TaintScript: An Intuitive Information-Flow Tracking System for Web Browsers

Matthew Shabet
Harvard University
mshabet@college.harvard.edu

Alex Encalada-Stuart
Harvard University
alex_encaladastuart@college.harvard.edu

Abstract

Encrypted communication is one of the fundamental components of modern web security. Web developers must ensure that sensitive user data sent from client to server maintains confidentiality, integrity, and authenticity through the use of secure communication protocols. However, various exploits allow adversaries to subvert this layer of security and reduce communications to an insecure protocol, which plainly exposes client data to attackers. Web developers must be vigilant in preventing sensitive data from being sent over compromised channels, however at present there is no easy-to-use framework for doing so. We present TaintScript, a testing framework for client-side JavaScript files that uses information flow control to identify locations where user data marked as sensitive leaves the browser via an insecure protocol, highlighting potential issues that should receive attention from the developer.

CCS Concepts: • **Security and privacy → Web protocol security**; **Information flow control**.

Keywords: encrypted communication, web security, taint tracking

1 Introduction

While over 90% of online web traffic is encrypted [9], certain attacks such as man-in-the-middle attacks and protocol downgrade attacks can coerce a browser into using an insecure protocol like HTTP instead of the encrypted HTTPS. Such an attack leaves data sent to and from the server wide open to eavesdropping and modification, which developers would do well to avoid if the data being exchanged is sensitive. Certain software developments have been created in an attempt to help mitigate this problem - for example, a server can mark cookies sent to a client with the 'Secure' attribute which only allows the cookie to be sent over an HTTPS channel [5]. This ensures that sensitive data stored in 'Secure' cookies by the client will only be sent over the network if the connection is adequately encrypted, protecting their confidentiality [1]. However, at this time only cookies have a simple feature like this that protects them from being sent over insecure channels, even though there are many other types of sensitive data that can be exchanged with the server such as JavaScript objects or data in web storage. Web developers may want to ensure that these elements are also protected from insecure channels, however there is no easy-to-use protection akin to the Secure cookie attribute for these types of sensitive data. This is the problem to which we propose a solution.

The attack model that we wish to defend against is one where the client's communication with the server is subverted to be insecure. However, there does not yet exist a simple framework that allows web developers to check if and where their application is liable to leak sensitive information over an insecure protocol. As discussed in the related work section, some existing frameworks are too narrow in scope and only focus on certain features like browser extensions. Meanwhile, while others give developers access to methods that can assign and check taint, these methods have to be manually inserted by the developer at each point of interest, which is nontrivial and prone to human error when working with larger applications. The current lack of an easy and manageable way for developers to ensure that secure data is protected presents a security risk, since many developers may not be experienced enough to know how to guarantee that their sensitive data will not be sent over an insecure connection. Or, maybe an application is too large to go through manually without facing the risk of human error. In particular, a developer might wish to verify that a script which is not intended to send sensitive data from the client to the server actually upholds this intention, and a developer who does intend to send some sensitive information from client to server may want to identify each location where this occurs.

We present TaintScript, an intuitive framework that allows developers to check their scripts for secure data leakage before they are put into production in a manner that is easy-to-use and requires minimal modification of existing code. This technology aims to be straightforward to integrate and simple to use so that web developers can efficiently and effortlessly apply it to their projects. Accessibility of security testing mechanisms is important, since a service that is easy to use encourages widespread adoption and facilitates a greater extent of security in the web development community. We provide an overview of related work in Section 2. In Section 3, we describe how to use the software, followed by a technical description of how it works. The paper will then conclude with an evaluation of TaintScript's abilities and suggestions on how it could be improved in the future.

2 Related Work

To the best of our knowledge, a simple and intuitive taint tracking mechanism has not yet been implemented in web browsers for the purpose of preventing leakage of sensitive data over insecure protocols. We will now discuss related work which attempts to guarantee the use of a secure protocol, specifically HTTPS, for sensitive data transfers through various methods on both the server- and client-sides. We also discuss prior work in browser taint tracking, which broadly consists of general-purpose taint trackers and taint trackers focused on preventing cross-site scripting, both of which have been implemented through modifications to browser JavaScript engines and potentially other browser components.

2.1 Ensuring Sensitive Data Transfer Protection

Existing methods of ensuring that sensitive data is protected via HTTPS involve trade-offs between the simplicity of the implementation and the granularity of the data protected. The simplest and most coarse-grained way for a web application to ensure this protection is to only allow communication over HTTPS. This completely prevents all information, including sensitive data, from being exchanged over an insecure protocol. However, HTTPS-only communication must be implemented in one of a few ways, each with some shortcomings. In a naïve implementation, a server could simply refuse all HTTP connections, but this would be impractical, since the server likely wants to suggest the client reconnect over HTTPS instead. To this end, the server can instead listen for HTTP requests, and upon receiving one, simply redirect the client to the HTTPS version of the desired web resource. This implementation still has a vulnerability, because the client's initial connection over HTTP is vulnerable to man-in-the-middle attacks, which could, among other things, allow an attacker to imitate the webpage over HTTP [3]. To reduce the attack surface for this vulnerability, a server can employ the HTTP Strict-Transport-Security (HSTS) header, which instructs a browser to only connect to the server over HTTPS for future requests. However, this requires the client to connect to the server at least once over HTTPS, since the HSTS header is only acknowledged by the browser when sent over an HTTPS connection. [12]. Also, only around 16.1% of websites incorporate it, which makes this option unavailable to the majority of web developers [15].

The use of HTTPS only can also be enforced on the client side, including by the Firefox browser's "HTTPS-Only Mode," which blocks all requests made over HTTP, giving a warning to the client which they can then opt to ignore and proceed with the request if they desire [10]. The popular browser extension "HTTPS Everywhere" takes a different approach, using rulesets for a large set of supported websites to redirect a client from HTTP to an HTTPS version of a page for a supported site whenever one is available [6]. HTTPS Everywhere also includes the option to block all non-HTTP requests, as in Firefox's HTTPS-Only Mode. However, many web users do not use this functionality.

At a more granular level, a web application can attempt to ensure that specific client data is only sent over HTTPS by posting to HTTPS from an HTML form containing the data the web developer wants to be secure [11]. This ensures that the data in the form is sent over HTTPS. There are no problems with this method when the page containing the form has itself been to the client over HTTPS. However, posting form data to HTTPS from an HTTP-loaded page is insecure, since it still allows for the page loaded over HTTP to have been compromised, allowing for a man-in-the-middle attack to subvert the intended protection of the HTTPS post [3, 8]. On the other hand, a page loaded over HTTPS posting to HTTP also poses a security risk, since loading the form over HTTPS gives the appearance of protection for the data which is sent out, even though it is not actually HTTPS-protected.

2.2 Browser Taint Tracking

Taint tracking, the observation of how data of interest flows through an execution path [7], has been implemented for web browsers in a number of prior works. TaintSNIFFER [13] implements dynamic taint tracking in Microsoft Research's experimental C3 web browser, taking advantage of browser's common .NET framework for its HTML-, CSS-, and JavaScript-processing components to make components throughout the browser taint-aware. TaintSNIFFER adds taint(), untaint(), and checkTaint() functions to the C3 JavaScript engine's set of pre-defined functions to allow for tainting, untainting, and checking the taint of JavaScript objects at any point in execution [2]. It implements taint tracking of resources across the browser, including cookies, history, bookmarks, and the DOM to create a framework which can be used to enforce a wide variety of policies. This includes policies to limit or prevent the ability of scripts and extensions to access tainted data, including to exfiltrate it during execution. A helpful example of this property given by the authors of [2] are policies designed to protect the data of a password manager browser extension, which might otherwise be vulnerable to a variety of attacks by malicious scripts on the web pages a user visits. This includes logging the password inputted by the extension, opening new pages to capture the user's passwords for other sites, and fabricating form submit events for other sites.

Browser taint tracking has also been implemented for more specific goals. [14] adds taint tracking code to Chromium by modifying the V8 JavaScript engine and the HTML rendering engine to guard against cross-site scripting (XSS) by tracking potentially attacker-provided strings, like a URL, and preventing their execution if they become compiled into JavaScript code. Finally, [4] offers a taint analysis framework

designed specifically for browser extensions, leveraging a novel hybrid taint tracking approach which performs static analysis on the available code for extensions complemented with a modified version of the V8 engine and Chromium to support dynamic taint tracking. The framework in [4] aims to use taint tracking to examine how safely data is handled by a variety of publicly-available browser extensions.

3 TaintScript Prototype

The TaintScript prototype consists of a translate.js file, which is executed with a base JavaScript file as input. Once the execution finishes, it prints out a new script which is identical to the base script except there is additional taint tracking code added in specific places - an example of this is in Section 3.3, where we show a before and after of applying TaintScript. With that said, what follows is a detailed description of how to use TaintScript to identify instances of insecure data transmission.

3.1 How to use TaintScript

1. First, the developer must go through the scripts they want to apply taint tracking to and find all instances of variables that are directly assigned sensitive values. For each one of these variables, the only change the developer needs to make to the script is to prefix each of these variable names with the string "TSsecure_". This label signals to the translator script that these variables should be marked as secure - this is the only way that taint can be introduced into the system. Taint is then propagated throughout the execution of the program. The label system also means that the developer cannot have any preexisting variable names which begin with "TSsecure_" unless they want the interpreter to taint this variable.

2. The developer then runs their JavaScript files through the translator script to produce new JavaScript files that include tainting logic. The translate.js script is intended to be run from the command line - it takes in a JavaScript file as a command-line argument, and then outputs the new file to the terminal. For example, if you choose to use node, the following command will take in a specified input file and output the result to a specified location:

```
node translate.js [input_file] > [output_file]
```

Ensure that the packages "esprima" and "escodegen" are installed in order for the script to work. If successful, a new file will be produced in the specified output path that contains all the taint tracking logic. Run this script for each file in which the developer wants to track taint.

3. Now, the developer should create a test server on which they can test the application. For each of the scripts that were translated, replace the old version on the test server with each of the newly created ones. Start up the test server and run the application, now using the translated JavaScript files that have taint tracking.

4. From here, the developer just needs to simulate client behavior over an insecure protocol - in our prototype, the insecure protocols we specifically checked for were HTTP and FTP. As the sensitive elements of the page are interacted with, the sensitive JavaScript files that work with those elements will begin to run and propagate taint throughout its execution. If at any point the script detects that sensitive data is sent out over this insecure channel, then a message of the following form will be outputted to the console:

```
TS ALERT: Object <name> sent over <protocol> via
<output_channel>
```

5. If the developer receives no such messages and has simulated all possible client interactions, then they can rest easy that their application is secure. On the other hand, if a TS ALERT message is sent to the console, then the response of the developer will vary on the situation. In the case where the developer did not intend for any secure data to leave the client browser, and they checked that it wasn't an instance of overtainting, then it could indicate that there is some bug or security vulnerability that allows sensitive data to be exfiltrated. On the other hand, if the developer did intend to send out sensitive data, then they need to make a creative decision on how to respond. One possible option is to check the current communication protocol before sending out sensitive data and refuse if the connection is not secure, which has the potential to break the functionality of the site at the benefit of not leaking sensitive data. The developer needs to decide how they want their site to function in such a circumstance and respond accordingly. If the developer opts to fix this issue and remove all TS ALERT messages, then they can modify the original JavaScript file, run it through the translator, and run it in the test server again until the problem is solved.

3.2 How translate.js works

To transform the base JavaScript file into one with taint tracking, the translator script follows a clearly defined procedure. For reference, we provide an example of what the output of the translation process looks like in the following section. With that said, the workflow is as follows:

1. Load the base file into memory and use Esprima to convert it into a syntax tree

2. The prototype assumes that the base file is a collection of independent methods: more specifically, running the script directly should do nothing, and instead each method will be invoked by a certain action on the webpage. So, each method in the body of the script will be tainted and tracked separately. For each of these methods, do the following procedure:

3. Add three empty arrays to the top of the method: TSobjects, TSlocalStorage, and TSsessionStorage. The TSobjects array contains the names of all tainted variables, meanwhile TSlocalStorage and TSsessionStorage contains the keys of all tainted entries in the localStorage and sessionStorage respectively.

4. Traverse through each element of the syntax tree that sits directly in the body of the method. Each element falls into one of three categories, which determines how it is parsed. The three relevant types of elements are variable declarations, if statements, and expression statements.

4a. Variable declarations: For variable declarations, first check if the name of the variable is preceded by "TSsecure_". If so, then insert a new element into the syntax tree above the current element. This new element is a JavaScript chunk, which adds the name of the variable into the TSobjects array. Otherwise, to determine if the newly declared variable is tainted, you need to check if the intialized value comes from a tainted source. So, we add a new JavaScript chunk above the declaration with an if statement that adds the variable to the TSobjects array conditional on the right side being tainted. The way we check if the right side is tainted is using a switch statement - if there's only one variable on the right, then just check if it's tainted. If there's a binary expression that involves two variables, check if either one is tainted. If there is a pull from web storage, then check if the lookup key is in either of the arrays that track tainted webstorage elements. These four checks are all the tainting checks that we currently have for variable declarations, and more can easily be added in future versions.

4b. If statements: For an if statement, we add a JavaScript chuck above each variable assignment in the body of the if, else if, and else sections. If the branch was conditional on the value of a certain variable, then this chunk will add the assignment variable to the TSobjects array if the branching variable was tainted.

4c. Expression statements: With expressions, we only check two things: if we are putting a value into web storage, or if we are trying to call a method that is known to transmit data. We will add a code chunk above the web storage expression that adds the key being stored to the corresponding tainted array if the value being stored is tainted. For methods that transmit data, we add a chunk that tests if the current communication protocol is insecure and if the values being sent out are tainted. If so, then an alert is outputted to the console for the developer to see.

5. Once we hit the end of the method and all elements of the tree have been parsed, the new syntax tree should have at least a few new elements added to it that are responsible for tracking, propagating, and responding to taint. The final step is to take the new syntax tree and use Escodegen to print the new file.

3.3 An example

In Figure 1 we can see a simple example of a base JavaScript file. It grabs an element from a form and marks the variable that contains it as sensitive. Then, it does some arbitrary JavaScript activity, assigning new values and setting/getting items from web storage. At the end, it sends out data over various communication channels - the script was written such that the data is always tainted, which our taint tracker correctly catches.

In Figure 2 you can see the output of the translate.js file. The blue lines were added by the framework, and they push new values to TSobjects, TSlocalStorage, and TSsession storage as necessary while also checking for insecure transmissions at the end.

```
// Initialize the sensitive variable with data from the form
var TSsecure_val0 = document.getElementById('rec').value;

// Simulate JavaScript activity
var val1 = TSsecure_val0;
var val2;
if (val1 == "yes") {
        val2 = " recommends it";
} else if (val1 == "no") {
        val2 = " does not recommend it";
}
var val3 = document.getElementById('name').value;
var val4 = val3 + val2;
sessionStorage.setItem('val4', val4);
var val5 = sessionStorage.getItem('val4');
localStorage.setItem('val5', val5);
var val6 = localStorage.getItem('val5');

// Now attempt to send the user data to the server
// 1. Send over the Fetch API
fetch("http://127.0.0.1:5000/", {method: "POST", body: "Fetch API: " + val6});
// 2. Send over using an XMLHttpRequest
var xhr = new XMLHttpRequest();
xhr.open("POST", "http://127.0.0.1:5000/", true);
xhr.send("XMLHttpRequest: " + val6);
// 3. Send over WebSocket
var socket = io();
socket.send(val6);
```

Figure 1. Before passing through TaintScript.

4 Evaluation

Our current prototype is designed to be an illustrative example rather than a fully robust piece of production software. As a result, it expects the input file to be in a specific format, the tainting logic only handles a few specific cases, and the AST parsing is rudimentary. However, this is fine, since the logic that we currently have implemented is correct within a defined scope and working as intended. Also, our project was never about innovating on the taint tracking logic, but instead on creating an intuitive and easy-to-use taint tracker.

```
var TSsessionStorage = new Array();
var TSlocalStorage = new Array();
var TSobjects = new Array();

// Initialize the sensitive variable with data from the form
TSobjects.push('TSsecure_val0');
var TSsecure_val0 = document.getElementById('rec').value;

// Simulate JavaScript activity
if (TSobjects.indexOf('TSsecure_val0') >= 0) {
    TSobjects.push('val1');}
var val1 = TSsecure_val0;
var val2;
if (val1 == 'yes') {
    if (TSobjects.indexOf('val1') >= 0) {
        TSobjects.push('val2');}
    val2 = ' recommends it';
} else if (val1 == 'no') {
    if (TSobjects.indexOf('val1') >= 0) {
        TSobjects.push('val2');}
    val2 = ' does not recommend it';
}
var val3 = document.getElementById('name').value;
if (TSobjects.indexOf('val3') >= 0 || TSobjects.indexOf('val2') >= 0) {
    TSobjects.push('val4');}
var val4 = val3 + val2;
if (TSobjects.indexOf('val4') >= 0) {
    TSsessionStorage.push('val4');}
sessionStorage.setItem('val4', val4);
if (TSsessionStorage.indexOf('val4') >= 0) {
    TSobjects.push('val5');}
var val5 = sessionStorage.getItem('val4');
if (TSobjects.indexOf('val5') >= 0) {
    TSlocalStorage.push('val5');}
localStorage.setItem('val5', val5);
if (TSlocalStorage.indexOf('val5') >= 0) {
    TSobjects.push('val6');}
var val6 = localStorage.getItem('val5');
// Now attempt to send the user data to the server
// 1. Send over the Fetch API
if (TSobjects.indexOf('val6') >= 0 && (location.protocol == 'http:' || location.protocol == 'ftp:')) {
    console.log('TS ALERT: Object (val6) sent over ' + location.protocol + ' via the Fetch API');}
fetch('http://127.0.0.1:5000/', {
    method: 'POST',
    body: 'Fetch API: ' + val6});

// 2. Send over using an XMLHttpRequest
var xhr = new XMLHttpRequest();
xhr.open('POST', 'http://127.0.0.1:5000/', true);
if (TSobjects.indexOf('val6') >= 0 && (location.protocol == 'http:' || location.protocol == 'ftp:')) {
    console.log('TS ALERT: Object (val6) sent over ' + location.protocol + ' via XMLHttpRequest');}
xhr.send('XMLHttpRequest: ' + val6);

// 3. Send over WebSocket
var socket = io();
if (TSobjects.indexOf('val6') >= 0 && (location.protocol == 'http:' || location.protocol == 'ftp:')) {
    console.log('TS ALERT: Object (val6) sent over ' + location.protocol + ' via WebSockets');}
socket.send(val6);
```

Figure 2. After passing through TaintScript.

To this end, we believe that we have succeeded. The developer has to do minimal work prefixing certain variables, and then the translation software does all the work. With our software, the process of adding taint tracking is fast, automated, and easy to complete, so it won't be prone to human errors. On top of that, the translated script is not much slower that the original file so the developer can expect similar speeds when testing. However, we recognize that there are certain shortcomings of the current prototype, and there are some features that we could include to make TaintScript even easier to use. We will detail all of these in the subsequent Future Work section.

5 Future Work

There are many ways in which the project can be expanded going forward:

1. Increase the compatibility of the translation script
As it stands now, the script assumes certain features about the input script. For example, the translation script will throw an error if conditional statements do not branch on a binary operator or if the arguments passed into a method are not in a particular format, among many other things. To be able to work on arbitrary input files, additional logic must be added that handle more complex input scripts.

2. Fixing undertainting
Currently there is some undertainting, as the translation script will not enter into user-defined methods or propagate taint if there are more than two variables to the right of an assignment (so a=b+c tracks taint but not a=b+c+d). In order to increase the set of input scripts that are compatible with our framework, we need to add more logic to the translation script that can properly propagate taint through different types of code structures. Resolving this is critical for the efficacy of the software, since undertainting implies that sensitive data could be transmitted insecurely without being caught by the taint tracker.

3. Fixing overtainting
Currently the prototype also overtaints - for example, there are edge cases like $a = b - b$ or $a = b \wedge b$ where a should not be tainted even if b is tainted since the result is a constant. Resolving this is not as critical as undertainting, however it would make the work of the web developers much easier as they won't have to backtrack to find false positives as much.

4. Increase the tracking on more output channels
Currently the script only monitors exfiltration through the Fetch API, XHRs, and WebSockets. However, while these are certainly the most popular ones, there are many other output channels (namely WebRTC) that are currently not being tracked and should be added in later for maximum compatibility.

5. Naming collisions
The current setup assumes that no insensitive variables in the base script are prefixed by "TSsecure_" and that there were no identifiers called TSobjects, TSlocalStorage, or TSsessionStorage. If an identifier with one of these names did exist, then there would be a naming collision where these two different objects were given the same name. While the existence of identifiers with these exact names is not likely, it is still possible and not terribly hard to fix.

6. Batch translating

If the developer wants to apply taint tracking to a large number of files, they will have to manually run the translation script on each one, then manually replace each of the files in the test server. While you can use shell scripting to remedy this, we could also add functionality to TaintScript to make the whole process more smooth.

7. Include some way of simulating client behavior
Right now, it is on the onus of the developer to check every path of the script that the client could take by clicking on buttons and interacting with various elements on the page. Professional software engineers likely already have a testing suite, however many developers likely do not. It would be ideal if we included one with high coverage and reasonable performance. That's probably the largest addition we could add to make the framework even easier to use.

6 Conclusion

In this paper we have presented TaintScript, a new information flow control framework implemented in JavaScript that is designed to track taint in web browsers and notify developers when sensitive data is tranmitted over an insecure protocol. We have shown that the design is both correct in a specific scope and also very easy to use, which satisfies all goals we had in mind for the project. There is also plenty of room for growth, and we look forward to expanding its feature set in the future.

References

[1] Adam Barth <abarth@eecs.berkeley.edu>. 2011. HTTP State Management Mechanism. https://tools.ietf.org/html/rfc6265#section-4.1.2.5

[2] Lujo Bauer, Shaoying Cai, Limin Jia, Timothy Passaro, Michael Stroucken, and Yuan Tian. 2015. Run-time Monitoring and Formal Analysis of Information Flows in Chromium. In *Proceedings 2015 Network and Distributed System Security Symposium*. Internet Society, San Diego, CA. https://doi.org/10.14722/ndss.2015.23295

[3] CheatSheets Series Team. 2020. Transport Layer Protection - OWASP Cheat Sheet Series. https://cheatsheetseries.owasp.org/cheatsheets/Transport_Layer_Protection_Cheat_Sheet.html#Rule_-_Use_TLS_for_All_Login_Pages_and_All_Authenticated_Pages

[4] Quan Chen and Alexandros Kapravelos. 2018. Mystique: Uncovering Information Leakage from Browser Extensions. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, Toronto Canada, 1687–1700. https://doi.org/10.1145/3243734.3243823

[5] MDN Contributors. 2020. Using HTTP cookies. https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies

[6] Electronic Frontier Foundation. 2011. HTTPS Everywhere. https://www.eff.org/https-everywhere

[7] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. 2010. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)*. USENIX Association, USA, 393–407. event-place: Vancouver, BC, Canada.

[8] Troy Hunt. 2013. Your login form posts to HTTPS, but you blew it when you loaded it over HTTP. https://www.troyhunt.com/your-login-form-posts-to-https-but-you/

[9] InfoTech News. 2019. HTTPS encryption traffic on the Internet has exceeded 90%. https://meterpreter.org/https-encryption-traffic/

[10] Christoph Kerschbaumer, Julian Gaibler, Arthur Edelstein, and Thyla van der Merwe. 2020. Firefox 83 introduces HTTPS-Only Mode. https://blog.mozilla.org/security/2020/11/17/firefox-83-introduces-https-only-mode

[11] MDN Contributors. 2020. Sending form data. https://developer.mozilla.org/en-US/docs/Learn/Forms/Sending_and_retrieving_form_data

[12] MDN Contributors. 2020. Strict-Transport-Security. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Strict-Transport-Security

[13] Aaron Miller and Paramjit Singh Sandhu. 2010. TaintSNIFFER: A Robust Dynamic Taint Tracking System For a Homogenous Web Browsing Environment. (2010), 15.

[14] muelli. 2016. Taint Tracking for Chromium. https://blogs.gnome.org/muelli/2016/03/taint-tracking-for-chromium/

[15] W3Techs. [n.d.]. Usage Statistics of HTTP Strict Transport Security for Websites, December 2020. https://w3techs.com/technologies/details/ce-hsts