.range "0" "9")))))
        (re.opt (re.++ (str.to.re ":")((_ re.loop 1 5)(re.range "0" "9"))))
        (str.to.re "/")
        (re.opt (re.* re.allchar)))
    (re.++ ;4b. URL's host is a loopback IPv6 address
        (re.* (re.range "a" "z"))
        (str.to.re "://[")
        (re.opt
            (re.*
            (re.++
                (re.* (str.to.re "0"))
                (str.to.re ":"))))
        (re.* (str.to.re ":"))
        (re.opt (re.* (str.to.re "0")))
        (str.to.re "1")
        (str.to.re "]")
        (re.opt (re.++ (str.to.re ":")((_ re.loop 1 5)(re.range "0" "9"))))
        (re.opt (re.++ (str.to.re "/") (re.* re.allchar)))))))
```

Snippet 3.7.2: SMT-LIB implementation of "*Is origin potentially trustworthy?*"

3.7.3 *Is URL potentially trustworthy?*

The algorithm *"Is URL potentially trustworthy?"* is used to apply the concept of trustworthiness on request and response URLs. All the URLs that are considered trustworthy origins are also trustworthy URLs.

On the other hand, certain URLs and schemes that are not considered potentially trustworthy origins are considered trustworthy URLs, as including those resources is considered safe. As an example, there is no risk in loading an iframe with the `about:blank` URL (that identifies an empty page).

```
If url is "about:blank" or "about:srcdoc"
    return "Potentially Trustworthy".

If url’s scheme is "data"
    return "Potentially Trustworthy".

Return the result of executing "Is origin potentially trustworthy?" on url’s
↪  origin.
```

Snippet 3.7.3: Algorithm "*Is URL potentially trustworthy?*"

3.7.4 Model the concept of potentially trustworthy URL

Similarly to the previous example, the trustworthiness of the URL is modeled as a function that matches the provided URL against a regex. In this case, the regular expression is much simpler as we need to match either the full URL (to check if it is either `about:blank` or `about:srcdoc`), or a prefix of the URL, to check if it is a `data:` scheme.

This results in an encoding that is equivalent to the specification algorithm.

```
(define-fun is-url-potentially-trustworthy ((URI String)) Bool
(or
    (str.in.re URI
        (re.union
        (str.to.re "about:blank")       ;1a. URL equals "about:blank"
        (str.to.re "about:srcdoc")      ;1b. URL equals "about:srcdoc"
        (re.++                          ;2. URL's protocol is data
            (str.to.re "data:")
            (re.* re.allchar))))
    (is-origin-potentially-trustworthy URI)))
```

Snippet 3.7.4: SMT-LIB implementation of "*Is URL potentially trustworthy?*"

3.8 Does *settings* prohibit mixed security contexts?

The following algorithm identifies the contexts where Mixed Content should be blocked.

While up until now only documents have been mentioned, there are other contexts where mixed content should be blocked. Web Workers for example should block mixed content as they can only be created in secure contexts (hence the origin is an HTTPS document).

The Web Worker API is standardized by the WHATWG HTML living standard. Workers allow running JavaScript code in the background, parallel to the main execution thread. This is particularly useful for performing tasks that are computationally intensive or time-consuming without blocking the main thread and thereby ensuring that the user interface remains responsive.

The WHATWG specification states that both documents and workers have an *environment settings object* that can be examined in order to determine whether they should restrict mixed content or not. The *environment settings object* specifies the origin of the environment and is used by the algorithm *Does settings prohibit mixed security contexts?*

The algorithm terminates immediately returning "Prohibits Mixed Security Contexts" if the settings object presents an origin that is potentially trustworthy. If not, and if the current settings object is a window (that is, we are not in a Worker), the frame ancestors of the current document are considered: the chain is checked to see if any of the ancestors have a potentially trustworthy origin. When a trustworthy origin is detected, the algorithm returns "Prohibits Mixed Security Contexts".

If no potentially trustworthy origin is detected among all the ancestors, the algorithm returns "Does Not Restrict Mixed Security Contexts".

```
1    If settings’ origin is a potentially trustworthy origin, then return "Prohibits
     ↪  Mixed Security Contexts".
2
3    If settings’ global object is a window, then:
4        Set document to settings’ global object's associated Document.
5        For each navigable navigable in document’s ancestor navigables:
6            If navigable’s active document's origin is a potentially trustworthy
             ↪  origin, then return "Prohibits Mixed Security Contexts".
7
8    Return "Does Not Restrict Mixed Security Contexts".
```

Snippet 3.8.1: Algorithm "*Does settings prohibit mixed security contexts?*"

3.8.1 Modeling the algorithm "*Does settings prohibits mixed security contexts?*"

Modeling the algorithm requires a more detailed analysis of potentially trustworthy origins and requires integrating `is-origin-potentially-trustworthy` with auxiliary checks.

In particular, the origin of a `blob:` URI is the origin of the context in which it was created, as a consequence, the trustworthiness is inherited from the parent.

The trustworthiness of `data:` URIs on the other hand is surprisingly more difficult to consider [35] [29] [38] [21]. The current version of the W3C Secure Context specification does not consider `data:` URI as a potentially trustworthy origin (in fact, the algorithm *Is origin potentially trustworthy?* discussed above does not list the scheme). On the other hand, browser implementations adopt the same approach of `blob:` URIs, where trustworthiness is inherited from the parent context [7] (but not by inheriting the origin like `blob:` does. In this case, only the trustworthiness is inherited). This behavior is mimicked by the algorithm: a check is performed on the frame ancestors when the current origin is not potentially trustworthy (lines 3-6). As a consequence, if a `data:` URI has trustworthy ancestors, mixed content will be blocked as if it were a trustworthy origin itself.

Such considerations on `blob:`and `data:` require the definition of auxiliary functions.

Obtain the origin of `blob:` URIs

The origin of `blob:` is the origin of the context in which they were created. Therefore, blobs created in a trustworthy origin will themselves be potentially trustworthy. The origin is included in the URL and follows the scheme.

The auxiliary function to get such information simply checks if the content following the prefix string `blob:` forms a potentially trustworthy origin.

```
(define-fun is-blob-url-potentially-trustworthy ((URI String)) Bool
    (and
        (is-prefix "blob:" URI) ;URL's protocol is blob
        (is-url-potentially-trustworthy (str.substr URI 0 5))))
```

Snippet 3.8.2: SMT-LIB function to obtain origin of a *blob:* URI

Obtain the trustworthiness of *data:* URIs

To understand if a `data:` URI is trustworthy we consider the ancestor chain. We start from the parent and check if it is a `data:` itself. If not, we return the URL of the parent. Else, if the parent frame also consists of a `data:` URI we continue the recursion by ascending the chain until we find a non-data URI to evaluate.

```
(define-fun-rec get-data-URI-origin ((ancestors (List String))) String
(match ancestors
    ((nil "")
    ((insert head tail) (ite (str.prefixof "data:" head) (get-data-URI-origin tail) head)))))
```

Snippet 3.8.3: SMT-LIB function to obtain origin of a *data:* URI

The second auxiliary function to deal with `data:` URI simply takes the ancestor chain and reverses it, then calls `get-data-URI-origin` to get the URL of a parent and calls `is-origin-potentially-trustworthy` on that. If a `data:` has an empty ancestor chain (this could happen if the user enters directly the URI in the searchbar), the function returns false (i.e. we consider `data:` URI not trustworthy on their own - much like the specification does).

```
(define-fun is-data-origin-potentially-trustworthy ((URI String) (ancestors (List String))) Bool
(and
    (is-prefix "data:" URI)
    (match ancestors
        ((nil false)
        ((insert head tail) true)))
    (is-origin-potentially-trustworthy (get-data-URI-origin (rev ancestors)))))
```

Snippet 3.8.4: SMT-LIB function to understand if *blob:* and *data:* URI are potentially trustworthy

The implementation of *Does settings prohibit mixed security contexts?* uses the auxiliary functions defined above and returns true if mixed content should be blocked in the context provided.

Note how the analysis of the chain is performed only for `data:` URIs: this is a more specific behavior than the one proposed by the algorithm (lines 3-6 of Snippet 3.8.1). This implementation choice was due to the fact that an invariant will be defined to cover the remaining cases, allowing for more precise feedback on the potential security violations detected. The invariant will be presented later.

```
(define-fun does-settings-prohibits-mixed-security-contexts ((origin-url
↪  String) (document-url-opt (Option String)) (type ResourceType) (ancestors
↪  (List String))) Bool
(forall ((document-url String))
    (or
        (is-origin-potentially-trustworthy document-url)
        (is-blob-url-potentially-trustworthy url)
        (is-data-origin-potentially-trustworthy document-url ancestors))))
```

Snippet 3.8.5: SMT-LIB implementation of "*Does settings prohibits mixed security contexts?*"

3.9 Upgrade a mixed content request to a potentially trustworthy URL, if appropriate

The specification proposes an approach to increase the security of a page in an opportunistic way. If upgradeable Mixed Content is requested by the page, the algorithm "*Upgrade a mixed content request to a potentially trustworthy URL, if appropriate*" modifies the requested URL and upgrades the protocol to HTTPS before the filtering takes place. This will have the effect of avoiding the blocking of the upgradeable requests by *Should fetching request be blocked as mixed content?*

As the specification notes, this could result in unwanted consequences as the protocol upgrade may result in the loading of a resource that the developer did not intend. If the original resource requested is an innocuous image served over `http://example.com/image.png`, after the auto-upgrading a different resource may be targeted without the developer or user's explicit consent: it could be, for example that `https://example.com/image.png` redirects to a tracking site.

In such cases, fetching the content can result in unwanted privacy issues. The specification, however, states that such an event is expected to be exceedingly rare in practice.

On the other end, the risk of auto-upgrading blockable content is higher as this could result in the of loading an out-of-date and vulnerable JavaScript library. For this reason, auto-upgrading is only performed on upgradeable content.

```
If one or more of the following conditions is met, return without modifying
↪  request:
    request's URL is a potentially trustworthy URL.
    request's URL's host is an IP address.
    Does settings prohibit mixed security contexts? returns "Does Not Restrict
      ↪  Mixed Security Contents" when applied to request's client.
    request's destination is not "image", "audio", or "video".
    request's destination is "image" and request's initiator is "imageset".
Return blocked.
```

Snippet 3.9.1: Algorithm "*Upgrade a mixed content request to a potentially trustworthy URL, if appropriate*"

3.9.1 Modeling the type of content: blockable vs upgradable

As observed in subsection 2.3.5, the specification proposes that the upgrade of mixed content should be integrated into the Fetch algorithm. This means that as soon as the request is issued (e.g. by the deliberate `fetch()` call, or by internal browser functions called when subresources are included in the page) the check for upgradeable content is performed and possibly, the scheme is upgraded.

Due to the instrumentation approach adopted for this project (i.e. a web extension), it is not possible to hook into such functions. The methods `fetch()` and `new XMLHttpRequest()` have been instrumented, but unfortunately, with browser extensions it's impossible to instrument the internal browser functions used for fetching subresources.

Moreover, the callbacks used to intercept all web requests sent happen when the auto-upgrade has already been performed.

As a consequence, it would be possible to monitor auto-upgrade only for requests made via `fetch()` and `new XMLHttpRequest()`, but this is only a minor part of the total number of web requests sent.

As a consequence, we do not actively monitor auto-upgrading, but we detect if it was performed by looking at the network requests. In particular, we expect that no upgradeable mixed content request will be sent.

If an upgradeable mixed content request is detected, we know that there is a problem with the (lack of) auto-upgrading.

To distinguish between upgradeable and blockable requests we define an auxiliary function that analyzes the request type and checks if it is about upgradeable content.

In particular, `type-media` is assigned by the browser to the following MIME types:

- audio/wave
- audio/ogg
- audio/mpeg
- audio/flac
- audio/3gpp
- audio/aac
- audio/mpeg
- audio/mp4
- video/3gpp
- video/3gpp2
- video/3gp2
- video/ogg
- video/mpeg
- video/mp4
- video/quicktime

Similarly, `type-image` is assigned to the MIME types:

- image/gif
- image/jpeg
- image/png
- image/svg+xml
- image/webp
- image/apng
- image/avif

If the MIME type of the request is any of the types above, the function `is-mixed-content-upgradeable` will return true.

```
(define-fun is-mixed-content-upgradeable  ((type ResourceType)) Bool
(or
    (= type type-image)
    (= type type-media)))
```

Snippet 3.9.2: SMT-LIB function to categorize mixed-content

3.10 Should fetching request be blocked as mixed content?

The core of the Mixed Content mechanism is web request filtering. This is done by the algorithm "*Should fetching request be blocked as mixed content?*" The logic of this algorithm is simple: if the context prohibits mixed content then the URL has to be potentially trustworthy. If the URL is not potentially trustworthy then the request is mixed content and should be blocked.

The specification allows the user agent to exclude a set of origins from filtering (line 4 of Snippet 3.10.1). This way the user can define a whitelist of web pages where filtering is disabled, but it's suggested that such a choice should be presented to users with careful communication of the risk involved.

Moreover, the filtering algorithm excludes top-level navigation from Mixed Content checks (line 5 of Snippet 3.10.1). Not applying the filtering is based on the fact that the user might have entered the URL themselves or clicked on a link. The browser's address bar and security indicators provide feedback about the security of the top-level navigation. If a user navigates to an insecure page, the browser can indicate that the page is not secure. In this case, there is no threat of compromising other securely delivered content, as top-level navigation doesn't have a "parent page" to compromise in this manner.

```
1   Return allowed if one or more of the following conditions are met:
2       Does settings prohibit mixed security contexts? returns "Does Not Restrict
        ↪  Mixed Security Contexts" when applied to request's client.
3       Request's URL is a potentially trustworthy URL.
4       The user agent has been instructed to allow mixed content.
5       Request's destination is 'document', and request's target browsing context
        ↪  has no parent browsing context.
6   Return blocked
```

Snippet 3.10.1: Algorithm "*Should fetching request be blocked as mixed content?*"

3.10.1 Modeling the algorithm "*Should fetching request be blocked as mixed content?*"

As discussed in the previous subsection, we are interested in detecting mixed content requests and categorize them either as upgradeable or blockable. This way it is possible to monitor the behavior of the protocol upgrading. To do so, we define two invariants to model the algorithm "*Should fetching request be blocked as mixed content?*" The first invariant models blockable requests filtering. The second invariant models upgradeable requests filtering.

Modeling the filtering of blockable mixed content

To model line 4 of the algorithm (allow mixed content requests targeting the top level), it is possible to look at the request type and if it is `type-main_frame` we know that the request target is the top-level.

Another type of request that should not be filtered are requests for popups (that is top-level navigation on a new tab). For popups, the request type is `type-sub_frame` (the same as iframes), but in the request, there will be no document URL field. This allows us to identify the request for popups and prevent filtering.

$$
\begin{aligned}
&\textsc{blockable-mixed-content-filtered}(tr) := \\
&\quad \forall \mathit{t1} \forall \mathit{url} \forall \mathit{method} \forall \mathit{type} \forall \mathit{origin\text{-}url} \forall \mathit{doc\text{-}url} \forall \mathit{ancestors} \forall \mathit{r\text{-}hds} \forall \mathit{r\text{-}bdy} \forall \mathit{id} \\
&\quad (\boldsymbol{\mathit{net\text{-}request}}(\mathit{id}, \mathit{url}, \mathit{method}, \mathit{type}, \mathit{origin\text{-}url}, \mathit{doc\text{-}url}, \mathit{ancestors}, \mathit{r\text{-}hds}, \mathit{r\text{-}bdy}) \\
&\quad @_{tr}\mathit{t1} \wedge \\
&\quad \neg(\mathit{type} = \mathit{type\text{-}main\_frame}) \wedge \\
&\quad \neg(\mathit{type} = \mathit{type\text{-}sub\_frame} \wedge document\_url \neq \emptyset) \wedge \\
&\quad \neg \mathit{is\text{-}mixed\text{-}content\text{-}upgradeable}(\mathit{type}) \wedge \\
&\quad \mathit{does\text{-}settings\text{-}prohibits\text{-}mixed\text{-}security\text{-}contexts}(\mathit{origin\text{-}url}, \mathit{doc\text{-}url}, \mathit{type}, \mathit{ancestors}) \\
&\\
&\quad \Rightarrow \mathit{is\text{-}url\text{-}potentially\text{-}trustworthy}(\mathit{url}) \vee \\
&\qquad\quad \mathit{is\text{-}blob\text{-}url\text{-}potentially\text{-}trustworthy}(\mathit{url})))
\end{aligned}
$$

Snippet 3.10.2: Formula for blockable mixed content filtering

The formula presented above can be encoded in the following SMT-LIB invariant.

```
(define-fun blockable-mixed-content-filtered ((tr (List Action))) Bool
(forall ((t1 Int)(url String) (method RequestMethod) (type ResourceType)
↪  (origin-url String) (document-url (Option String)) (frame-ancestors (List
↪  String)) (request-headers RequestHeaders) (request-body (Option String))
↪  (id String))
   (=>
   (and
       (at t1 tr (net-request id url method type origin-url document-url
       ↪  frame-ancestors request-headers request-body))
```

```
        (not (= type type-main_frame))
        (not
            (and
                (= type type-sub_frame)
                (= document-url-opt (some document-url))))
        (not (is-mixed-content-upgradeable type))
        (does-settings-prohibits-mixed-security-contexts origin-url
        ↪  document-url type frame-ancestors))
    (or
        (is-url-potentially-trustworthy url)
        (is-blob-url-potentially-trustworthy url)))))
```

Snippet 3.10.3: SMT-LIB encoding of the invariant for blockable mixed content filtering

Modeling the filtering of upgradable mixed content

The invariant to model the filtering of upgradeable mixed content is very similar to the invariant on blockable content. The main difference is that we do not negate the result of the function `is-mixed-content-upgradeable`.

$$
\begin{aligned}
&\textsc{upgradeable-mixed-content-filtered}(tr) := \\
&\quad \forall \mathit{t1} \forall \mathit{url} \forall \mathit{method} \forall \mathit{type} \forall \mathit{origin\text{-}url} \forall \mathit{doc\text{-}url} \forall \mathit{ancestors} \forall \mathit{r\text{-}hds} \forall \mathit{r\text{-}bdy} \forall \mathit{id} \\
&\quad (\boldsymbol{net\text{-}request}(\mathit{id}, \mathit{url}, \mathit{method}, \mathit{type}, \mathit{origin\text{-}url}, \mathit{doc\text{-}url}, \mathit{ancestors}, \mathit{r\text{-}hds}, \mathit{r\text{-}bdy}) \\
&\quad @_{tr}\mathit{t1} \land \\
&\quad \neg()\mathit{type} = \mathit{type\text{-}main\_frame}) \land \\
&\quad \neg(\mathit{type} = \mathit{type\text{-}sub\_frame} \land document\_url \neq \emptyset) \land \\
&\quad \mathit{is\text{-}mixed\text{-}content\text{-}upgradeable}(\mathit{type}) \land \\
&\quad \mathit{does\text{-}settings\text{-}prohibits\text{-}mixed\text{-}security\text{-}contexts}(\mathit{origin\text{-}url}, \mathit{doc\text{-}url}, \mathit{type}, \mathit{ancestors}) \\
&\\
&\quad \Rightarrow \mathit{is\text{-}url\text{-}potentially\text{-}trustworthy}(\mathit{url}) \lor \\
&\qquad \mathit{is\text{-}blob\text{-}url\text{-}potentially\text{-}trustworthy}(\mathit{url})))
\end{aligned}
$$

Snippet 3.10.4: Formula for upgradeable mixed content filtering

```
(define-fun upgradeable-mixed-content-filtered ((tr (List Action))) Bool
(forall ((t1 Int)(url String) (method RequestMethod) (type ResourceType)
↪  (origin-url String) (document-url (Option String)) (frame-ancestors (List
↪  String)) (request-headers RequestHeaders) (request-body (Option String))
↪  (id String))
    (=>
    (and
        (at t1 tr (net-request id url method type origin-url document-url
        ↪  frame-ancestors request-headers request-body))
        (not (= type type-main_frame))
        (not
            (and
                (= type type-sub_frame)
                (= document-url-opt (some document-url))))
        (is-mixed-content-upgradeable type)
        (does-settings-prohibits-mixed-security-contexts origin-url
        ↪  document-url type frame-ancestors))
    (or
        (is-url-potentially-trustworthy url)
        (is-blob-url-potentially-trustworthy url)))))
```

Snippet 3.10.5: SMT-LIB encoding of the invariant for upgradeable mixed content filtering

3.11 More mixed content algorithms and invariants

In this section, we analyze another algorithm proposed by the specification and one more invariant proposed by us.

3.11.1 Should response to request be blocked as mixed content?

Even if a request was not filtered, it may be relevant to block the reply received from the server as some relevant information is available only *a posteriori*. One case would be if the TLS authentication of the server (when communicating via HTTPS or WSS) fails due to an expired or invalid certificate. The algorithm *Should response to request be blocked as mixed content?* acts in the very same way as the request filter, this time considering information related to the response received.

```
Return allowed if one or more of the following conditions are met:
    Does settings prohibit mixed security contexts? returns "Does Not Restrict
    ↪  Mixed Security Contexts" when applied to request's client.
    request's URL is a potentially trustworthy URL.
    The user agent has been instructed to allow mixed content
    request's destination is "document", and request's target browsing context
    ↪  has no parent browsing context.
Return blocked.
```

Snippet 3.11.1: Algorithm "*Should response to request be blocked as mixed content?*"

Considerations on modeling the algorithm

There is a series of considerations to make when considering how to model the algorithm presented above.

When browsers do detect an error in the TLS handshake (e.g. an invalid certificate), the connection will not be established and no HTTP message will be exchanged with the server. This means that there is no response to be filtered (as no request was sent). This is true for all the possible certificate misconfigurations (expired, self-signed, invalid, revoked, certificate from untrusted CA and weak cipher suites), and when this happens a TLS error is shown in the browser console (as well as in the network panel). If the content of a subresource had to be fetched via the connection that failed, it wouldn't be populated, and a visual error message would be presented if the subresource was an iframe, image, or video.

In general such errors are handled by the browsers with checks performed within TLS-related methods, and not by the mixed content implementation.

One more situation to consider is the possibility of receiving a redirect to HTTP from a server that was contacted via HTTPS. In such a case, the handshake succeeded, and both the request and response (e.g. `301 Moved Permanently`) were sent over a secure channel.

Then the web browser will send a new request to the redirected URL. If such URL is not potentially trustworthy then the request will be blocked by the algorithm *Should fetching request be blocked as mixed content?*.

Once again there is no need to implement filtering on responses.

In fact, web browsers do not implement mixed content filtering on responses as the checks proposed by the algorithm "*Should response to request be blocked as mixed content?*" are already performed by other security mechanisms (e.g. TLS implementation).

We therefore decided that there is no need to model such an algorithm with an invariant.

3.11.2 Model the filtering of content in nested contexts

As anticipated in the previous section, we decided to have a dedicated invariant that models the filtering of mixed content in nested contexts. This decision was based on the consideration that nested contexts do represent one of the most difficult aspects of the Mixed Content specification. Moreover, dealing with mixed content presence in framed pages has been quite challenging for browser vendors, with several bugs and security violations reported in the past [5] [6] [4] [10].

Having a dedicated invariant to detect mixed content violation in nested contexts allows to evaluate the correct browser implementation of functions that compute origin trustworthiness in framed pages.

In Snippet 3.11.2 the auxiliary function that models lines 4-6 of the algorithm *Does settings prohibit mixed security contexts?* presented in 3.8.

Instead of checking for potential trustworthiness, only a check for the `https:` scheme is performed. This simpler check is in line with the implementation of web browsers [3] and allows for better performances compared to using the full regex of Snippet 3.7.2. We verified that in practice this is possible without loss of generality.

Snippet 3.11.2: SMT-LIB function to identify if the ancestor chain permits mixed content

```
(define-fun-rec ancestors-permit-mixed-content ((ancestors (List String))) Bool
  (match ancestors
    ((nil true)
      ((insert a acs)
        (and
          (not (str.in.re a
            (re.++
              (str.to.re "https://")
```

```
                (re.* re.allchar))))
            (ancestors-permit-mixed-content acs))))))
```

With the auxiliary function defined above it is possible to model an invariant that checks if a mixed content request should be blocked as the nested context is composed of potentially trustworthy origins.

Once again we exclude top-level navigation from the filtered requests.

The invariant is equivalent to the formulation: “If the ancestor chain disallows mixed content, then requests must have a potentially trustworthy URL”

$$
\begin{aligned}
&\textsc{mixed-content-filtered-in-nested-contexts}(tr) := \\
&\quad \forall \mathit{t1} \forall \mathit{url} \forall \mathit{method} \forall \mathit{type} \forall \mathit{origin\text{-}url} \forall \mathit{doc\text{-}url} \forall \mathit{ancestors} \forall \mathit{r\text{-}hds} \forall \mathit{r\text{-}bdy} \forall \mathit{id} \\
&\quad (\boldsymbol{\mathit{net\text{-}request}}(\mathit{id}, \mathit{url}, \mathit{method}, \mathit{type}, \mathit{origin\text{-}url}, \mathit{doc\text{-}url}, \mathit{ancestors}, \mathit{r\text{-}hds}, \mathit{r\text{-}bdy}) \\
&\qquad @_{tr} \mathit{t1} \land \\
&\qquad \neg(\mathit{ancestors\text{-}permit\text{-}mixed\text{-}content}(\mathit{ancestors})) \land \\
&\qquad \neg(\mathit{type} = \mathit{main\_frame}) \\
&\quad \Rightarrow \mathit{is\text{-}url\text{-}potentially\text{-}trustworthy}(\mathit{url}))
\end{aligned}
$$

Snippet 3.11.3: Formula for mixed content filtering in nested contexts

```
(define-fun mixed-content-filtered-in-nested-frames ((tr (List Action))) Bool
(forall ((t1 Int) (url String) (method RequestMethod) (type ResourceType)
↪  (origin-url String) (document-url (Option String)) (frame-ancestors (List
↪  String)) (request-headers RequestHeaders) (request-body (Option String))
↪  (id String))
    (=>
        (and
            (at t1 tr (net-request id url method type origin-url document-url
            ↪  frame-ancestors request-headers request-body))
            (not (ancestors-permit-mixed-content frame-ancestors))
            (not (= type type-main_frame)))
        (or
            (is-url-potentially-trustworthy url)
            (is-blob-url-potentially-trustworthy url)))))
```

Snippet 3.11.4: SMT-LIB encoding of the invariant for mixed content filtering in nested contexts

html	7445
encoding	1342
referrer-policy	1301
content-security-policy	821
fetch	775
svg	723
websockets	717
editing	601
dom	488
IndexedDB	464
xhr	415
mathml	405
navigation-api	395
workers	321
webvtt	320
service-workers	299
webdriver	279
streams	253
webaudio	250
wasm	246
bluetooth	230
speculation-rules	209
shadow-dom	200
upgrade-insecure-requests	197
WebCryptoAPI	183
webrtc	168
mixed-content	163
pointerevents	156
webmessaging	154
infrastructure	150
webxr	137
custom-elements	137
web-animations	136
scroll-animations	127
webcodecs	122
resource-timing	122
scheduler	108
encrypted-media	106
client-hints	104
eventsource	100
FileAPI	90
trusted-types	88
layout-instability	81
web-locks	78
media-source	78
permissions-policy	77
performance-timeline	76
fullscreen	76
url	75
encoding-detection	75
selection	71
intersection-observer	69
cookies	69
user-timing	62
largest-contentful-paint	61
signed-exchange	60
cookie-store	60
compression	59
serial	58
webidl	55
forced-colors-mode	55
event-timing	54
paint-timing	53
navigation-timing	53
mediacapture-streams	52
preload	51
webnn	50
feature-policy	50
webusb	49
webstorage	49
fs	48
loading	47
clipboard-apis	47
element-timing	46
portals	44
uievents	43
quirks	39
animation-worklet	39
webtransport	37
webauthn	36
js	35
document-policy	34
storage	32
import-maps	32
accessibility	32
compat	31
compute-pressure	30
web-bundle	29
focus	29
domparsing	29
soft-navigation-heuristics	28
cors	27
payment-request	26
shape-detection	25
webrtc-encoded-transform	24
mediacapture-image	24
domxpath	24
credential-management	24
reporting	23
worklets	22
density-size-correction	22
orientation-event	21
input-events	21
inert	20
requestidlecallback	19
longtask-timing	19
visual-viewport	18
storage-access-api	18
long-animation-frame	18
hr-time	18
screen-wake-lock	17
resize-observer	17
notifications	17
mediacapture-record	17
js-self-profiling	17
battery-status	17
urlpattern	16
orientation-sensor	16
measure-memory	16
geolocation-API	16
screen-orientation	15
old-tests	15
browsing-topics	15
beacon	15
web-share	14
imagebitmap-renderingcontext	14
background-fetch	14
secure-payment-confirmation	13
presentation-api	13
picture-in-picture	13
payment-handler	13
console	13
scroll-to-text-fragment	12
is-input-pending	12
font-access	12
accelerometer	12
web-nfc	11
speech-api	11
page-visibility	11
network-error-logging	11
idle-detection	11
geolocation-sensor	11
server-timing	10
screen-capture	10
sanitizer-api	10
pending-beacon	10
mediacapture-insertable-streams	10
media-capabilities	10
magnetometer	10
gyroscope	10
audio-output	10
ambient-light	10
webrtc-extensions	9
touch-events	9
permissions	9
webgl	8
video-rvfc	8
subapps	8
secure-contexts	8
keyboard-map	8
document-picture-in-picture	8
webvr	7
remote-playback	7
pointerlock	7
mediasession	7
mediacapture-fromelement	7
keyboard-lock	7
fledge	7
x-frame-options	6
webrtc-stats	6
shared-storage	6
gamepad	6
file-system-access	6
content-dpr	6
close-watcher	6
badging	6
webrtc-svc	5
wai-aria	5
push-api	5
lifecycle	5
delegated-ink	5
content-index	5
clear-site-data	5
webrtc-identity	4
vibration	4
ua-client-hints	4
proximity	4
payment-method-basic-card	4
mimesniff	4
merchant-validation	4
device-memory	4
virtual-keyboard	3
trust-tokens	3
top-level-storage-access-api	3
timing-entrytypes-registry	3
screen-details	3
periodic-background-sync	3
parakeet	3
netinfo	3
mst-content-hint	3
generic-sensor	3
autoplay-policy-detection	3
acid	3
webrtc-priority	2
webhid	2
savedata	2
png	2
permissions-revoke	2
permissions-request	2
managed	2
intervention-reporting	2
installedapp	2
html-media-capture	2
direct-sockets	2
deprecation-reporting	2
contenteditable	2
background-sync	2
apng	2
window-placement	1
webrtc-ice	1
web-otp	1
webmidi	1
subresource-integrity	1
private-click-measurement	1
payment-method-id	1
page-lifecycle	1
media-playback-quality	1
mediacapture-region	1
mediacapture-handle	1
mediacapture-extensions	1
input-device-capabilities	1
eyedropper	1
entries-api	1
ecmascript	1
custom-state-pseudo-class	1
contacts	1
avif	1
accname	1

Table 3.2: Considered WPT tests. Total: 24855 tests. WPT Version: `d888ebb`

4 Results

4.1 Results for Webkit

At the time of writing, not all the tests have been run on WebKit. About 6,000 tests have been run on a total of 24,000.

In Table 4.1 the tests that reported SAT for some mixed-content invariant.

The mixed-content tests of the folder `tentative/autoupgrade` do all return SAT for `upgradeable-mixed-content-filtered`, as well as a test in `upgrade-insecure-requests`. This is because WebKit does not comply with the current version of the Mixed Content specification. As mentioned in subsection 2.3.4, previous versions of the specification allowed vendors to choose between blocking upgradeable content or allowing it and showing an in-between security indicator.

WebKit has not been updated to perform auto-upgrading, so mixed content requests for upgradeable content are sent over the network.

Folder	Test	SAT Invariant
upgrade-insecure-requests	`gen/srcdoc-inherit.meta/unset/img-tag.https.html`	mixed-content-filtered-in-nested-frames
mixed-content	`nested-iframes.window.html`	
	`csp.https.window.html`	blockable-mixed-content-filtered
	`gen/top.http-rp/opt-in/beacon.https.html`	
	`gen/top.meta/opt-in/beacon.https.html`	
	`gen/top.meta/unset/beacon.https.html`	
	`tentative/autoupgrades/audio-upgrade.https.sub.html`	upgradeable-mixed-content-filtered
upgrade-insecure-requests	`tentative/autoupgrades/video-upgrade.https.sub.html`	
	`tentative/autoupgrades/image-upgrade.https.sub.html`	
	`upgrade-insecure-requests/gen/top.meta/unset/img-tag.https.html`	

Table 4.1: Tests that returned SAT for some invariant in WebKit

In the following, we explore a powerful attack that was found thanks to the SAT reported on the tests `mixed-content/csp.https.window.html` and `mixed-content/nested-i` frames.window.html.

4.1.1 sandbox attribute bypass Mixed Content restrictions in WebKit

Our framework reported a violation for `blockable-mixed-content-filtered` with the WPT test mixed-content/csp.https.window.html. The test consists of a webpage that presents the CSP's `sandbox allow-scripts` directive. The directive has the same effects as the
`sandbox allow-scripts` attribute in a frame (block form submission, disable API, and set the origin to `null`) applying restrictions on the whole webpage.

By analyzing the trace we observed how the webpage is loaded via HTTPS, so mixed content should be prohibited. Nevertheless, a fetch request targeting an HTTP endpoint was not blocked.

Upon further analysis, we identified how the cause of the filtering bypass was that the CSP directive was effectively setting the origin variable to `null`. As the `null` origin is not potentially trustworthy, mixed content checks were not performed.

4.1.2 Framed pages bypass Mixed Content restrictions in WebKit

Another violation was reported for `mixed-content-filtered-in-nested-frames` with the WPT test wpt/mixed-content/nested-iframes.window.html. In the browser trace, we observed a fetch request to an insecure endpoint coming from a frame whose origin was potentially trustworthy. After some investigation, we concluded that WebKit was performing the mixed content checks incorrectly: secure pages embedded from insecure origins were not considered potentially trustworthy, and therefore mixed content checks were not performed for most of the content. Only scripts, stylesheets, and insecure WebSocket requests were still blocked.

4.1.3 The attack: CVE-2023-38592

It is possible to combine the effects of the two security violations of subsection 4.1.1 and subsection 4.1.2 by framing a page with the `sandbox` attribute inside an HTTP top level.

This way all the mixed content requests (for both upgradeable and blockable content) are not blocked.

This dangerous scenario is exploitable by an attacker: if he finds out that https://bank.com includes a mixed content script he can frame it in http://attacker.com and when a user visits the page the request will be sent, with a network attacker capable of intercepting it and obtaining arbitrary code execution on the page https://bank.com.

We reported the problem to Apple via the WebKit Bugzilla platform. The following snippet is our message for the disclosure of the vulnerability.

```
Safari version: 16.5 (16615.2.9.11.6, 16615)
Safari does not enforce mixed content checks for active content in secure pages
↪  embedded from insecure origins.
```

```
An attacker at http://evil.com can frame a page from a secure origin
 ↪  (https://bank.com) that attempts to load active mixed content.
The mixed content which would normally be blocked is now allowed and can be
 ↪  intercepted and tampered with by a network attacker.
This can potentially lead to attacker-controlled code in an https page. Additional
 ↪  capabilities are provided by embedding websites in an iframe using the sandbox
 ↪  attribute.
When embedding the secure page from an iframe with the sandbox attribute, all mixed
 ↪  content is allowed.
However, without the sandbox attribute, <script> <style> and all active mixed content
 ↪  loaded from insecure WebSockets (ws://) is blocked.
Nonetheless, an attacker can still tamper with all of the following types of active
 ↪  mixed content:

- <iframe src="...">
- fetch("...")
- XMLHttpRequest()
- <object data="...">
- <form action="...">
- CSS @font-face {... src: url("...")}

According to the W3C's Mixed Content specification
 ↪  (https://www.w3.org/TR/mixed-content), all active mixed content included from a
 ↪  secure origin should be blocked.
This is the behavior of both Chrome and Firefox, but not Safari.
```

Details on CVE and impact

About one week after we disclosed the vulnerability, the WebKit Security team confirmed the problem and issued it with priority P1 ("Serious security issue" according to the WebKit bug prioritization guidelines [39]).

Two weeks after being confirmed, the issue was marked as resolved in the Bugzilla platform. Then two weeks later Apple published a series of security updates with the fix of the vulnerability and the CVE that was assigned to it: CVE-2023-38592.

The CVE vectors are reported in Table 4.2. The complexity of the attack is low, as the only requirement is to frame an HTTPS page from an HTTP one. User interaction is required as the user must visit the attacker's page. The scope is high as the attacker can tamper with all the mixed content requests.

Note that as the vulnerability was found in WebKit, it affects all the browsers that use it as an engine. This includes Safari, as well as all the browsers for iOS and iPadOS as Apple does not allow other engines to be used on its devices.

Other projects also use the WebKit engine, for example, the GNOME Web browser (formerly known as Epiphany) and WebKitGTK, the port of WebKit to GTK+. Within one week after the publication of the CVE, all the browsers above delivered a fix for the vulnerability.

The bug report submitted is not yet accessible to the public, but the entry on the bug tracker platform should be made open in the next weeks.

Base Score	Impact Subscore	Exploitability Subscore	NVD Published Date
8.8 High	5.9	2.8	07/28/2023

Attack vector	Attack complexity	Privileges Required	User Interaction
Network	Low	None	Required

Scope	Confidentiality Impact	Integrity Impact	Availability Impact
High	High	High	High

Table 4.2: Details of the CVSS vector attributes for CVE-2023-38592

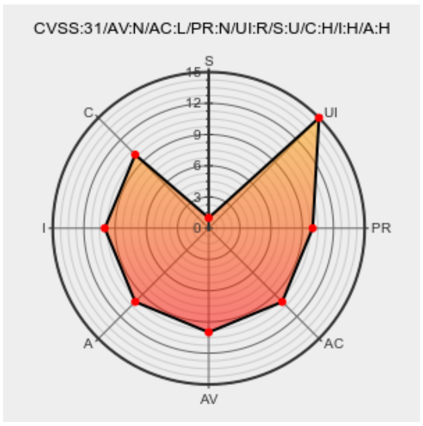

Figure 4.1: Circular heatmap of the CVSS vector attributes for CVE-2023-38592

Apple fix

The fix was delivered with the commit `bc09b6f` [11]. The commit message can be seen in Snippet 4.1.1.

According to Apple, two root causes enabled the bypass of mixed content checks:

- **Framing security violation**: the mixed content checks were not performed on the whole frame ancestors chain, only the top frame and the request URL were considered.
- **Sandboxing security violation**: the mixed content checks were not performed on sandboxed documents and frames as information about the origin was lost (as it was set to `null`).

```
It is possible to bypass mixed content restrictions on pages which are
framed. There are two issues here: secure frames embedded in
insecure frames can bypass and frames that are sandboxed can bypass.

In the former case, we are only checking for mixed content in the
frame making the request as well as the top frame. So if an insecure frame
embeds a secure frame, that secure frame could then embed an insecure frame and
make mixed content requests without being blocked since the middle frame
is not checked against the URL for mixed content.
```

```
For the latter case we check whether or not the security origin
of the requestor has an "https" protocol. The problem is sandboxed iframes
are given an opaque origin which does not have the "https" protocol (or
any protocol for that matter) and so we were skipping the mixed content check.
...
```

Snippet 4.1.1: Commit message for the fix of CVE-2023-38592

To fix the first problem, Apple changed the mixed content filtering to perform checks on the whole frame ancestors chain. In Snippet 4.1.2, the new function that performs the check is shown. A loop is performed to recursively ascend the chain: the method `isMixedContent()` is called passing the frame that is currently analyzed and the URL of the request. If the method returns `true`, the request is mixed content, and it will be blocked.

The request is not blocked only if the method returns `false` for all the frames in the chain, including the top level.

```cpp
static bool foundMixedContentInFrameTree(const LocalFrame& frame, const URL&
↪ url)
{
    auto* document = frame.document();

    while (document) {
        RELEASE_ASSERT_WITH_MESSAGE(document->frame(),
↪ "An unparented document tried to load or run content with url: %s",
↪ url.string().utf8().data());

        if (isMixedContent(*document, url))
            return true;

        auto* frame = document->frame();
        if (frame->isMainFrame())
            break;

        auto* abstractParentFrame = frame->tree().parent();
        RELEASE_ASSERT_WITH_MESSAGE(abstractParentFrame,
↪ "Should never have a parentless non main frame");
        if (auto* parentFrame =
↪ dynamicDowncast<LocalFrame>(abstractParentFrame))
            document = parentFrame->document();
    }

    return false;
}
```

Snippet 4.1.2: How Apple fixed the framing security violation.

The second problem was fixed by changing the mixed content checks to consider the real origin of the frame. The function that performs the check is shown in Snippet 4.1.3.

Previously the method `document.securityOrigin().protocol()` was used to get such information, but if the `sandbox` attribute is used, the method returns `null` as origin protocol.

The fix adds a check to see if `document.securityOrigin()` returns an opaque origin, and if so, the correct value for the origin is obtained with a call to `document.url()`.

```
static bool isMixedContent(const Document& document, const URL& url)
{
    // sandboxed iframes have an opaque origin so we should perform the mixed
↪   content check considering the origin
    // the iframe would have had if it were not sandboxed.
    if (document.securityOrigin().protocol() == "https"_s ||
↪   (document.securityOrigin().isOpaque() &&
↪   document.url().protocolIs("https"_s)))
        return !SecurityOrigin::isSecure(url);

    return false;
}
```

Snippet 4.1.3: How Apple fixed sandboxing security violation.

After Safari was updated, we ran again our framework on the WPT tests that previously reported SAT. Those tests now return UNSAT, as the insecure web requests are now blocked.

4.1.4 Mixed content beacon endpoints not filtered in WebKit

In Table 4.1 it is possible to see how a set of tests on bacon endpoints is giving SAT for the invariant `blockable-mixed-content-filtered`.

A beacon request is a special type of request that is sent by the browser to an endpoint when the user clicks on a link of an anchor element that presents the `ping` attribute. For example, if the following anchor is clicked, a beacon request will be sent:

`<a href="https://test.com" ping="https://example.com/ping">Click me</a>`.

When the user clicks on the link the browser will send a POST request to `https://example.com/ping`. In this case, the endpoint URL is potentially trustworthy. If on the other hand, the endpoint URL was not potentially trustworthy, the request is mixed content, and it should be blocked (as we noted in the previous chapters, beacon endpoint requests are considered blockable).

Let's consider now one of the tests that produced SAT: `mixed-content/gen/top.meta/unset/beacon.https.html`. The test attempts to send various beacon requests to endpoints that are not potentially trustworthy. This includes both same-origin and cross-origin endpoints, as well as endpoints that are secure at first but then downgraded after a server redirect. All those subtests do result in SAT for the invariant `blockable-mixed-content-filtered`. After studying the problem, we understood that WebKit does not perform mixed content checks for beacon requests.

We reported this problem to Apple, and we are waiting for confirmation.

4.2 Results for Firefox

At the time of writing all the tests have been run on Firefox (the full list is presented in Table 3.2).

The tests that returned SAT for some invariants are reported on Table 4.3.

Similarly to WebKit, the tests of the folder `tentative/autoupgrade` do all return SAT for `upgradeable-mixed-content-filtered`. This is caused by the same reason: Firefox did not comply yet with the current version of the Mixed Content specification, as protocol upgrading is not performed for upgradeable requests.

On the other hand, contrary to WebKit, Firefox is starting to implement the new version of the specification: on Firefox Experimental (the nightly build) protocol upgrading is performed for image and video types.

The developers are slowly adding support for protocol upgrading on the remaining types of content as well. For the implementation of the new specification, Firefox chose a conservative approach by giving time to web developers to update their websites and serve the content via HTTPS.

4.2.1 Problem with WebSockets in Firefox

There is a set of tests about WebSocket connections that return SAT for the invariant `blockable-mixed-content-filtered`. The tests are looking at a specific scenario: a WebSocket request sent from a Worker using an insecure protocol: `ws:`. The worker is created from a secure page, so its origin is potentially trustworthy. As a consequence we expect the request to be blocked as mixed content, but it is not.

We investigated the problem and found out that Firefox is implementing incorrectly the filtering for WebSocket requests. In particular, filtering is not performed if the origins are:

- **`blob:` origins**: if the insecure WebSocket request is sent from a `blob:` origin, it is not filtered.
- **`data:` URI Worker origins**: if a Worker is created from a `data:` URI and the WebSocket request is sent from it, it will not be filtered.

On Snippet 4.2.1 the message that we sent to Mozilla for the disclosure of the problem. We received a reply from the developers within a day, and they confirmed the problem.

A fix has been already merged in the Firefox codebase, and it will be released in the next weeks.

```
As part of a research project at TU Wien, we observed an inconsistency on Firefox
↪  affecting websockets and mixed content policy restrictions.
In particular, connections to insecure WebSockets (ws://...) are not blocked by the
↪  mixed content policy when the WebSocket is created from a potentially trustworthy
↪  origin via blob: or by a Worker loaded via a data: URI.
We provide the following payloads to test this behavior:
1) To test a Worker loaded via data: URI, you can execute new
↪  Worker("data:text/javascript,new WebSocket('ws:example.com')");
   from a potentially trustworthy origin (e.g., https://example.com).
```

```
2) To test blob: URI, execute the script below from a potentially trustworthy origin
    window.open(URL.createObjectURL(
            new File([`<!DOCTYPE html><script>new
            ↪  WebSocket("ws://example.com");</script>`],
                'test.html',
                {type: 'text/html'})));

The Worker script (1) triggers a connection request in Firefox, while Chromium and
↪  Safari block it with a mixed-content exception.
Similarly, the blob example (2) does not cause any error on Firefox, but it is
↪  blocked on Chromium and Safari.
```

Snippet 4.2.1: Message for the disclosure of Firefox WebSocket filtering problem

4.3 Results for Chromium

All tests have been run in Chromium. No SAT was reported in any mixed content invariants.

Folder	Test	SAT Invariant
upgrade-insecure-requests	gen/srcdoc-inherit.meta/unset/img-tag.https.html	mixed-content-filtered-in-nested-frames
	gen/iframe-blank-inherit.meta/unset/img-tag.https.html	
	gen/worker-classic-data.meta/unset/websocket.https.html	blockable-mixed-content-filtered
	gen/worker-module-data.meta/upgrade/websocket.https.html	
	gen/worker-module-data.http-rp/upgrade/websocket.https.html	
	gen/worker-module-data.meta/unset/websocket.https.html	
	gen/worker-classic-data.meta/unset/websocket.https.html	
mixed-content	gen/worker-classic-data.meta/upgrade/websocket.https.html	
	gen/worker-module-data.meta/unset/websocket.https.html	
	gen/worker-module-data.http-rp/opt-in/websocket.https.html	
	gen/worker-module-data.meta/opt-in/websocket.https.html	
	gen/worker-classic-data.meta/opt-in/websocket.https.html	
	gen/worker-classic-data.http-rp/opt-in/websocket.https.html	
	tentative/autoupgrades/audio-upgrade.https.sub.html	upgradeable-mixed-content-filtered
	tentative/autoupgrades/video-upgrade.https.sub.html	
	tentative/autoupgrades/image-upgrade.https.sub.html	
	gen/top.meta/unset/img-tag.https.html	
	gen/top.meta/unset/audio-tag.https.html	
clear-site-data	resource.html	
upgrade-insecure-requests	gen/top.meta/unset/img-tag.https.html	
	gen/iframe-blank-inherit.meta/unset/img-tag.https.html	
	/upgrade-insecure-requests/gen/top.meta/unset/img-tag.https.html	

Table 4.3: Tests that returned SAT for some invariant in Firefox

4.4 Mixed Content in the wild

After having analyzed the behavior of the browsers and identified critical vulnerabilities in the implementation of mixed content filtering, we wanted to understand how many websites are currently including mixed content.

After some research, we determined that no recent work was done on this topic. Therefore, we decided to crawl the top 100K websites to analyze mixed content presence on the web.

4.4.1 Previous results

One of the most recent researches about mixed content presence on the web was performed in 2015 by P. Chen et al. with the paper "A Dangerous Mix: Large-scale analysis of mixed-content websites" [14].

The paper presents an analysis of mixed-content inclusions on Alexa's top 100,000 Internet domains. To crawl the web they used HtmlUnit, a headless browser capable of running JavaScript content. The browser does not render webpages per se but provides high-level Java API that can be used to inspect the content of fetched webpages.

From the top 100,000 list, they considered only websites having at least 200 pages served over HTTPS. This was done by using the Bing search engine API to get a list of pages belonging to the same origin. For instance, the querying for `site:example.com` will return pages hosted on `example.com` as well as subdomains. This way, from the initial list they obtained a subset of 18,526 sites. They then fetched a total of 481,656 HTTPS pages, scanning on average 26 HTTPS pages per website.

Their study shows that 7,980 (43%) websites presented at least one type of mixed content. In particular, they detected a total of 620,151 mixed-content requests for 191,456 different files. In total, the number of vulnerable HTTPS pages was 74,946.

They categorized the mixed content inclusions into the following categories:

- **Images**: 406,932 requests from 5,557 websites (30%).
- **iframes**: 25,362 requests from 2,593 websites (14%).
- **CSS**: 35,957 requests from 2,223 websites (12%).
- **JavaScript**: 150,179 from 4,816 websites (26%).

4.4.2 Crawling the web

To compare the results with the scan performed in 2015, we decided to crawl the top 100K websites as well. As Alexa's list was discontinued in May 2022, we obtained an up-to-date list from the Tranco project [37].

The list is research-oriented and consists of the top 1 million websites, obtained by averaging all available rankings over 30 days.

For the crawling, we used Puppeteer, a Node.js library that provides a high-level API to control a headless Chromium instance. Puppeteer allows to automate the crawling of websites, and it is used by many projects to perform web scraping.

To detect mixed content we used the DevTools protocol to identify mixed content error messages in the browser console. We therefore use information coming directly from the browser as it is less resource-intensive than having to manually scan the DOM and JavaScript code. The headless Chromium will render the webpage and execute the JavaScript code, essentially behaving the same as a normal execution under user navigation.

As we observed with the results of our automated testing framework, Chromium resulted in the browser with the best implementation of the Mixed Content policy (no violation was found). We can therefore assume that using the error messages shown in the console is a reliable way to detect mixed content inclusions as the number of false negatives due to a security flaw in the implementation should be negligible.

This approach permits easy categorization of the different types of mixed content inclusions, as the error messages are specific for each type.

Each website is scanned by visiting its homepage and then following all the links that point to the same domain. This process is repeated until all the pages of the website are visited or if the time limit of 5 minutes is reached. A fixed time of 5 seconds is used in between page visits, to avoid overloading the website with requests. This defines an upper limit of 60 pages that can be visited for each website.

We added a custom request header to inform the website that we are performing crawling for research purposes, including an email to contact in case of problems.

Using anchors to visit different pages

Once a specific domain is extracted from the list, we visit the homepage of the website. We then collect the anchors to create a list of other pages to scan. In particular, we consider only anchors that point to the same origin, and that do not contain a fragment part. We exclude pages that were already visited and pages that point to the same URL as the current page. The code at Snippet 4.4.1 shows the filtering that is performed to populate the crawling list.

```javascript
const filtered = allElements
    .filter((el) => el.localName === "a" && el.href) // Element is an anchor
    ↪  with an href.
    .filter((el) => typeof el.href == "string" && !el.href.includes("#")) //
    ↪  Element does not have fragment part
    .filter((el) => el.href !== location.href) // Link doesn't point to page's
    ↪  own URL.
    .filter((el) => {
        if (sameOrigin) {
            return new URL(location).origin === new URL(el.href).origin;
        }
        return true;
    })
    .map((a) => a.href);

```

Snippet 4.4.1: Populating crawling list using anchors present in the current document

Understanding type of mixed content

To identify and categorize mixed content requests we use the error messages that are shown in the browser console. Mixed content errors do include the substring “Mixed Content”. It is possible to categorize the mixed content requests into blockable and upgradeable content by looking at the error message. If the message includes “WAR”, then the message is informative to communicate that auto-upgrading was performed on an upgradeable request. If the message does not include “WAR”, then the request was blocked as mixed content. The message includes the URL of the request and the content type.

We store the full content of the message in one of two different files based on the mixed content type. Such files will be post-processed at the end of the crawl to obtain the final results.

In Snippet 4.4.2, the code used to obtain the mixed content request classification.

```
function isMixedContentError(errorMessage) {
    if (errorMessage.includes("Mixed Content")){
        if (errorMessage.includes("WAR")) {
            console.log(
                chalk.bgYellow.black("upgradeable-mixed-content:"),
                errorMessage
            );
            fs.appendFile(
                "./results/upgradeable.txt",
                errorMessage + "\n",
                function (err) {
                    if (err) throw err;
                });
        }
        else {
            console.log(
                chalk.bgRed.black("blockable-mixed-content:"),
                errorMessage
                );
                fs.appendFile(
                    "./results/blockable.txt",
                    errorMessage + "\n",
                    function (err) {
                        if (err) throw err;
                });
        }
    }
}
```

Snippet 4.4.2: Obtain mixed content request classification while crawling

4.4.3 Results of crawling

In total, 30 parallel Chromium instances were used, each scanning a subset of the top 100K websites. As the crawling process is CPU intensive, we used a machine with 32 cores and 40

GB of RAM. The available bandwidth was 1 Gbps. With such a configuration, the crawling took about ten days to complete.

The results were post-processed to obtain the final statistics.

We divide the analysis into two parts: we first consider the top 1000 websites, then we look at the full list of 100K websites and compare the results with the data from 2015 presented in the paper "A Dangerous Mix: Large-scale analysis of mixed-content websites".

Top 1000 websites

We consider now the most popular sites on the web. One preliminary assumption on this subset is that the complexity of pages should be higher on average compared to the full list. We may expect these websites to include many dynamic pages each with several external resources.

Another consideration could be that some websites in this subset could include older pages that may still include external resources loaded via HTTP. This aspect could be exacerbated by the fact that some websites do have many web pages that are not visited often, with most of the traffic concentrated on a few pages. A representative example of this is university websites that often include personal pages of professors and researchers.

On the other hand, we also expect that most of these websites are updated and maintained, as they are more popular and visited by many users. In particular, the websites of big companies should be updated more often, as the budget for the maintenance of the website is higher.

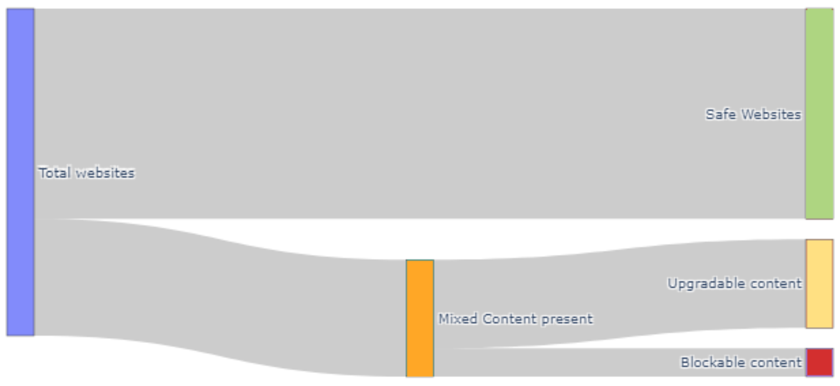

Figure 4.2: Websites from the top 1000 list that present Mixed Content

We consider first the number of websites that include mixed content per category. This can be visualized in Figure 4.2. In particular, we have that 357 websites (35.7%) include at least one type of mixed content. This is split into:

- **Websites with upgradeable mixed content**: 271 websites (27.1%).
- **Websites with blockable mixed content**: 86 websites (8.6%).

We now consider the requests that are mixed content based on the content type. This information can be visualized in Figure 4.3. A total of 48,605 requests were found to be mixed content. These were divided as follows:

- **Upgradeable requests**: 39,202 requests that have been upgraded.
  - **Element**(subresources): 39,153 requests.
  - **Image**(javascript fetch): 49 requests.
- **Blockable requests**: 9,380 requests have been blocked.
  - **Script**: 6,370 requests.
  - **Endpoint** (`fetch`): 1,470 requests.
  - **Favicon**: 1,020 requests.
  - **Stylesheet**: 187 requests.
  - **Frame**: 131 requests.
  - **Font**: 197 requests.
  - **Plugin resource** (`<object>`): 2 requests.
  - **Resource** (`<embed>`): 1 request.

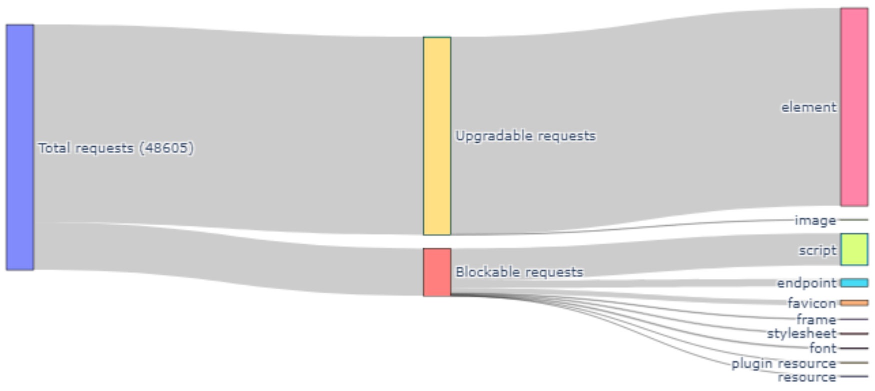

Figure 4.3: Mixed Content requests in the top 1000 websites

Top 100K websites

We now move to the full list of the top 100K websites. From the websites in this list, we expect a lower complexity of pages on average, this should translate into a lower number of external resources per page.

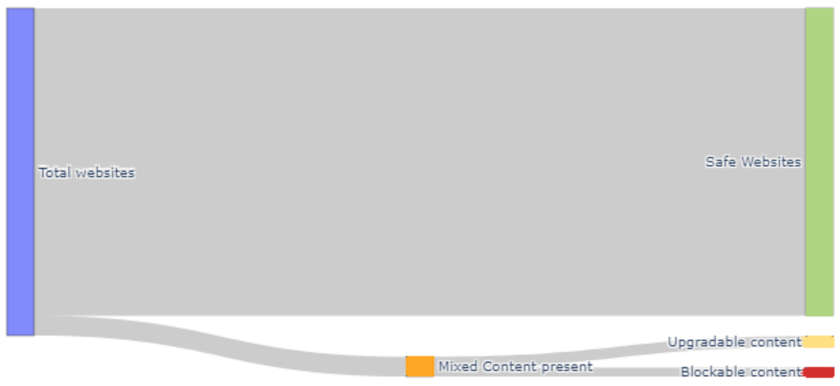

Figure 4.4: Websites from the top 100K list that present Mixed Content

First, we look at the number of websites that include mixed content. This can be visualized in Figure 4.4. In particular, we have that 6,075 websites (6.075%) do include at least one category of mixed content. This number is split into:

- **Websites with upgradeable mixed content**: 4,407 websites (4.407%).
- **Websites with blockable mixed content**: 2,692 websites (2.692%).
- **Websites with both**: 1,024 websites (1.024%).

If we consider the number of websites that include mixed content, we can see that the percentage is lower than the one for the top 1000 websites. This should confirm the initial hypothesis that websites have a lower webpages complexity on average and therefore include fewer external resources.

We now consider the requests that are mixed content per each type. This information can be visualized in Figure 4.5. In total, 769,919 requests were found to be mixed content. These are divided into the following types:

- **Upgradeable requests**: 679,553 requests that have been upgraded.
  - **Element** (subresources): 677,000 requests.
  - **Image** (javascript fetch): 2,320 requests.

  - **Audio** (javascript fetch): 106 requests.
  - **Image** (javascript fetch): 24 requests.

- **Blockable requests**: 90,366 requests have been blocked.
  - **Script**: 41,023 requests.
  - **Stylesheet**: 13,946 requests.
  - **Font**: 12,045 requests.
  - **Frame**: 6,230 requests.
  - **Favicon**: 4,192 requests.
  - **Endpoint** (`fetch`): 8,504 requests.
  - **XHR request**: 2,820 requests.
  - **Plugin resource** (`<object>`): 459 requests.
  - **Manifest**: 119 requests.
  - **Prefetch resource** (CORS requests): 128 requests.
  - **Resource** (`<embed>`): 30 request.
  - **Hyperlink auditing endpoint** (anchor with ping): 11 requests.

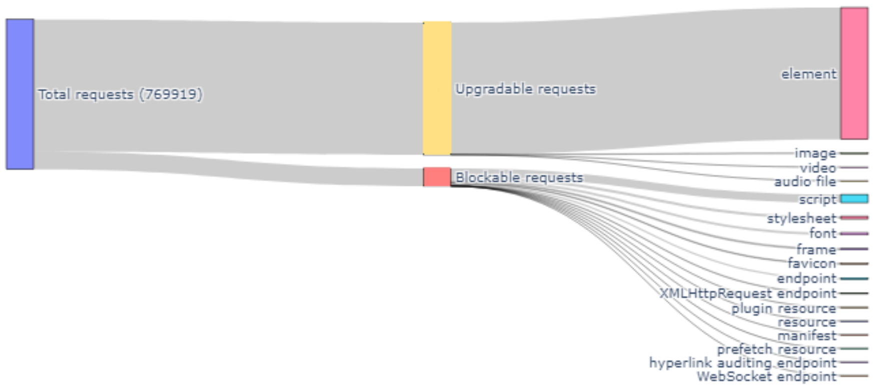

Figure 4.5: Mixed Content requests in the top 100K websites

4.4.4 Mixed content in the wild: 2015 and 2023

We compare now the study of P. Chen et al. from 2015 with the data obtained with our crawl (July 2023). Such comparison can be useful to reason about the effectiveness of the mixed content filtering approach proposed by the W3C specification. The results are summarized in Table 4.4.

We can see that the number of websites with mixed content has decreased noticeably. This happened even though the number of websites scanned by the study in 2015 was smaller than the set we considered. In the study of P. Chen et al. the list of pages to scan was provided by using the Bing API. They focused on sites having at least 200 pages served over HTTPS, so their analysis was conducted on 18,526 sites. On the other hand, we obtained the list from anchors in pages, and our study was performed on all the sites of the top 100K list.

The fact that despite that the number of blockable mixed content requests detected in our study has been reduced by 57.5% shows how in the last 5 years the approach of blocking this type of requests has been effective in forcing web developers to update their websites to serve the content via HTTPS.

On the other hand, the number of upgradeable requests has increased. This can be addressed by considering that older versions of the W3C specification did not force vendors to block upgradeable requests. As stated by the specification authors, this approach was not effective and our numbers do confirm this statement.

The latest version of the specification was published in February 2023 and proposes a strict blocking approach for upgradeable requests. It was quickly implemented in April 2023 by Chromium, while Firefox is starting now to implement it.

We expect to see a decrease in the number of upgradeable requests in the next years thanks to this new approach, as the same policy was effective in reducing the number of blockable requests.

Metric	2015	2023	Difference
Websites with mixed content	43%	6.07%	-36.93%
Mixed Content requests	620,151	769,919	+24.2%
Upgradeable requests	406,932	679,324	+66.9%
Blockable requests	213,219	90,595	-57.5%

Table 4.4: Comparison of mixed content in the wild: 2015 and 2023

5 Conclusions

The framework presented in this thesis allows testing the implementation of security mechanisms in web browsers in a fully automated way. Its design is modular and can be easily extended to support new security mechanisms. We used it to formally verify the implementation of the Mixed Content policy in the three major browsers.

The Mixed Content Policy is a fundamental client-side security mechanism of the Web Platform. It has been regulated by the W3C specification for almost a decade now, and since then it has been implemented in all major browsers.

Despite the importance of the policy, we detected implementation flaws in two of the three major browsers, with one of them being trivially exploitable. The flaws found were present since the beginning of the standardization process. This means that the implementation of the standard was not properly tested by browser vendors.

After having verified the implementation of the Mixed Content policy, we scanned the top 100K websites to evaluate the presence of mixed content. We found that as of today, about 6% of the websites still include some type of mixed content. While this is a significant improvement compared to the situation in 2015, we can affirm that the issue is still present in a non-negligible number of websites. The approach proposed by the W3C specification to strictly disallow blockable content was very effective in reducing the presence of this type of security risk. On the other hand, the approach proposed with upgradeable content was not as efficient.

A new version of the Mixed Content specification was released 8 months ago, introducing a new approach for filtering upgradeable content. The approach proposes a more strict policy that increases security while minimizing the risk of breaking web page functionalities. Despite that, two of the three major browser vendors did not comply yet with the latest version of the specification, and while one of them is planning to do so soon, the other has not yet announced any plans to update the implementation.

Bibliography

[1] The Chromium Authors. Automatically lazyloading offscreen images and iframes for lite mode users. https://blog.chromium.org/2019/10/automatically-lazy-loading-offscreen.html.

[2] The Chromium Authors. Chromium contribution guideline. https://chromium.googlesource.com/chromium/src/+/refs/heads/main/docs/contributing.md#Running-automated-tests.

[3] The Chromium Authors. Function measurestricterversionofismixedcontent. https://source.chromium.org/chromium/chromium/src/+/main:third_party/blink/renderer/core/loader/mixed_content_checker.cc;l=242;drc=f649fd2e9a7f00f657c025800f5d7ed3d1bffd21.

[4] The Chromium Authors. Mixed content iframe leads to crash in navigationrequest::getassociatedrfhtype. https://bugs.chromium.org/p/chromium/issues/detail?id=1482252.

[5] The Chromium Authors. Mixed content parent frame checking logic might be wrong. https://bugs.chromium.org/p/chromium/issues/detail?id=623486.

[6] The Chromium Authors. Oopif: layout tests failures when inspecting a cross-site iframe's properties with –site-per-process. https://bugs.chromium.org/p/chromium/issues/detail?id=623268.

[7] The Chromium Authors. *data:* uri inheriting trustworthiness. https://source.chromium.org/chromium/chromium/src/+/main:third_party/blink/renderer/core/loader/mixed_content_checker_test.cc;l=49;drc=974a4bdf9247c35d13a73e33ebbe06640762cad9.

[8] The Chromium Authors. Wpt integration in chromium. https://chromium.googlesource.com/chromium/src/+/HEAD/docs/testing/web_platform_tests.md.

[9] The Chromium Authors. Chromium repository on github. https://github.com/chromium/chromium, 2023.

[10] The WebKit Authors. Consider mixed-content iframes to be mixed script instead of mixed display. `https://bugs.webkit.org/show_bug.cgi?id=116065`.

[11] The WebKit Authors. Repository commit to fix cve-2023-38592. `https://github.com/WebKit/WebKit/commit/bc09b6fca3255742370fdd150cf9627b53794c1e`.

[12] The WebKit Authors. Webkit contribution guideline. `https://webkit.org/contributing-code/`.

[13] The WebKit Authors. Wpt integration in webkit. `https://docs.webkit.org/Infrastructure/WPTTests.html#import-wpt-tests-from-a-local-checkout-of-wpt`.

[14] Ping Chen, Nick Nikiforakis, Christophe Huygens, and Lieven Desmet. A dangerous mix: Large-scale analysis of mixed-content websites. In Yvo Desmedt, editor, *Information Security*, pages 354–363, Cham, 2015. Springer International Publishing.

[15] WPT Contributors. Wpt-testharness api. `https://web-platform-tests.org/writing-tests/testharness-api.html`.

[16] WPT Contributors. Web-platform-tests. `https://web-platform-tests.org/`, 2019.

[17] ECMA. Ecmascript standard. Technical report, ECMA, 2023. `https://www.ecma-international.org/publications-and-standards/standards/ecma-262/`.

[18] Gertjan Franken, Tom Van Goethem, Lieven Desmet, and Wouter Joosen. A bug's life: Analyzing the lifecycle and mitigation process of content security policy bugs. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 3673–3690, Anaheim, CA, August 2023. USENIX Association.

[19] Google. Https encryption on the web - transparency report. `https://transparencyreport.google.com/https/overview?hl=en`.

[20] James Graham. Integration of wpt in firefox repository. `https://bugzilla.mozilla.org/show_bug.cgi?id=945222`, 2014.

[21] Ivan Herman. Find the best terminology to restrict the usage of data urls. `https://github.com/w3ctag/design-reviews/issues/635`.

[22] Internet Engineering Task Force (IETF). The web origin concept. `https://www.rfc-editor.org/rfc/rfc6454`, 2011.

[23] Internet Engineering Task Force (IETF). Media type specifications and registration procedures. `https://www.rfc-editor.org/rfc/rfc6838`, 2013.

[24] Internet Engineering Task Force (IETF). Http/1.1. https://www.rfc-editor.org/rfc/rfc9112, 2022.

[25] Collin Jackson and Adam Barth. Forcehttps: Protecting high-security web sites from network attacks. In *Proceedings of the 17th International Conference on World Wide Web*, 2008.

[26] Mozilla. Mozilla contribution guideline. https://www.mozilla.org/en-US/about/governance/policies/commit/access-policy/.

[27] Mozilla. Wpt integration in firefox. https://firefox-source-docs.mozilla.org/web-platform/index.html.

[28] Dirk Pranke. Integration of wpt in blink engine repository. https://bugs.chromium.org/p/chromium/issues/detail?id=413454, 2014.

[29] Titouan Rigoudy. Data url iframes are considered secure contexts only if sandboxed. https://github.com/w3c/webappsec-secure-contexts/issues/83.

[30] Jörg Schwenk, Marcus Niemietz, and Christian Mainka. Same-Origin policy: Evaluation in modern browsers. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 713–727, Vancouver, BC, August 2017. USENIX Association.

[31] Clarck B. Sengupta S., Tal L. Web almanac 2022: Security. https://almanac.httparchive.org/en/2022/security.

[32] Suphannee Sivakorn. Understanding flaws in the deployment and implementation of web encryption. In *Ph.D. Thesis, Columbia University*, 2017.

[33] M. Squarcina, P. Adão, L. Veronese, and M. Maffei. Cookie crumbles: Breaking and fixing web session integrity. In *USENIX Security '23*, 2023.

[34] Emily Stark, Mike West, and Carlos IbarraLopez. Mixed content. Candidate recommendation, W3C, February 2023. https://www.w3.org/TR/2023/CRD-mixed-content-20230223/.

[35] Anne van Kesteren. Can data: Urls be part of a secure context? https://github.com/w3c/webappsec-secure-contexts/issues/69.

[36] L. Veronese, B. Farinier, P. Bernardo, M. Tempesta, M. Squarcina, and M. Maffei. Web-spec: Towards machine-checked analysis of browser security mechanisms. In *44th IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2023.

[37] Samaneh Tajalizadehkhoob Maciej Korczyński Victor Le Pochat, Tom Van Goethem. A research-oriented top sites ranking hardened against manipulation. https://tranco-list.eu/.

[38] Jonathan Watt. Sandboxed data: Uri in a localhost page should be a secure context. https://github.com/w3c/webappsec-secure-contexts/issues/26.

[39] WebKit. Webkit bug prioritization. https://webkit.org/bug-prioritization/.

[40] Mike West. Secure contexts. Candidate recommendation, W3C, September 2021. https://www.w3.org/TR/2021/CRD-secure-contexts-20210918/.

[41] WHATWG. Html standard. Technical report, whatwg, 2021. https://html.spec.whatwg.org/multipage/webappapis.html.

[42] WHATWG. Dom standard. Technical report, whatwg, 2023. https://dom.spec.whatwg.org/.

[43] WHATWG. Fetch standard. Technical report, whatwg, 2023. https://fetch.spec.whatwg.org/.

[44] WHATWG. Url living standard. Technical report, whatwg, 2023. https://url.spec.whatwg.org/.

[45] WHATWG. Xmlhttprequest living standard. Technical report, whatwg, 2023. https://xhr.spec.whatwg.org/.