HTTPS-Only: Upgrading all connections to `https` in Web Browsers

Christoph Kerschbaumer
Mozilla Corporation
ckerschb@mozilla.com

Julian Gaibler
Mozilla Corporation
jgaibler@mozilla.com

Arthur Edelstein
Mozilla Corporation
arthur@mozilla.com

Thyla van der Merwe†
ETH Zürich
tvdmerwe@ethz.ch

***Abstract*—The number of websites that support encrypted and secure `https` connections has increased rapidly in recent years. Despite major gains in the proportion of websites supporting `https`, the web contains millions of legacy `http` links that point to insecure versions of websites. Worse, numerous websites often use `http` connections by default, even though they already support `https`. Establishing a connection using `http` rather than `https` has the downside that `http` transfers data in cleartext, granting an attacker the ability to eavesdrop, or even tamper with the transmitted data. To date, however, no web browser has attempted to remedy this problem by favouring secure connections by default.**

We present *HTTPS-Only*, an approach which first tries to establish a secure connection to a website using `https` and only allows a fallback to `http` if a secure connection cannot be established. Our approach also silently upgrades all insecure `http` subresource requests (image, stylesheet, script) within a secure website to use the secure `https` protocol instead. Our measurements indicate that our approach can upgrade the majority of connections to `https` and therefore suggests that browser vendors have an opportunity to evolve their current connection model.

I. Introduction

The Hypertext Transfer Protocol (generally displayed as `http` in a browsers address-bar) [10] is the fundamental protocol through which web browsers and websites communicate. However, data transferred by the regular `http` protocol is unprotected and transferred in cleartext, such that attackers are able to view, steal, or even tamper with the transmitted data. Carrying `http` over the Transport Layer Security (TLS) protocol (generally displayed as `https` in the address-bar of a browser) [11] fixes this security shortcoming by creating a secure and encrypted connection between the browser and the website. More precisely, TLS enables the browser to authenticate the identity of the web server to the browser, and ensures that messages sent between the browser and the server are kept confidential from all other parties. Browsers typically display the lock icon (🔒) in the address-bar to indicate that the connection is an `https` connection and therefore encrypted and secure.

Over the past few years we have witnessed tremendous progress towards migrating the web to rely on `https` instead of the outdated and insecure `http` protocol. Efforts like HTTP Strict Transport Security (HSTS) [14] and the vitally important *Let's Encrypt* initiative [21] have helped to accelerate this migration. HSTS informs the browser that a server prefers secure connections, and *Let's Encrypt* [21] allows web servers to automatically obtain a browser-trusted certificate, enabling secure connections over `https` between browser and server. Importantly, the majority of websites already support `https` connections, and those that do not are increasingly uncommon. And yet, regrettably, the web contains millions of legacy `http` links that point to insecure versions of websites. Additionally, websites frequently fall back to using the insecure and outdated `http` protocol. Web browsers traditionally do not make any effort to adjust this security drawback by trying to upgrade the request and establish a secure connection instead.

As of December 2020, all major browsers (Chrome, Firefox, Edge, Safari) do not attempt to upgrade the scheme of a URL when the user clicks any outdated legacy `http` link. Additionally, all browsers default to using `http` when the user enters a scheme-less URL, (e.g., typing `example.com` into the address-bar). They also do not attempt to upgrade the scheme of a URL to `https` when the user enters or pastes an `http` URL into the address-bar. This industry-wide browser preference for `http` connections has been the status quo since the inception of the web and has not changed even after the introduction of `https`. Browser vendors are understandably hesitant to upgrade connections when such upgrades could downgrade a user's experience in any form.

On the server side, converting all legacy `http` links to `https` in websites is time-consuming and expensive. To successfully migrate a whole website, it's necessary to serve not only top-level documents but also all subresources (such as images, stylesheets, or scripts) over `https`, to make sure that no page content is blocked by web browsers' Mixed Content Blocker [36]. Thus it is also not surprising that not all websites have yet fully migrated to `https`.

To compensate, we present *HTTPS-Only Mode*, a new security feature that tries to upgrade all connections (top-level and subresource) to rely on the secure `https` protocol. The principle idea behind the *HTTPS-Only* approach is that resources are increasingly likely to be available over `https` as the web progresses towards `https`. *HTTPS-Only* first tries to establish a secure `https` connection to a website. If and only if that secure connection cannot be established, our algorithm presents the end user with an exception page, explaining the security risk and giving the user an option to either abandon the connection attempt, or to connect using the insecure and outdated `http` protocol. Our approach aims to pave the way for a reform in industry-wide browser design to make `https` the default protocol for the web.

†Work completed whilst this author was employed by Mozilla.

Workshop on Measurements, Attacks, and Defenses for the Web (MADWeb) 2021
25 February 2021, Virtual
ISBN 1-891562-67-3
https://dx.doi.org/10.14722/madweb.2021.23010
www.ndss-symposium.org

The remainder of this paper is structured as follows:

- In Section II we provide background information on the related mechanisms HTTP Strict Transport Security and Mixed Content Blocking.
- In Sections III and IV we present the design and implementation details of *HTTPS-Only Mode*, a browser security feature which upgrades all connections from `http` to `https`, implemented in Firefox (v.83.0).
- In Section V we examine the effectiveness of the proposed *HTTPS-Only* approach by evaluating real world data reported by Firefox end users.
- In Section VII we discuss the feasibility of changing the industry wide default to use `https` instead of `http` in all web browsers.

II. Background

Before presenting the design and implementation of *HTTPS-Only Mode*, we give an overview of two relevant security technologies: (a) *HTTP Strict Transport Security* and (b) *Mixed Content Blocking*. Both technologies are essential to the *HTTPS-Only* approach and to the objective of making the web rely on secure connections only.

A. HTTP Strict Transport Security (HSTS)

HTTP Strict Transport Security (HSTS) [14] is a browser security mechanism that allows a website to signal that the browser should only interact with a website using secure `https` connections and never with insecure `http` connections. For HSTS-enabled websites, the browser will require an `https` connection to a website even when the URL the browser is following is a non-secure `http` URL.

A website implements an HSTS policy by sending the `Strict-Transport-Security` response header in `https` server responses during `https` connections. The presence of the header indicates that the browser should automatically upgrade `http` links to resources on the website to the corresponding `https` links. Because browsers will ignore the header if it is sent over non-secure `http`, web servers utilising HSTS first have to redirect and upgrade the non secure `http` request to a secure `https` request. The header can further provide a `max-age` directive specifying how long the browser should cache the provided HSTS information.

When the browser receives an HSTS header, it caches the fact that the sending website wishes to be upgraded, and upgrades future requests to the website. This allows the browser to automatically turn any non-secure `http` link for a website into a secure `https` link. For example, suppose `https://siteA/index.html` embeds a script from `http://siteB/widget.js`. If `siteB` deploys HSTS, and the browser has previously visited `siteB` and additionally received an HSTS header, then the browser will load the script over `https` despite the `http` link.

HSTS has a vulnerability, however: before a browser has visited a website and received HSTS information, a request may still occur using plain `http` and is therefore still be vulnerable to downgrade attacks by tools such as SSLStrip [22]. To decrease the risk of any kind of downgrade attack, modern browsers introduced the *HSTS Preload* mechanism - a compiled list of HSTS supporting domains which is shipped with the browser [1], [23]. The browser will only make secure `https` connections to websites on the *HSTS Preload* list. Unfortunately, the web contains billions of websites [16] and hence scaling such lists remains challenging. For example, as of December 2020, the Firefox preload list contains roughly 100,000 entries.[1]

B. Mixed Content Blocking

The *Mixed Content Blocker* [36] is a browser security mechanism that blocks insecure content on pages that are supposed to be secure. That is, if the top level page is `https` then a browser's *Mixed Content Blocker* blocks requests for non-secure subresources within that page. The mixed content specification distinguishes between active (blockable) and passive (optionally-blockable) content.

a) ***Passive Mixed Content****:* Content, such as images, audio, and video resources, cannot alter other portions of a website page. For example, an attacker could replace an image served over non-secure `http` with an inappropriate or deceptive image, but could not otherwise change the behaviour of the page. According to the mixed content specification [36], blocking such content is optional and browser vendors may decide whether to block or allow such content. For a long time all major browsers followed the suggestion of the specification and loaded mixed passive content, additionally providing an indicator in the browser UI signalling that mixed content had loaded. More recently, the successor specification named *Mixed Content Level 2* suggests that browsers should auto-upgrade passive mixed content [39].

b) ***Active Mixed Content****:* In contrast, blockable content such as scripts or stylesheets have access to all or parts of the Document Object Model (DOM) [34]. Loading active mixed content into an otherwise fully `https` compatible website allows an attacker to change the behaviour of, or even steal sensitive user information from, the `https` website. For example, if a script were loaded over non-secure `http`, an attacker could inject a modified script that would log the user's keystrokes to an attacker's server. This is why modern browsers now warn users when a non-secure website page has a password field, even if the password is submitted over `https` [24]. Because of the high risk posed by active content, the mixed content specification suggests that browsers always block mixed active content.

We recognise that both of the aforementioned technologies are critical security enhancements to the browser ecosystem. Nevertheless, HSTS and Mixed Content Blocking do not fully solve the problems posed by web connection security. More precisely, if a website does not get upgraded to `https`, then the Mixed Content Blocker has no effect and if a website gets upgraded to `https`, then the Mixed Content Blocker might block relevant resources necessary for the functionality of the website, therefore potentially downgrading the end user's experience.

[1] https://searchfox.org/mozilla-central/source/security/manager/ssl/nsSTSPreloadList.inc

III. Design

The fundamental security problem of the current browser practice of defaulting to use insecure `http`, instead of secure `https`, when initially connecting to a website is that attackers can intercept the initial request. Hijacking the initial request suffices for an attacker to perform a man-in-the-middle attack which in turn allows the attacker to downgrade the connection, eavesdrop or modify data sent between client and server.

Using `http` as the browser default was sensible when the bulk of websites supported `http`. In 2020, however, we see that the majority of websites support `https`. Regrettably, misconfigured websites frequently default to the insecure and outdated `http` protocol even though they already support `https`. Worse, the web contains millions, if not billions, of legacy `http` links that point to insecure versions of websites. When a user clicks on such a link, a browser traditionally connects to the website using the insecure `http` protocol. To overcome this legacy problem and to establish a secure-by-default connection mechanism, our proposed approach attempts to silently upgrade connections. More precisely, when the user clicks an `http` link or enters an `http` URL in the address-bar, *HTTPS-Only* first tries to establish a secure connection to the website using `https`.

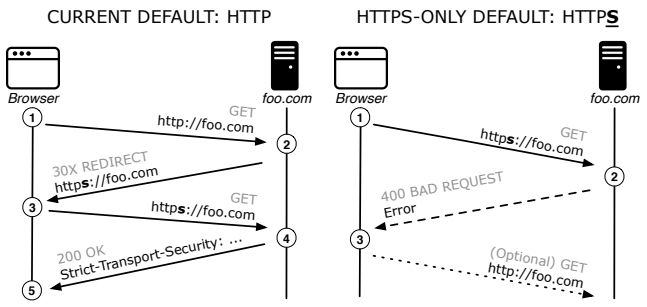

Fig. 1: Current browser behaviour defaulting to `http` (left) vs. *HTTPS-Only* behaviour defaulting to `https` (right).

Current best practice to counter the explained man-in-the-middle security risk primarily relies on HSTS (see Section II). However, HSTS does not solve the problems associated with performing the initial request in plain `http`. As illustrated in Figure 1 (left), the current browser default is to first connect to `foo.com` using `http` (see ①). If the server follows best practice and implements HSTS, then the server responds with a redirect to the secure version of the website (see ②). After the next `GET` request (see ③) the server adds the HSTS response header (see ④), signalling that the server prefers `https` connections and the browser should always perform `https` requests to `foo.com` (see ⑤) .

In contrast and as illustrated in Figure 1 (right), the presented *HTTPS-Only* approach first tries to connect to the web server using `https` (see ①). Given that most popular websites currently support `https`, our upgrading algorithm commonly establishes a secure connection and starts loading content. In a minority of cases, connecting to the server using `https` fails and the server reports an error (see ②). The proposed *HTTPS-Only Mode* then prompts the user, explaining the security risk, to either abandon the request or to connect using `http` (see ③).

IV. Implementation

In brief, our proposed security-enhancing feature internally upgrades (a) top-level document loads as well as (b) all subresource loads (images, stylesheets, scripts) within a secure website by rewriting the scheme of a URL [8] from `http` to `https`. While this overall approach is simple in principle, implementing a product-ready version of the presented upgrading algorithm entails many corner cases and potential pitfalls, which we describe in detail in this section.

We implement our upgrading mechanism within Firefox (v.83.0) which enforces a Secure by Default [18] loading mechanism and attaches meta-information to every resource load. Building on these efforts, we implement *HTTPS-Only*, by instrumenting and subsequently encoding additional information into the meta object attached to every single request. This encoding of additional information in a request's (top-level or subresource) meta object allows us to silently upgrade any connection from `http` to `https` in Necko, the network layer in Firefox.

A. Upgrading Top-Level (HTML Document) Loads

Upgrading a top-level request (that is, the top-level HTML document [40] of a web page) with *HTTPS-Only* entails uncertainties about the response that the browser needs to handle. If everything works optimally, then Firefox using *HTTPS-Only* simply connects to a website using `https` and the browser proceeds to load the web page securely. If, however, connecting securely to the web server fails, then *HTTPS-Only* prompts the user with an exception page, explaining the problem and the security risk, and provides an option for the end user to *'Accept the Risk'* and connect using `http`. Our approach supports this fallback mechanism because at present not all websites on the Web support `https`, and simply blocking any connection to those websites would downgrade the end user's experience. To accommodate for websites which do not yet support secure connections, we distinguish between the following two error cases:

a) ***Immediate Errors****:* If there is an error response, either from the remote server itself or a firewall, then our approach can instantly respond to that error. Error responses can range from a `TCP Reset packet` to server responses with some type of TLS error. Therefore, our approach interprets all errors reported by the networking code as an *HTTPS-Only* error. If detected, our approach prompts the user with an exception page that explains the problem and the security risk, and provides an option to connect to the website using `http`.

b) ***Timeouts****:* A non-responding firewall or a misconfigured or outdated server that fails to send a response can result in long timeouts which ultimately downgrade an end user's experience, forcing the user to wait a long time for the exception page to appear before they can continue browsing. When testing *HTTPS-Only* we experienced timeouts taking as long as 90 seconds. To mitigate this problem, *HTTPS-Only* first sends a top-level request for `https`, and after an N-second delay, if no response is received, sends an additional `http` background request. If the background `http` connection is established prior to the `https` connection, then this signal is a strong indicator that the `https` request will result in a timeout. In this case, the `https` request is canceled and the user is shown the *HTTPS-Only Mode* exception page.

```
1  void PotentiallySendBGRequest(nsIChannel* aOrigChannel) {
2    // if not top-level load, then there is nothing to do
3    nsILoadInfo* loadInfo = aOrigChannel->GetLoadInfo();
4    if (loadinfo->Type() != TYPE_DOCUMENT)){
5      return;
6    }
7
8    // if already https, then there is nothing to do
9    nsIURI* uri = aOrigChannel->GetURI();
10   if (!uri->SchemeIs("http")) {
11     return;
12   }
13   NewTimerWithCallBack(3000, SendBGRequest(aOrigChannel));
14 }
15
16 void SendBGRequest(nsIChannel* aOrigChannel) {
17   nsIURI* uri = aOrigChannel->GetURI();
18   uri->StripPathAndQuery();
19
20   uint32_t loadFlags = LOAD_ANONYMOUS
21                      | LOAD_BYPASS_CACHE
22                      | LOAD_HTTPS_ONLY_EXEMPT;
23
24   nsIChannel* bgChannel = NewChannel(uri, loadFlags);
25
26   BGListener *bgListener = new BGListener(aOrigChannel);
27   bgChannel->SetNotificationCallBacks(bgListener);
28   bgChannel->AsyncOpen();
29 }
30
31 void BGListener::OnStartRequest() {
32   if (mOrigChannel->IsLoading()) {
33     return;
34   }
35   mOrigChannel->Cancel(POTENTIAL_TIMEOUT);
36 }
```

Listing 1: *HTTPS-Only Mode* trying to establish top-level `https` connection while simultaneously connecting using `http` in the background to avoid long timeouts.

Whenever a user enters an insecure URL into the address-bar, or clicks a legacy link, our *HTTPS-Only* approach upgrades the connection to `https`, but subsequently calls the function `PotentiallySendBGRequest()`, providing an `nsIChannel` argument which is Firefox's internal representation of a network request in Necko. The function `PotentiallySendBGRequest()` determines if our mechanism should send a background request so that end users do not have to wait for a connection timeout in case the website does not respond to the upgraded `https` request (thereby improving the user experience of our *HTTPS-Only* mechanism).

As illustrated in Listing 1, this function first queries the `nsILoadInfo` of the `nsIChannel` (Line 3), a data structure which provides varied loading information [18]. Amongst other things, the `nsILoadInfo` object conveys the load type. Since our approach only performs a background request for top-level loads it can immediately return if the load type of the request is not `TYPE_DOCUMENT` (Line 5). Similarly, if the request is not `http` (Line 10), meaning the user has already entered an `https` scheme, then there is no need to send a background request and the function can return. If, however, the checks have not caused the function to return, then we send a background request by calling the function `SendBGRequest()` using an N-second delay (Line 13). Using a delay is crucial because we need to account for enough time such that the browser and the web server can negotiate parameters, perform the TLS handshake, and establish a secure connection. We empirically found that setting N=3000 milliseconds allows for the best results on Desktop and Mobile (please see Figure 6 in the Evaluation section).

Path and query information is irrelevant in determining whether a server responds to a connection request, and users browsing using *HTTPS-Only Mode* expect the browser to not leak any user sensitive information. Therefore, in the first step in the function `SendBGRequest()`, we strip any path and query information from the URL (Line 18). While our approach strips user sensitive information from the background request so as to not reveal any user data by default, we also provide an option for cautious users to opt out of sending the background request.[2] The downside is that such cautious users will have to wait until the request times out before the exception page appears.

Next, we equip the new background network connection (Line 20) with three flags: `LOAD_ANONYMOUS`, `LOAD_BYPASS_CACHE`, `LOAD_HTTPS_ONLY_EXEMPT`. These three flags ensure that we (a) perform an anonymous request by not attaching any cookies, (b) ask that the request does not end up in our cache, and (c) exempt the load from *HTTPS-Only*, otherwise our upgrading mechanism would upgrade the connection to `https` later in Necko which we explicitly want to avoid for sending the background request.

Before opening the background channel and connecting to the server (Line 28), we have to create a Listener (Line 26) and set the Notification callbacks (Line 27). These two mechanisms allow our code to track progress of the original, upgraded top-level channel. In particular, within the function `OnStartRequest()` on Line 31, we check if the original channel has already started loading by consulting `mOriginalChannel->IsLoading()`. If that function returns true, then we know that the original channel, which was upgraded to `https`, is already loading. If that function returns false, however, meaning that the upgraded channel encounters a problem, then we cancel the original channel (Line 35). This causes the exception page to appear and signals to the end user that the server has not responded to the `https` request. The user then can then to choose to connect to the website using plain `http`, if desired.

B. Upgrading Subresource (Image, Stylesheet, Script) Loads

Any given web page consists of many different resources that are fetched by the browser as requested by the top-level HTML document [40], including images, stylesheets, scripts and other content linked via URLs [8].

Upgrading the top-level document only to use `https` would provide limited security guarantees because an active network attacker could still eavesdrop and tamper with subresources loaded over `http`. Additionally, not upgrading subresources to `https` results in mixed content (see Section II), and a browser's implementation of the *Mixed Content Blocker* starts blocking active content like scripts, thereby downgrading an end user's experience.

Instead of only upgrading top-level requests, our holistic approach also upgrades subresources by rewriting the scheme in the URL. However, if loading a subresource over `https` fails, then in contrast to the handling of top-level loads, our

[2] security.https_only_mode_send_http_background_request

approach does not provide any kind of fallback mechanism. Instead *HTTPS Only* logs a message to the *Browser Console* indicating that the upgrade attempt failed but, as previously mentioned, does not fall back to trying to load the request using insecure `http`.

In addition to comprehensively enforcing `https` for subresources, *HTTPS-Only* also accounts for WebSockets [13]. A WebSocket provides full-duplex communication channels over a single TCP connection and enables interaction between a web browser (or other client application) and a web server with lower overhead than half-duplex alternatives such as `http` polling. Similar to other subresource loads, *HTTPS-Only* upgrades a WebSocket's scheme from `ws:` to `wss:`.

It can happen that a website itself is available over `https` but resources within the website page, such as images or videos, are not available over `https`. Consequently, such websites may not look right or might malfunction. To compensate, our implementation provides an option which allows users to disable *HTTPS-Only* for a website that has been loaded with `https`. For any website that has been upgraded by *HTTPS-Only Mode*, Firefox shows a user-interface widget in the security doorhanger, accessed by clicking the lock icon (🔒) in the address-bar, which allows the user to temporarily or permanently disable *HTTPS-Only* for that website.

Beyond these considerations, a practical implementation of *HTTPS-Only* also needs full integration with two critical browser security mechanisms related to subresource loads: (a) *Mixed Content Blocker*, and (b) *Cross-Origin Resource Sharing* (CORS).

a) Interaction with the Mixed Content Blocker: As explained in Section II, the *Mixed Content Blocker* blocks `http` subresources within a top-level `https` page. Since our approach upgrades resources in Necko, the networking layer in the Firefox web browser, we have to adjust the *Mixed Content Blocker* implementation to not block the potential mixed content subresource load, relying on Necko to upgrade the load later.

```
1  bool BlockMixedContent(
2    nsIURI* aTopLevelURI, nsIChannel* aSubresourceChannel) {
3    if (!aTopLevelURI->SchemeIs("https")) {
4      return false;
5    }
6
7    nsIURI* channelURI = aSubresourceChannel->GetURI();
8    if (!channelURI->SchemeIs("http")) {
9      return false;
10   }
11
12   if (Preferences::HTTPS_ONLY_MODE_ENALBED()) {
13     nsILoadInfo* loadInfo = aSubresourceChannel->LoadInfo();
14     if (loadInfo->HttpsOnlyMode() != HTTPS_ONLY_EXEMPT) {
15       return false;
16     }
17   }
18   // evaluate if subresource is active or passive content
19   return true;
20 }
```

Listing 2: Interaction of *HTTPS-Only* with the Mixed Content Blocker.

For every subresource load within Firefox, the browser consults the function `BlockMixedContent()`. As illustrated within Listing 2, if a the top-level document load is not `https`, then a browser's *Mixed Content Blocker* has nothing to evaluate and can return at this point (see Line 3). Further, if the subresource load is not `http` (which is the case when loading `data:` [9] or also `blob:` [38]), then the *Mixed Content Blocker* also has nothing to evaluate and can return (see Line 8). Generally, the actual work for the mixed content algorithm starts after Line 10, when we know the top-level load is `https` and the subresource load is `http`, indicating the load is actually mixed.

On Line 12 we have a check for whether the preference for *HTTPS-Only* is enabled. If the preference for *HTTPS-Only* is not set to true, then the *Mixed Content Blocker* implementation works as for every other subresource load despite having a branch for the presented *HTTPS-Only* implementation. If the preference is true, then we have to check the actual mode using HttpsOnlyMode() stored in the `nsILoadInfo` (Line 14). This flag ensures the load is not exempt from upgrading; more details on Exemptions are given in Section IV-C. If the load is not exempt from upgrading, then we return at Line 15, not blocking the subresource load as mixed content with the knowledge that the subresource request will be upgraded to `https` later in Necko before loading any data over the network.

b) Interaction with CORS: Cross-Origin Resource Sharing (CORS) [35] is a mechanism that uses additional headers to tell browsers to give a web application running at one origin, access to selected resources from a different origin. A web application executes a cross-origin `http` request when it requests a resource that has a different origin (scheme, host or port) [12] from its own.

Before performing any kind of CORS check, the browser first evaluates whether the request is same-origin with the encompassing security context. To illustrate, lets assume a top-level document of `https://www.example.com` performs an XMLHttpRequest (XHR) [41] to `http://www.example.com/foo`. In such a case, the browser has to issue a CORS request because the schemes of `http` and `https` differ, meaning the request is cross origin and hence requires CORS to succeed. But since our proposed approach will upgrade the XHR request to `https://www.example.com/foo` it will also render the CORS request obsolete and hence we need to discard it.

```
1  bool CheckHTTPSOnlyPreventsCORS(
2    nsIURI aTopLevelDocURI, nsIChannel* aSubresourceChannel) {
3
4    nsIURI* channelURI = aSubresourceChannel->GetURI();
5    if (!channelURI->SchemeIs("http")) {
6      return false;
7    }
8
9    nsString topLevelDocHost = aTopLevelDocURI->GetHost();
10   nsString subresourceHost = channelURI->GetHost();
11
12   if (topLevelDocHost.Equals(subresourceHost)) {
13     return true;
14   }
15   return false;
16 }
```

Listing 3: Interaction of *HTTPS-Only* with CORS.

Once Firefox's CORS implementation detects that a request is cross-origin, it will normally enforce CORS (note that CORS only applies to `http(s):` requests and is discarded for all other schemes). Since our approach potentially upgrades

requests from http to https, we have to add a carve-out to account for that scenario by consulting the function CheckHTTPSOnlyPreventsCORS().

As illustrated in Listing 3, the newly added helper function, named CheckHTTPSOnlyPreventsCORS(), first checks if a request is http (Line 5), as otherwise our approach would not influence the result of the load. If the load is in fact http, meaning that it will be upgraded to https later in the loading cycle, the helper function simply needs to verify that the host of the top-level document (Line 9) and the host of the subresource load (Line 10) are equal (Line 12). If they are equal, meaning that only the scheme differs, then there is no need to issue a CORS request because our implementation makes the request to be same-origin by upgrading the subresource scheme from http to https. If that is not the case, and the hosts differ, then our approach has no influence on the result and CORS will be applied to the request, as it would without operating in *HTTPS-Only Mode*.

C. Upgrading Exemptions

Generally, and as previously mentioned, we designed *HTTPS-Only Mode* following the principle of *Secure by Default* which means that by default, our approach will upgrade all outgoing connections from http to https. Following this principle allows us to provide a future-proof product implementation where exceptions to the rule require explicit annotation, such as in the following two illustrative examples:

a) ***OCSP Requests****:* The Online Certificate Status Protocol (OCSP) [29] allows the browser to validate or determine the revocation status of an X. 509 certificate by sending an OCSP request to the OCSP server listed in the certificate. OCSP commonly works under the soft-fail principle, which means that if the server cannot be reached, then it is assumed that the certificate is still valid. Unsuccessfully upgrading such a request would carry a higher risk for users than allowing it to be sent over http. Internally, when creating the meta object named nsILoadInfo [18] for an OCSP request, we annotate the request by setting the flag HTTPS_ONLY_EXEMPT, which instructs our algorithm to exempt that request from being upgraded to https. We are mindful that having to rely on http is not optimal for preserving the user's privacy. While Firefox still supports OCSP, this privacy drawback is one of the many reasons Firefox will soon switch to CRLite [20] to determine certificate revocation status.

b) ***Captive Portal Requests****:* Captive portals are special websites that users need to visit to access a public network, commonly at hotel lobbies, airports or coffee shops, before access to the network and internet is granted. A common method to implement such captive portals is to intercept all web traffic to a web server and return an http redirect to a captive portal. To determine whether a user has access to the internet, Firefox sends a network request in the background to a specific website. If the website responds as expected, then Firefox can assume the user is connected to the internet. Since https requests cannot be modified by the network to redirect to the captive portal, this background request needs to remain a plain http request. Hence, we explicitly annotate these requests by setting the flag HTTPS_ONLY_EXEMPT on the nsILoadInfo-object, so that our *HTTPS-Only* algorithm does not upgrade them.

V. Evaluation

We break down the evaluation of our approach into four segments. We start out by first providing statistics, gathered over recent years, showcasing the percentage increase of websites relying on https (Section V-A). Second, we lay out fundamentals about our data-gathering process including the time frame and participating users of our study (Section V-B). Third, we provide detailed numbers on the effectiveness of our approach by showing how many top-level (document) loads our mechanism is able to upgrade (Section V-C), and finally, we present similar details for subresource loads (Section V-D).

A. State of the Internet - Websites loaded using https

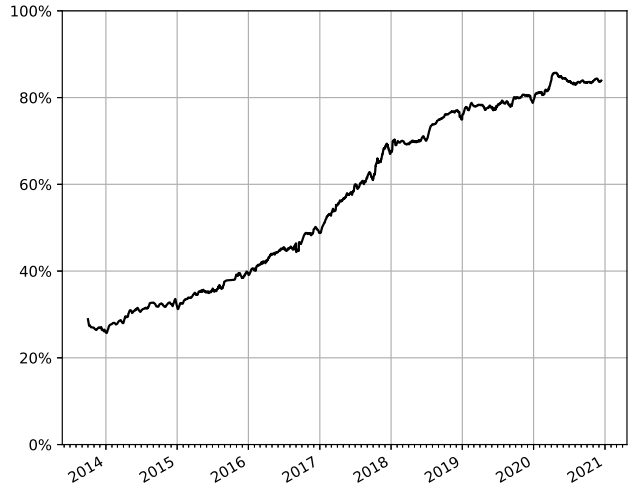

Fig. 2: Evolution of websites relying on https for the years 2014 to 2020. (Note: Mozilla also reports this evolvement of websites relying on https to *Let's Encrypt* [21]).

In 2015, a mere six years ago, not even 40% (Figure 2) of web pages were loaded using https, leaving more than half of the websites vulnerable to man-in-the-middle attacks, thereby jeopardising an end user's security and privacy.

In 2016, *Let's Encrypt* was launched, and likely had a large impact on increasing the number of websites relying on https by making it possible for website owners to automatically obtain a free browser-trusted certificate. As illustrated in Figure 2, the graph shows a doubling of https use between the years 2016 and 2019.

By 2020, around 84% (Figure 2) of web pages were loaded using https. While the exact usage rate varies by region, we note that in the United States, for example, the number of websites loading over https is as high as 92%. Taking a closer look at the development between the years 2019 and 2020, however, reveals that the increase of websites loading over https flattens. This flattening of the curve signals that we have entered a new phase. We hypothesize that new approaches, such as *HTTPS-Only Mode*, can help to close the gap and allow more websites to load using https, ultimately delivering a fully secure web with a 100% of connections being over https.

B. Background on Data-Gathering

We examine the capabilities and potential limitations of our approach by inspecting real world data from Firefox end users. *HTTPS-Only Mode* was launched as an opt-in security feature in Firefox 83, which was released on November 17th, 2020. Users can enable the feature by visiting `about:preferences#privacy` and flipping the radio button to enable *HTTPS-Only Mode*.

The *Mozilla Telemetry Portal* [26] allows us to present information which we gathered during one full Firefox release cycle, from **November 17th to December 17th, 2020**. Please note that our telemetry mechanisms measure and collect non-personal information, such as performance, usage and customisations and sends this information to Mozilla on a daily basis. All of the collected information used for analysis is subject to *Mozilla's Data Privacy Principles* which are covered in the Firefox Data Documentation [25].

During our collection period of one month we were able to **collect real world data of 103,431 Firefox users** who participated in our study and agreed to report information back to Mozilla. Evaluating real-world browsing behaviour of Firefox release users allows us to quantitatively examine *HTTPS-Only*, as presented in the following two subsections.

C. Upgrading Top-Level (HTML Document) Loads

To give a detailed picture of the potential of *HTTPS-Only* on top-level (document) loads, we report the results of four measurements: (1) the fraction of successful upgrades to `https` from legacy addresses, where browsers would traditionally connect using the outdated and insecure `http` protocol, (2) the fraction of secure page loads in general, (3) the rate at which users see an *HTTPS-Only* exception page due to the fact that the navigated site does not currently support `https`, and (4) the time required to establish a secure `https` (TLS) connection with a remote website.

1) ***Success rate of top-level (document) upgrades of legacy addresses****:* Our approach specifically aims to ensure connections use the secure `https` protocol, where browsers traditionally would connect using the `http` protocol. To target this data set, our telemetry logs a success when our mechanism is able to upgrade a connection by rewriting the scheme of a URL from `http` to `https` and a load succeeds.

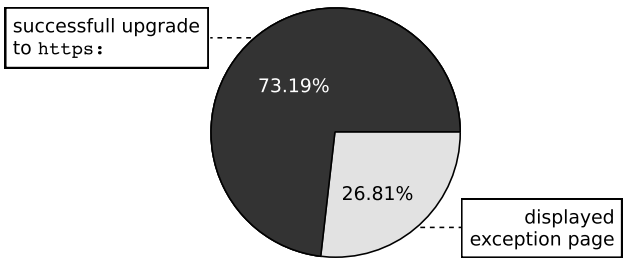

Fig. 3: Attempts to upgrade top-level (document) legacy addresses from `http` to `https` via *HTTPS-Only*.

As illustrated in Figure 3, the *HTTPS-Only* mechanism successfully upgrades top-level (document) loads from `http` to `https` for more than 73% of legacy address. These 73% of successful upgrades originate from the user clicking legacy `http` links, or entering `http` (or even scheme-less) URLs in the address-bar, where the target website, fortunately, supports `https`.

Our observation that *HTTPS-Only* can successfully upgrade seven out of ten of top-level loads from `http` to `https` reflects a general migration of websites supporting `https`. At the same time, this fraction of successful upgrades of legacy addresses also confirms that web pages still contain a multitude of `http`-based URLs where browsers would traditionally establish an insecure connection.

2) ***Rate of secure web page loads****:* In addition to measuring the upgrade ratio of our approach, we further examine the impact of *HTTPS-Only* on a browser's security by measuring the overall fraction of pages that load using `https`. Our telemetry allows us to compute and report the fraction of `http` to `https` connections for all top-level (document) loads.

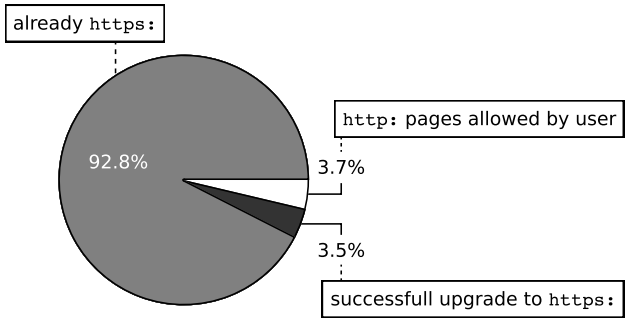

Fig. 4: Use of `https` for top-level (document) loads when *HTTPS-Only* is enabled.

As illustrated in Figure 4, telemetry from participating users in our study reports that in 92.8% of cases the top-level URL is already `https` without any need to upgrade. This rate is higher than the reported 84% of websites relying on `https` which we described in Section *State of the Internet* (see Section 2). We attribute the higher intrinsic `https` rate to the browsing behaviour of participating users. If a secure connection cannot be established, *HTTPS-Only Mode* shows an exception page, where users are then free to choose to abandon the connection attempt and not to visit a site that doesn't support `https`.

We further see that *HTTPS-Only* users experienced a successful upgrade from `http` to `https` in 3.5% of the loads, such that, overall, 96.3% of page loads are secure.

The remaining 3.7% of page loads are insecure, on websites for which the user has explicitly opted to allow insecure connections. Note that the total number of insecure top-level loads observed depends on how many pages a user visited on the website or websites they had exempted from *HTTPS-Only*.

*3) **Rate at which users see an exception page**:* We think that users are likely to perceive frequent exception pages as a degradation of their browsing experience. Hence we believe that the biggest impact to a user's experience when browsing the web using *HTTPS-Only* is the rate at which they will see an exception page. Our recording mechanism allows us to observe how often the user encounters an exception page as a result of *HTTPS-Only* trying to establish a secure `https` connection, but the remote website does not respond positively to the `https` request.

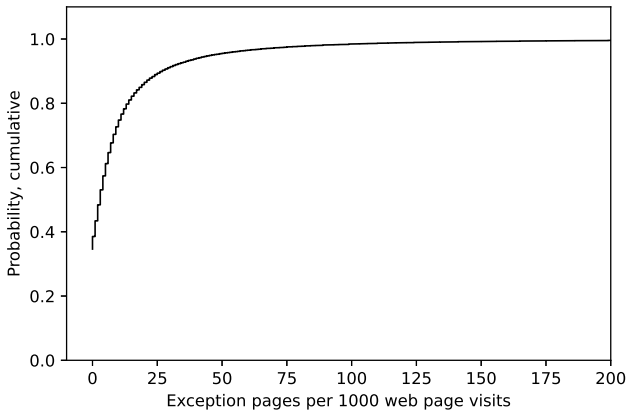

Fig. 5: Cumulative probability distribution of the rate at which users see exception pages as they browse.

We compute the frequency of exception pages seen by users per 1,000 web page visits. When users browse the web using *HTTPS-Only* they experience a range of error page frequencies, depending on the locale of the visited pages and their browsing preferences, hence we audit and examine the distribution.

In summary, the error page distribution, as illustrated in Figure 5, shows:

- The median user experienced 4.3 exception pages per 1,000 page visits
- 34% of users saw no exception pages at all when visiting 1,000 pages, and
- 95% of users saw fewer than 48 exception pages per 1,000 visits.

Our measurements show that across the population, the vast majority of attempts to visit a page securely succeed, and only a small minority of visit attempts result in an error page.

*4) **Time to successfully establish an `https` connection**:* We further record the time it takes the browser to successfully establish an `https` connection to a website. This information allows us to empirically determine after how many milliseconds it is unlikely to be able to connect securely to a website. As explained in Section III, and in particular in Listing 1, we use this information to mitigate connection timeouts and in turn improve the user experience of *HTTPS-Only*.

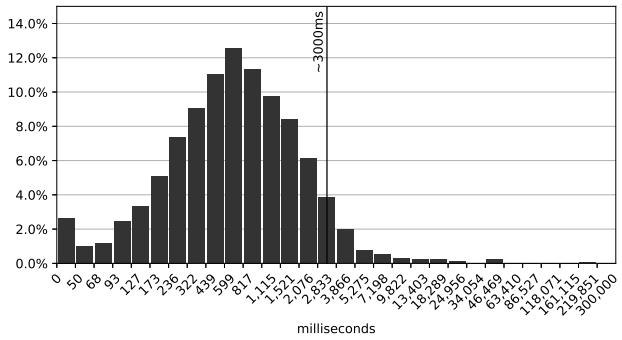

Fig. 6: Time in milliseconds for establishing a top-level (document) `https` connection.

As illustrated in Figure 6, we observe that in more than 96% of cases we are able to establish a secure `https` connection to a top-level page in less than 3,000 milliseconds. Therefore, after 3,000 milliseconds, we launch a background `http` request. If the server-client round-trip time of the background `http` request still manages to establish an `http` connection faster than the upgraded top-level request, then our algorithm cancels the top-level `https` request and shows the user an exception page giving them the option to connect using `http`.

Our approach aims to change the default connection mechanism within a browser and first tries to connect using `https`. Any feasible implementation of *HTTPS-Only*, however, needs to provide some kind of fallback to `http`, even if only as an interim solution. Again, our approach aims to upgrade as many connections to rely on `https` as possible but does not try to eliminate the reachability of legacy websites which do not (yet) support `https`.

D. Upgrading Subresource (Image, Stylesheet, Script) Loads

Similar to what is reported in Figure 4, showing that 93% of top-level loads already rely on `https`, the vast majority of subresource loads also already rely on `https`.

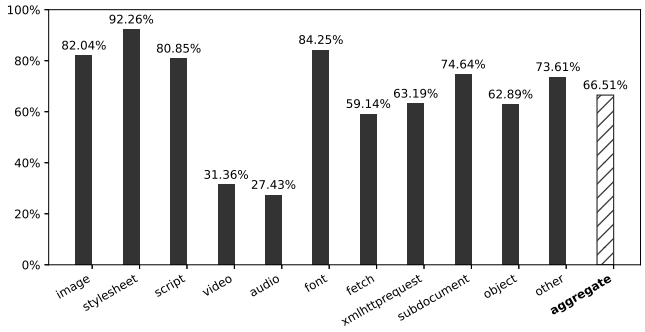

Fig. 7: Successful subresource upgrades of legacy addresses from `http` to `https` via *HTTPS-Only*.

Here, however, we focus on the success rate of *HTTPS-Only* for the minority of links using legacy `http` URLs, where browsers traditionally would carry out the subresource load using the insecure `http` protocol. Our recording logs a successful upgrade when *HTTPS-Only* is able to rewrite the scheme of a legacy URL from `http` to `https` and the subresource load succeeds.

Image, Stylesheet, Script: As illustrated in Figure 7 our approach is able to upgrade the majority of the three fundamental subresources for legacy addresses, namely resources of type image in 82% of cases, stylesheet in 92% of cases, and script in 80% of cases. In practice, website maintainers sometimes focus on securing the top-level document by making it available over `https` but do not update hyperlinks for subresources correspondingly. We believe that the high success rate reported is due to the fact that legacy image, style and script tags are mostly same-origin with the top-level document and therefore mostly have a secure version available.

Audio, Video, Font: Insecurely linked subresources of type video, audio or font appear much less often on websites than images, stylesheets or scripts. Our logging shows that on an average day we record 3,906 insecure video requests, 2,073 insecure audio requests and 3,079 insecure font-face requests. In contrast, on an average day we record 1,189,977 insecure image requests. As illustrated, we record successful upgrades for insecure video in 31% of cases, insecure audio in 27% of cases and insecure font-face in 84% of cases. We think that respective tags for audio and video gained popularity only recently - during the same time websites increasingly started to rely on `https` - in practice most video is streamed from services like `youtube.com` which all rely on `https` to begin with, meaning that our upgrading mechanism does not kick in.

Fetch and XMLHttprequest: Although fetch and XMLHttpRequest provide similar functionality, our recording shows that users encounter insecure XMLHttpRequests more than three times as often as insecure fetch requests. Most likely because the fetch specification gained popularity recently and developers were already aware of the security drawbacks of not using `https`. We see that both perform similarly in terms of upgrade success - almost 60% of fetch upgrades and more than 63% of XMLHttpRequests upgrades succeed.

Subdocument and Object: Loading a subdocument is in fact very similar to loading a top-level document, with the important difference being that the subdocument load appears in a frame or iframe. Hence the success ratio mirrors what we report in Figure 3: our approach is able to upgrade 74% of insecure subdocument loads from `http` to `https`.

Other: Firefox internally distinguishes between over 50 different content types for loading resources. We explicitly list the success ratio of the the most popular types above, and label the remaining types, such as download or web-manifest, as "other." Our measurements indicate a success rate of around 73% for all the types assigned to this bucket.

Summary of upgrades in aggregate: As mentioned above, we observe that the majority of subresources within an `https` document are already `https` in the first place. For the remaining legacy-linked subresources, which would load using `http` in a traditional browser setting, our approach is able to upgrade 66% in aggregate. Again, these 66% of successful upgrades originate from our approach being able to upgrade legacy `http` URLs, where browsers traditionally would carry out the subresource load using the insecure `http` protocol.

VI. Related Work

Relying on `https` is key to providing protection for any traffic between a web browser and the server. Studies conducted by Sivakorn et al. [30], [31], Drakonakis et al. [3] and Englehardt et al. [5] confirm the problematic situation of websites not properly deploying `https`. These studies conclude that websites which do not, or only partially, deploy `https` expose private information to attackers. Our approach mitigates the risk of exposing private user information by always encrypting and delivering content using `https`.

In 2008, Jackson and Barth proposed a mechanism [17] that allows web servers to force browsers to interact with a website only using secure `https` connections. This mechanism built the foundation for the HSTS specification [14] which, if properly deployed [19], allows websites to protect themselves against protocol downgrade attacks. However, the initial request remains unprotected from active attacks. To mitigate this risk, all major browsers deploy HSTS Preload lists which contain known websites supporting HSTS, thereby allowing a browser to perform the initial request over `https`. Unfortunately, HSTS Preload lists do not scale to the size of the internet and therefore leave the majority of websites unprotected against downgrade attacks. Further, HSTS has the downside that it opens up an additional tracking vector [28], [33]. Our proposed *HTTPS-Only* mechanism overcomes these limitations and automatically updates all requests so that websites can rely on `https`, starting with the initial request.

Statistics gathered by Porter Felt et al. [6] show that by 2017 over 50% of web browsing took place over `https`. In 2020, *Let's Encrypt* reported that their free, automated, and open Certificate Authority issued more than 1.5 million certificates daily. These numbers confirm that `https` is on the rise, and hence different approaches are being launched to try to deliver more content over `https`. For example, the Content-Security-Policy [32] (CSP) hosts the directive `upgrade-insecure-requests` [37], which provides a mechanism for websites to upgrade all of their requests from `http` to `https`. While this opt-in directive provides a mechanism for sites to upgrade legacy content within their own host range, it requires active deployment by the website maintainer.

The famous browser extension *HTTPS Everywhere* [4], and related mechanisms *Smart HTTPS* [15] and *HTTPZ* [2] allow browsers to fully encrypt communications with many major websites. The downside of any extension approach is that browsers typically block active mixed content before the extension gets the opportunity to upgrade a request.

Finally, the World Wide Web Consortium (W3C) is working on a specification for auto-upgrading mixed content [39] and the *Chrome* browser recently announced that it would ship this behaviour [7]. Ultimately, all upgrading mechanisms have to acknowledge that web architecture permits resource requests to return different content when queried using `http` or `https`. As discussed by Paracha et al. [27] the number of sites that exhibit such behaviour is small and we argue that this problem will vanish over time once `https` becomes the norm. While auto-upgrading mixed content only upgrades subresources of type image and media, our *HTTPS-Only* approach upgrades all subresources, and additionally attempts to upgrade top-level (document) connections to rely on `https`.

VII. Discussion

Currently, all major browsers (Chrome, Firefox, Edge, Safari) default to using `http` when the user enters a schemeless URL into the address-bar. Similarly, all web browsers, by default, do not try to upgrade the scheme of the URL to `https` when the user clicks an outdated legacy `http` link or enters an `http` URL into the address-bar. We attribute this industry-wide behaviour of relying on the insecure `http` protocol rather than its secure successor to the desire of browser vendors to avoid downgrading a user's experience. Unfortunately, relying on a processing model which sends the initial request to a website in cleartext jeopardises the security and privacy of end users because it allows attackers to prevent subsequent upgrades to a secure connection.

Currently, it seems that browsers rely in on websites following best practice and redirecting an initial request to `https`. To illustrate the severity of this security shortcoming we note that the web application security community has been focussing on exactly this problem by introducing new mechanisms including HSTS and CSP's `upgrade-insecure-requests`. Both of these approaches require considerable effort from the website, however. For example, it is almost impossible for the developer of a website that uses HSTS to tell whether a given browser accessing the website has received HSTS information for the website. Getting a domain on a browser's HSTS Preload list, so that the browser only uses secure connections to interact with a website, presents an additional burden for website developers. Given the billions of websites, it is questionable whether HSTS Preload lists can ever scale to accommodate most of the web.

We argue that a better approach for user agents is to first try to establish a secure connection. This approach reduces the burden for website authors, and puts the security back under the control of user agents, operating in the best interests of the user. Requiring an explicit user opt-in to insecure `http` connections provides further protection against active downgrade attacks. Thus we believe that browser vendors should consider integrating some kind of native support similar to our proposed *HTTPS-Only Mode*, which first tries to establish a secure connection to a website and only falls back to an insecure connection in the case where the website is not reachable using the `https` protocol.

To summarize: a large majority of web connections already support `https`. Because the percentage of web pages loaded using `https` currently exceeds 84%, as reported by *Let's Encrypt* [21], we believe that we are entering a final phase of securing the web, where we anticipate that, ultimately, almost every website will be available over `https`. We think that the time has come for browser vendors to reconsider the default protocol used for establishing a connection to a website. Additionally, the presented upgrading mechanism has the ability to compensate for legacy `http` links, incorrectly entered URLs in the address-bar or incorrectly linked resources in websites, and therefore provides a mechanism to silently fix security compatibility issues. We think that the extra protection provided by explicitly prompting users to opt-in before they connect insecurely to a website is becoming more feasible now that such prompts need be shown only occasionally.

VIII. Conclusion

We have presented *HTTPS-Only*, a new browser connection model which first tries to establish a secure connection to a website using `https` and only allows a fall back to `http` if a secure connection cannot be established. Additionally, our approach silently upgrades all insecure `http` subresources (image, stylesheet, script) within a secure website to use `https`.

Evaluating real-world browsing behaviour of over 100,000 Firefox release users shows that our approach is capable of upgrading over 70% of top-level (document) loads to rely on `https`, where browsers traditionally would connect using the insecure and outdated `http` protocol. The success of *HTTPS-Only* in upgrading legacy links, together with our observation that the median user can successfully connect to a website using `https` 995 times when visiting 1,000 sites, suggests that our approach is useful and practical. We believe that *HTTPS-Only* highlights the path to achieving a fully secure web, and that now is a good time for browser vendors to begin re-evaluating their current connection model.

Acknowledgments

We would like to thank everyone in Security Engineering at Mozilla for their feedback, reviews, and insightful comments. In particular we are grateful to following Mozillians for making *HTTPS-Only Mode* possible: Meridel Walkington, Eric Pang, Martin Thomson, Steven Englehardt, Alice Fleischmann, Angela Lazar, Mikal Lewis, Wennie Leung, Frederik Braun, Tom Ritter, June Wilde, Sebastian Streich, Leli Schiestl, Daniel Veditz, Prangya Basu, Dragana Damjanovic, Valentin Gosu, Chris Lonnen, Andrew Overholt, and Selena Deckelmann.

References

[1] Chromium Project. Check HSTS preload status and eligibility. https://hstspreload.org/, 2012. (checked: January, 2021).

[2] claustromaniac. HTTPZ. https://github.com/claustromaniac/httpz, 2020. (checked: January, 2021).

[3] K. Drakonakis, S. Ioannidis, and J. Polakis. The cookie hunter: Automated black-box auditing for web authentication and authorization flaws. In *Proceedings of the Conference on Computer and Communications Security*, 2020.

[4] EFF. HTTPS:// Everywhere. https://www.eff.org/https-everywhere, 2014. (checked: January, 2021).

[5] S. Englehardt, D. Reisman, C. Eubank, P. Zimmerman, J. Mayer, A. Narayanan, and E. W. Felten. Cookies that give you away: The surveillance implications of web tracking. In *Proceedings of the International Conference on World Wide Web*, 2015.

[6] A. P. Felt, R. Barnes, A. King, C. Palmer, C. Bentzel, and P. Tabriz. Measuring HTTPS Adoption on the Web. In *USENIX Security Symposium*, 2017.

[7] Google. No More Mixed Messages About HTTPS. https://blog.chromium.org/2019/10/no-more-mixed-messages-about-https.html, 2020. (checked: January, 2021).

[8] IETF. Uniform Resource Locators (URL)). https://tools.ietf.org/html/rfc1738, 1994. (checked: January, 2021).

[9] IETF. The "data" URL scheme. https://tools.ietf.org/html/rfc2397, 1998. (checked: January, 2021).

[10] IETF. Hypertext Transfer Protocol – HTTP/1.1). https://tools.ietf.org/html/rfc2616, 1999. (checked: January, 2021).

[11] IETF. HTTP Over TLS. https://tools.ietf.org/html/rfc2818, 2000. (checked: January, 2021).

[12] IETF. The Web Origin Concept. https://tools.ietf.org/html/rfc6454, 2011. (checked: January, 2021).

[13] IETF. The WebSocket Protocol. https://tools.ietf.org/html/rfc6455, 2011. (checked: January, 2021).

[14] IETF. HTTP Strict Transport Security (HSTS). https://tools.ietf.org/html/rfc6797, 2012. (checked: January, 2021).

[15] ilGur. Smart HTTPS. https://mybrowseraddon.com/smart-https.html, 2020. (checked: January, 2021).

[16] Internet Live Stats. Total number of Websites. http://www.internetlivestats.com/total-number-of-websites/, 2014. (checked: January, 2021).

[17] C. Jackson and A. Barth. Forcehttps: protecting high-security web sites from network attacks. In *Proceedings of the International Conference on World Wide Web*, 2008.

[18] C. Kerschbaumer. Enforcing Content Security by default within Web Browsers. In *Proceedings of Cybersecurity Development Conference*, 2016.

[19] M. Kranch and J. Bonneau. Upgrading https in mid-air: An empirical study of strict transport security and key pinning. In *NDSS*, 2015.

[20] J. Larisch, D. Choffnes, D. Levin, B. M. Maggs, A. Mislove, and C. Wilson. Crlite: A scalable system for pushing all tls revocations to all browsers. In *Symposium on Security and Privacy*, 2017.

[21] Let's Encrypt. Let's Encrypt Stats. https://letsencrypt.org/stats/, 2020. (checked: January, 2021).

[22] Moxie Marlinspike. New Tricks For Defeating SSL In Practice. https://www.blackhat.com/presentations/bh-dc-09/Marlinspike/BlackHat-DC-09-Marlinspike-Defeating-SSL.pdf, 2009. (checked: January, 2021).

[23] Mozilla. Preloading HSTS. https://blog.mozilla.org/security/2012/11/01/preloading-hsts/, 2012. (checked: January, 2021).

[24] Mozilla. Insecure password warning in Firefox. https://support.mozilla.org/en-US/kb/insecure-password-warning-firefox, 2017. (checked: January, 2021).

[25] Mozilla. Mozilla Data Documentation. https://docs.telemetry.mozilla.org/, 2020. (checked: January, 2021).

[26] Mozilla. Telemetry Portal. https://telemetry.mozilla.org/, 2020. (checked: January, 2021).

[27] M. T. Paracha, B. Chandrasekaran, D. R. Choffnes, and D. Levin. A Deeper Look at Web Content Availability and Consistency over HTTP/S. 2020.

[28] Paul Syverson and Matthew Traudt. HSTS supports targeted surveillance. In *USENIX Security Symposium*, 2018.

[29] B. Santesson, M. Myers, R. Ankney, A. Malpani, S. Galperin, and C. Adams. X.509 Internet Public Key Infrastructure - Online Certificate Status Protocol - OCSP. https://tools.ietf.org/html/rfc6960, 2013. (checked: January, 2021).

[30] S. Sivakorn, A. D. Keromytis, and J. Polakis. That's the way the cookie crumbles: Evaluating https enforcing mechanisms. In *Proceedings of the Workshop on Privacy in the Electronic Society*, 2016.

[31] S. Sivakorn, I. Polakis, and A. D. Keromytis. The cracked cookie jar: HTTP cookie hijacking and the exposure of private information. In *Symposium on Security and Privacy*, 2016.

[32] S. Stamm, B. Sterne, and G. Markham. Reining in the Web with Content Security Policy. In *World Wide Web*, 2010.

[33] M. Traudt and P. Syverson. Does Pushing Security on Clients Make Them Safer? https://petsymposium.org/2019/files/hotpets/proposals/pushing-insecurity-hotpets19.pdf, 2019.

[34] W3C. Document Object Model (DOM). http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407/DOM3-Core.pdf, 2004. (checked: January, 2021).

[35] W3C. Cross-Origin Resource Sharing (CORS). http://www.w3.org/TR/cors, 2010. (checked: January, 2021).

[36] W3C. Mixed Content. https://www.w3.org/TR/mixed-content/, 2014. (checked: January, 2021).

[37] W3C. Upgrade Insecure Requests. https://www.w3.org/TR/upgrade-insecure-requests/, 2015. (checked: January, 2021).

[38] W3C. File API - The Blob Interface and Binary Data. https://w3c.github.io/FileAPI/#blob-section, 2020. (checked: January, 2021).

[39] W3C. Mixed Content Level 2. https://w3c.github.io/webappsec-mixed-content/, 2020. (checked: January, 2021).

[40] WHATWG. HTML. https://html.spec.whatwg.org/, 2020. (checked: January, 2021).

[41] WHATWG. XMLHttpRequest - Living Standard. https://xhr.spec.whatwg.org/, 2020. (checked: January, 2021).