Università
Ca'Foscari
Venezia

Master's Degree

in Computer Science

Current Deployment and Correct Configuration

Final Thesis

The State of Strict Transport Security

Current Deployment and Correct Configuration

Supervisor
Ch. Prof. Stefano Calzavara

Graduand
Marco Busato
Matricolation number
852074

Academic Year
2020 / 2021

Abstract

Nowadays, the topic of security has become a popular issue due to the contemporary digital society in constant development. Thus, HTTPS only is not enough in order to ensure a high security level (e.g. feeling protected while surfing the net).
Therefore, a new mechanism has increased its employment: the Strict Transport Security, which enhances the security between a user agent and a server. In this thesis, it will be defined the best application of this system by observing the common settings over the internet and the reason why it is preferable declaring some directives and carry out its current deployment among the sites. Furthermore, the sites considered were tested in various scenarios and their security level was assessed in a report that was drawn up.
This study, using some tools such as Puppeteer, wants to inform users about the correct configuration and gives an overview about the current deployment of this essential security mechanism.

Keyword: strict transport security, web security, hsts

Acknowledgement

To my family and friends that supported me through this moment,
and still do everyday.

Contents

Chapter 1

Introduction

Nowadays, in a digitalised society where everything is connected, the importance of feeling safe on the internet has a lot of meaning in our daily life. In this sense, the HTTP protocol does not give the necessary security in avoiding attacks from malicious attackers.
Thus, it has been necessary to implement a mechanism to encrypt communications over the internet in order to avoid plain communication and possible attack by malicious users. At the beginning the SSL (secure socket layer) protocol was used to encrypt connections between client and server. Once it became deprecated in 2015, TLS (Transport Layer Security)[18] took its place. However, this type of protocol does not fully protect websites from "man-in-the-middle" attacks such as protocol downgrade or cookie hijacking.
In 2009 there was a proposal from Jeff Hodges from PayPal, Collin Jackson, and Adam Barth of a mechanism that prevents "Man-In-The-Middle" (MITM) attacks and allows to improve the security over the internet. This proposal became an IETF standard in late 2012 regulated by RFC 6797[9]. This proposal is called **Strict Transport Security**.
We choose to investigate on this security header and assess its current deployment over the net.

In chapter 2, we give some brief introduction on the technologies used on the web and how they evolved during the years. The HTTP which is the foundation of the World Wide Web for exchanging hypermedia files. This protocol is based on request-reply communication in the client-server computing model. The client, which is typically a web browser, submits a request to the server.

The server returns a response message to the client.The HTTP protocol is insecure, since there are no security mechanisms to encrypt the communication between client and server. The protocol for securing web traffic most widely used is HTTPS, which includes layering HTTP traffic over TLS/SSL [18] protected transport protocols in order to ascertain confidentiality and integrity.
The TLS [18] is cryptographic protocol designed to ensure a secure communication over the network. The protocol prevents tampering, falsification and interception of data by encrypting the data flow between client and server.
In order to encrypt the data flow, TLS bases its cipher suite on "Diffie-Hellman key exchange" (DHE) which is an encryption algorithm. This method is used for exchanging keys securely over an insecure channel.

In chapter 3, we talk about the Strict Transport Security which is a security policy that protects a server/client interaction by preventing any protocol downgrade attack and cookie hijacking. Strict transport security (HSTS) teaches browsers that particular domains must only be accessed through HTTPS. It is the main countermeasure to HTTPS stripping. Sites may specify this policy dynamically by HTTP header, or browsers can preload it for ordinary domains. This information is conveyed in the HTTPS web response header field named *Strict-Transport-Security* and includes a *max-age* mandatory directive and other two optional directive: *includeSubDomains* and *preload*.
Moreover, we talk about the potential inconsistencies and limitation of the HSTS policy. These are: (1) neglecting the header or (2) setting the *max-age* value to a negative number. A possible limitation is the so-called Network Time Attack. This type of attack makes less effective the directive on those users that rely on network protocol (such as Network Time Protocol, NTP). The strict transport security was declared a internet standard by the Internet Engineering Task Force (IETF), which is an organization that promotes standard for internet, in the late 2012 and regulated by the Request For Comment (RFC) number 6797 [9].

In chapter 4, we describe our automated application with NodeJs [16] and Puppeteer [2]. Puppeteer Puppeteer is a tool that crawls the web pages in order to retrieve the responses header to check whether HSTS is set correctly or not. On the other hand NodeJs makes possible to run the code outside the context of a web browser. We take the sites for our tests on Tranco[14]

list. The application that we built up collect as much headers as possible from domains and subdomains in order to check the fields contained in these headers.
We also rely on some other mechanism such as content security policy, which prevent code injection attacks in the trusted web page and cookie, which is a piece of data stored in the user computer used for caching stateful informations.
Then, we define a series of layers in which we place the crawled sites and relatives subdomains in order to classify their vulnerabilities and potential entries for an attacker. At each tier corresponds a possible scenario that an attacker may encounter in order to exploit the site's vulnerabilities.

In chapter 5, we illustrate the experiments conducted with the tool and the results that came out. In particular, we replicated the experiments taken from the articles on which this thesis is based [8] [13]. Thus, we crawl the top n-sites of Tranco list [14] in order to carry out the state of HSTS over the net. Then, we collect as much headers as possible from insecure and secure version of the sites present in the list and for the latter one we crawled its subdomains for a correct inclusion in the tables and starting from the headers previously collected, we divided the sites in tiers. Finally, we show some significant common HSTS header setting and the mistakes that we encounter during our tests.

In chapter 6, we describe the two papers that were taken in consideration as a starting point for this thesis. They briefly review the use of HSTS among sites, and thus are useful in order to better understand the topic at stake. The first paper is "The State of HSTS Deployment"; it concerned about the study conducted by Lucas Garron, Andrew Bortz and Dan Boneh, in 2013. They make a scan of the top visited sites of the Alexa top 1 million list in order to understand the correct set of the HSTS header and reveal the common pitfalls. In particular, this case study is the first analysis on HSTS. The second paper taken into account, "Upgrading HTTPS in Mid-Air", introduces an in-depth study of HSTS and public key pinning.

In chapter 7, we will discuss the limitations and possible solutions for these. A limitation of STS is the dimension of the preload list. It is reasonable to think that every site in the net would not be part of the preload list and consequently any conversation among the user and the site would not start

securely. Moreover, the most common mistakes that we encountered are from sites' subdomains: a clerical mistake is that there is confusion in the declaration of HSTS.
A possible further work would be thinking about how to implement a mechanism that helps HSTS in order to avoid that every first communication to a site begins over plain connections. To make the configuration easier to use, it would be useful to establish a standard configuration for declaring the strict transport security header with the possibility to be part of the preload list, or alternatively force all server to respond over a secure channel.

Chapter 2

Background

The most used protocol for securing web traffic is HTTPS, which consists of layering HTTP traffic over protected transport protocols, like TLS and SSL [18], in order to ascertain confidentiality and integrity. Though TLS has some minor cryptographic vulnerabilities, the most critical issue has been HTTPS implementation that has been inconsistent and incomplete. In stripping attacks, network attackers try to downgrade the connection of a victim to untrustworthy HTTP notwithstanding support for HTTPS to both the server and the client. These attacks are possible by browsers that flawlessly support a combination of HTTP and HTTPS connection.

The **Strict Transport Security** (often called HTTP Strict Transport Security, shorten HSTS or STS) is a security policy that protects a server/client interaction by preventing any protocol downgrade attack and cookie hijacking [23] [3].
HSTS is conveyed in the HTTPS web response header field named *Strict-Transport-Security*. In the header field it is also specified a period of time in which the policy should be considered valid for a particular user agent. This directive is called *max-age*, and it must be set to a positive value in order to make the policy effective.
Websites that receive HSTS header over HTTP automatically redirect the connection over HTTPS (even if the specification does not require it) or they can reject the connection.
According to RFC [9], if for any reason a user contacts a web server through HTTP, the server responds with a 30x code (usually 301 code, "moved permanently") and the connection is upgraded to HTTPS. However, attackers

can exploit the vulnerability of the first insecure connection to implement a stripping attack.

In this chapter, we give some background informations in order to better understand how internet technologies work. We further explain how this technologies are related with Strict Transport Security.

2.1 HTTP and HTTPS

The **HyperText Transfer Protocol**, also known as **HTTP**, is the foundation of the World Wide Web for exchanging hypermedia files [7].
This protocol is based on request-reply communication in the client-server computing model. The client, which is typically a web browser, submits a request to the server. The server returns a response message to the client.
The HTTP protocol is insecure, since there are no security mechanisms to encrypt the communication between client and server. Thus, the conversation is visible by any users with malicious intent.

The secure version of HTTP, i.e., the HTTPS, relies on the TLS [18] protocol, or formerly SSL, which encrypts the communication between client and server. The motivations that led to choose the adoption of a secure version are the authentication on the web and the protection of the data while in transit.
The HTTPS establishes a secure channel over an insecure network, ensuring a significant level of security against eavesdroppers, those who secretly listen to private connections and "man-in-the-middle" attacks, in which a malicious user is positioned between the user agent and the server and can hijacking the communication.

2.2 Transport Layer Security

The TLS (Transport Layer Security) [18], and its predecessor SSL (Secure Socket Layer), are cryptographic protocols designed to ensure a secure communication over a network. The TLS protocol prevents tampering, which means the intentional modification of informations in order to make them

harmful to the user, falsification and interception of data by encrypting the data flow between client and server. In 2014, all block ciphers present in SSL 3.0 were found to be vulnerable to the POODLE attack [15]. The acronym stands for "Padding Oracle On Downgraded Legacy Encryption". This type of attack takes advantage of the encryption process, which is based on padding that expands the message in order to be compatible with low-level cryptographic algorithms. The attackers take advantages of this vulnerabilities in order to break the cypher and decrypt the connection. The SSL protocol became deprecated in 2015.

TLS authentication is unilateral, which means that only the web-server needs to be trusted to the client. The client validates the server's certificate by checking that the digital signature is acknowledged by a certificate authority (CA) using a public key cipher. The protocol operation can be resumed in:

1. The handshake, which is the first step to begin a communication, starts when the client presents a list of supported ciphers, then the server picks one from the list and communicates to the client the chosen cipher

2. Client and server exchange their keys, usually using "Diffie-Hellman Key Exchange" (DHE) which is useful to generate a unique session key for encryption and decryption of the messages

3. The handshake operation terminates and begins the messages exchanging, that is encrypted and decrypted with the session key previously generated by DHE.

If any of the above steps fails, the connection is not created.

However, the HTTP over TLS alone is not enough against malicious intents. A common scenario in which a secure connection is established is the following:

- A user sends a request to a web server via HTTP;
- The web server responds with a HTTPS redirect;
- The user sends a secure request and the secure communication begins.

In 2009, an elaborate type of attack against HTTPS called SSL stripping was presented. This attack takes its potential from user trust, which means

that the user is certain that the communication starts securely. The attacker places itself in the HTTPS redirection and intercepts the requests form client to server. Then it acts like a bridge between the server, maintaining a secure channel, and the client with an HTTP connection.

In order to avoid the so-called SSL stripping attack, the best practice is enabling the strict transport security, which will be further explained in the next chapter.

2.3 TLS Cryptography Protocol

In this section we give a briefly introduction on the "Diffie-Hellman key exchange" (DHE) which is one of the most common encryption algorithm used in the TLS protocol [18]. This algorithm is used for exchanging keys securely over an insecure channel.
This encryption algorithm involves two or more participants and assigns them a pair of keys, respectively called **public** and **private**.
The **public** key can be obtained by anyone and is used to decrypts the encrypted received message(s).
The **private** key is kept confidential to the client and is used to encrypt the message that the client wants to send over the communication channel.
Before we explain more in detail the algorithm, we give two technical definitions:

- A number is called **prime** if and only if it is divisible only by one and itself. Examples of prime number are 2, 3, 5, 7, 11;
- A number in relation with another number is called **coprime** if and only if the only two dividers in common are 1 and -1. Example of coprime number are 6 and 35, but 6 and 27 are not coprime since they have in common the number 3.

We have two entities, A and B, in which they publicly agree on a prime number p and a coprime number g between 1 and p-1. Then, they secretly choose a random number in which:

- A sends to B $g^a \bmod p = \text{x}$;
- B sends to A $g^b \bmod p = \text{y}$.

These computed values, x and y, are sent over the communication channel. At receiving time, in order to compute the secret s:

- A computes: s=y^a mod p;
- B computes: s=x^b mod p.

Now, A and B know and share the secret s which is the same value for both parties. The value s can be used to encrypt any message sent across the same open communications channel.
The strength of this algorithm comes from the fact that, if any intruder E listens to the communication between A and B and knows p, g, x, and y, will take extremely long times to compute the secret s.

The Diffie-Hellman algorithm is used because , to generate a new key pair at each session is fast and discards this pair at the end of the session. This particular characteristic is called *forward secrecy* and is one of the protocols used in the TLS [18] cipher suits.

Chapter 3

Security Best Practices For HSTS

The first time that a web site is accessed and it receives the strict transport security header, the browser records this information and knows that any further access to that site must be only over secure connection, in other words, using an HTTPS connection.
It is important to highlight the fact that if a browser receives an HSTS header over an insecure connection, it will be ignored. This is due to the fact that a malicious user can intercept the connection and inject or modify the header (Man-In-The-Middle attack). If the connection starts over a secure channel the directive is compliant.

When the expiration time is elapsed, the next attempt to connect the site via HTTP proceeds as usual and the client contacts the server through insecure channel.
In order to avoid the expiration of time specified in the *max-age* directive, every access to the web server automatically refreshes the information. If the *max-age* is set to 0, the strict transport security header immediately expires, allowing access via HTTP.

There are other two additional directives that can enforce HSTS policy:

- ***includeSubDomain***, if set, specifies that HSTS directive should be applied to all site's subdomain(s) as well.
  For example, if *example.com* sets a valid HSTS policy with *includeSub-*

Domains, then all further connection to *example.com* as well as *foo.example.com* and *bar.foo.example.com* must be contacted over HTTPS only. This directive covers subdomains of the entire site regardless of depth;

- ***preload***, it can be set after submitting the site via the from provided by Google [19] and if the directive is present, it enforces HSTS policy by including the domain in a list maintained by Google. By setting this policy, the web browser is notified that the host should only be contacted via TLS/SSL connection in order to avoid every first attempt to create a plain connection. The major browsers have their own preloaded list based on Chrome list.

It is important to recall that:

- the directives cited before must be declared once in the header field;
- the unrecognized directive must be ignored by the user agent, and proceed the navigation with the recognised directives;
- the order of appearance is not significant. Declaring first *includeSubDomains* or *max-age* is not important;
- the directives are case insensitive, which means that directives declared uppercase have the same effectiveness of those declared lowercase or with capital letters.

However, the *preload* directive is not part of HSTS specification, thus it is not secure to enable it by default. Moreover, to join the preloaded list, the site must be submitted via an online form[19].
In order to join the list, the site is submitted it must satisfy the following requirements:

- Provide a valid certificate;
- If listening on port 80, there must be a redirect from HTTP to HTTPS;
- Serve all subdomains over HTTPS. In particular, the site must support HTTPS for the www subdomain if a DNS record for that subdomain exists;
- Serve an HSTS header on the base domain for HTTPS requests:

- The *max-age* must be at least 31.536.000 seconds (one non leap year);
- The *includeSubDomains* directive must be specified;
- The *preload* directive must be specified.

In the next section we give more details about *preload* directive.

3.1 HSTS Preload

Even if it is not part of the directive mentioned in the RFC 6797 [9], aiming to get on the preloaded list should be achieved by all sites. The list is maintained by Google and used by Chrome and most of browsers base their own preloaded list on this. We also mentioned, in the previous section, how it can be done to be part of the list.
However, including the *preload* directive in the header is not recommended by default. But, what does being preloaded mean?
The concept of being preloaded regards the fact that, the browsers are already informed about the requirement of using a secure connection by the host before the communication begins. To be preloaded removes any malicious intent from an attacker to intercept and tamper any communications that are redirected from HTTP. The submission via registration form[19] has permanent consequences both on the site and on the preload list.
Domains can be unsubscribed from the list, but the operation takes months for a change to reach users with a Chrome update and cannot make any guarantees about other browsers. Also for the removal there are some precautions, such as:

- Be preloaded or having a pending preload;
- Serve valid certificate;
- HSTS header must be set correctly. The *preload* directive must not be included.

3.1.1 Preload List Management

As we explained previously, a public list maintained by Google exists and there can be found information about sites that have included the *preload*

directive inside HSTS header.
The Google preload list is stored in a json format, and includes:

- Name field, which is the site name, e.g, site.com/;
- Policy field, which is use for list maintenance and indicated under which condition the site is included in the list;
- other field are optional. The fields that occur most are *mode*, usually set at "force-https" and *include-subdomain* which is a boolean (true/false).

Here is an example:

```
{ "name": "***.com", "policy": "custom", "mode": "force-https" }
```

All of these elements of the entry together, ends up meaning that for the site specified in the name field that have a custom policy it is applied a force-https mode (update an insecure connection to a secure one).
For sites that do not specify the *includeSubDomain* or do have HPKP[1]/Expect-CT the policy is "custom". That means that, when the site is included in the list, it had a HPKP/Expect-CT field in the header response. The Http Public Key Pinning (HPKP) is a now deprecated security mechanism that allows HTTPS websites to resist against impersonators of misused certificates.

Furthermore, also Mozilla implements its own preload list. Even if it is based on chrome preload list, it implements a script that checks periodically the site's status. Every day the script attempts to connect to each site of the list and, if the site responds with a HSTS header with at least a *max-age* directive of 18 weeks expressed in seconds, the entry for that site is updated, otherwise the host is not included.
The host is not removed by the script in case of bad connection, which means that the script will not remove a host from the list whether there happens to be a host disconnection. For site removal the host must not sent an HSTS header or the directive has to be less than 18 weeks in seconds.

3.2 Browser Support

Since HSTS became a IETF standard, the most common browsers have taken actions in order to support this security policy. Therefore, nowadays, HSTS

[1]Deprecated

is supported by all modern browsers.

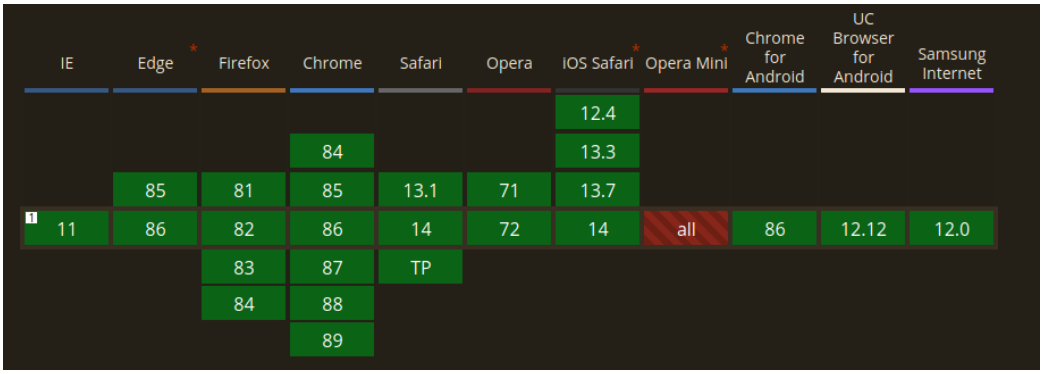

Figure 3.1: HSTS, browsers support[6]

In figure 3.1, the number inside the colored box states the version of the site that supports HSTS. The green box indicates that the version declared inside supports HSTS. The version of the browser that does not support HSTS is indicated by a red box. Unfortunately all versions of Opera Mini browser does not supports HSTS.
The next figure shows the use of the browser.

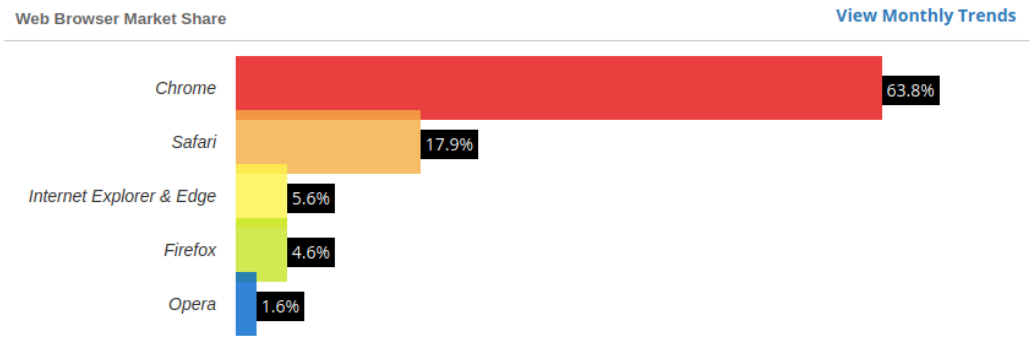

Figure 3.2: Browser use (October 2020 [20])

3.3 Best Practices

There are some instructions to follow in order to set up an HSTS header correctly. These best practices can be resumed as follows:

- Hosts should declare HSTS policy at their top level domain name. In other words, the policy should be enabled for https://example.com;
- Have the *max-age* directive set up to 1 non leap year in seconds (31.536.000 seconds);
- *includeSubDomains* should be present.

The HSTS policy must never be send over plain HTTP.

Moreover, other than the best practices listed above, a subdomain should comprehend a request from its origin domain in order to ensure that HSTS is set for the origin itself and make sure that the user is protected from possible cookie injection attack that may inject a reference to the parent domain or to a non-occurring subdomain, to which the attacker at stake would respond.

3.4 Inconsistencies and Limitations

Two potential inconsistencies in HSTS deployment exist. These have dangerous security implications which are: (1) neglecting the header or (2) setting the max-age value to a negative number. In the first scenario, it is dangerous due to the fact that it may imply that sometimes HSTS is not operative, mainly when no HSTS-protected page (on the same host) or a parent domain (i.e. includeSubdomains) was visited. Worse is setting a max-age value to a negative value, because it disables HSTS for that host. It is important to state that the browser implements this so that a possible entry for the host is dropped, rather than stoking the directive. Therefore, if the origin domain sets includeSubdomains while the host itself is setting a max-age to zero, HTTP connection to that host are permitted until the origin is once again visited, thus implementing HTTPS for all its branches.
These inconsistencies can be exploited by an attacker to build up an attack on the site. Moreover, in certain cases (i.e. when the communication is insecure) an attacker can phishing (i.e. impersonating some trustworthy entity)

or sniff (i.e. watch the flow communication between entity A and B) cookies even they are flagged *Secure*.

According to the RFC [9] specification, there are some vulnerabilities that HSTS presents. A possible vulnerability is the **Bootstrap MITM** (Man-In-The-Middle).
This type of vulnerability shows up when the user manually enters, or clicks on a link to an unknown host that uses an HTTP connection instead of HTTPS. Because of this, the user agent is subjected to various attacks, such as cookies hijacking.
Another limitation that HSTS presents is the so-called **Network Time Attack**. This type of attack makes less effective the directive on those users that rely on network protocol (such as Network Time Protocol, NTP). Briefly the NTP is a protocol that synchronize the computers clock connected on the internet. If the attacker can operate on such protocol, he/she can modify HSTS directive in order to make it less effective or even totally ineffective.
We invite the reader to take a look at the RFC 6797 [9] in section 14.

Chapter 4

Tool

In order to assess the current deployment and establish the correct configuration of HSTS, we identify three major tests to perform on sites:

1. Giving the state of HSTS deployment in the wild by crawling the first thousand hundred sites. That is, how many sites responds over secure channel and present HSTS field in the response header. Then, collect the directives that the header carries with itself;

2. Crawling the sites in order to give a more specific overview of HSTS security mechanism. In particular, determine which sites follow the best practices (section 3.3) and which of these present inconsistencies and limitations on its deployment;

3. By analysing the level of HSTS deployment, we defined a series of tiers in which we collocate the sites after subjecting them in various scenarios, in order to evaluate their security level.

To figure out how HSTS is implemented in practice, we built up an automated application with NodeJs[16] and Puppeteer[2].
Puppeteer is a tool that crawls the web pages in order to retrieve the responses header to check whether HSTS is set correctly or not. NodeJs makes possible to run the code outside the context of a web browser.
We take the sites for our tests on Tranco[14] list. This list take the advantages from other type of lists like Alexa[11], Cisco Umbrella[10], Majestic[12] and Quantcast[17] and it collects the data from these lists and applies the Dowdall rule in which at each candidate site is given a score from 1 point, to

the first candidate, to 1/N, to the last candidate.

The application that we built up with Puppeteer [2], collects as much headers as possible in order to check the fields contained in these headers. We start by collecting the headers from parent domain (i.e example.com, where "example.com" is replaced by the name present in the list). Then, once the home page responds, we collect the relative subdomains and headers of these subdomains by retrieving all sites that belong to the domain referred from "href" html tags present in the page. An example of subdomain is *www.example.com* or *a.example.com*.
Once these headers are collected, we focus on the following fields:

- **Status**, which tells us if there are a redirect (i.e. code 30x) in addition to the location field or if the request has a successful response (i.e. code 200);
- **Location**, which tells if where we have to looking for the redirection;
- **Strict-Transport-Security**, which tells if there are or not the presence of the policy that we are going to state its current deployment;
- **Content-Security-Policy**, in particular the directive *upgrade-insecure-request* if present, to check if there are the possibility to have mixed-content in the page, we explain this in section 4.1.1;
- **Set-Cookie**, to check the presence of the flag *Secure* and if the sites use the so called cookies prefixes, more details are given in section 4.1.2.

Moreover, Puppeteer [2] provides some function to extract some other feature from the response header puppeteer object like the security details so as to check if provides a valid certificate or uses the last version of TLS[18].

4.1 Related Mechanisms

In order to optimise the inclusion of sites in the various scenarios proposed, we also rely on the headers **Content-Security-Policy** and **Set-Cookie**. In particular we focused on the directives *upgrade-insecure-request* and cookie setting's variant called "cookie prefixes".

4.1.1 Content Security Policy

The content security policy is a standard introduced to prevent code injection attacks resulting from execution of malicious content in the trusted web page [21] [5].
Nowadays, Content Security Policy (CSP) has numerous directives to restrict framing, content inclusion or even enforce TLS; as a matter of fact we focus on the *upgrade-insecure-request* in this chapter.
The *upgrade-insecure-request* directive of the content security policy, instructs the user agent to treat all the insecure URLs of the site as if they were secure. This replace, in part, and act like the deprecate directive *block-all-mixed-content*, that is, as the name suggest, a mechanism to instruct the browser to block any insecure reference inside the page. Every reference to insecure URLs will be rewritten before the request is made.

The *upgrade-insecure-request* of CSP instructs the tool implemented to check if a site allows the presence of mixed content.
An HTTPS page that includes context that references HTTP pages, is called mixed-content page. These types of pages are partially encrypted and thus open to attacks by malicious intent. The mixed content is divided in two categories:

- Passive mixed content: A mixed-content page contains reference that cannot alter other portion of the page. The reference can be an image, in which an attacker can masquerade malicious content in such image;

- Active mixed content: A mixed-content page contains reference that can alter all or large portion of the page. This reference can be a script or a link, in which an attacker can intercept the request to the reference, rewrite the content and inject malicious code in the page in order to steal sensitive data.

Without a valid HSTS header, the *upgrade-insecure-request* directive will not ensure that visiting your site through third-party sites will be upgrade to HTTPS. This because the *upgrade-insecure-request* will not replace HSTS header and leaving the sites opened to the SSL stripping attack. Thus, this can be a potential vulnerability to exploit by an attacker.

4.1.2 Cookies

A cookie is a piece of data stored in the user computer used for caching stateful informations . They can also be used to remember informations previously entered into form fields, such as: names, addresses, passwords and payment card numbers [22] [4].
Cookies use *Set-cookie* HTTP header which is sent as a response from the web server. Moreover this header is sent again back to the server in the future request. Given that the HTTP protocol is stateless, using cookies is helpful in order to remember to users the information in the state. Notwithstanding some mechanism useful to send them in a secure way, throughout HSTS inconsistency these cookies can be leaked even though they are flagged as *Secure*. With the stated Secure attribute, the cookies are restricted to encrypted connection. Nevertheless, if the communication occurs in a non-secure connection and the cookies are retained Secure, these cookies themselves can be waylaid by a man-in-the-middle attack.

The cookies prefixes aim to mitigate some drawbacks of the cookie implementations. This type of implementation helps to set cookies "Secure" and locked to a domain. An advantage of cookies prefixes is that they are related only to the secure origin, in other words, if a non secure origin tries to set them they will be rejected.
This mechanism helps us to check if there might be a theft of cookie domain in case of lack of the *includeSubDomains* directive.
There can be two version of cookies prefixes:

- If the cookie name starts with "__Host-" ignore all cookie unless if it not presents any "Domain" directive, the cookie path is equals to "/" and is flagged "Secure";
- If the cookie name starts with "__Secure-" reject it unless it is flagged "Secure" and the request URI is considered "secure" by the user agent.

If the cookie does not match any of these cases it will be rejected.

4.2 Tiers

In this section we define a series of layers in which we place the crawled sites and relatives subdomains in order to classify their vulnerabilities and poten-

tial entries for an attacker.
Each tier corresponds to a possible scenario that an attacker may encounter in order to exploit the site's vulnerabilities. Therefore, in order to make this type of ranking at each scenario more readable, it is assigned an alphabetic letter, from "E", the most insecure, to "A", that is close to be very secure.
In each tier we evaluate the correct declaration of HSTS both on parent domain and on its subdomains. Moreover, we check the presence of relative mechanisms illustrated in the previous section. In particular, the *upgrade-insecure-request* of **CSP** tells us the presence of mixed content and the *Set-cookie* field tells in case of HSTS deactivate, with a max-age value lower or equal to 0 or the lack of *includeSubDomains* directive, if there are cookies flagged "Secure".

Before we illustrate the various tier of our study, we would like to highlight some assumptions.
In order to make our choice consistent, there would be a "trust on first use", in other words there is not any interference from any attacker on every first connection to the site. The other assumption regards the fact that the navigation of the site begins on the landing page (i.e, the home page) of the site.

- Tier A: The site complies to all requested HSTS features. It implements all HSTS best practice and moreover is preloaded.
  The site placed in this tier is not subjected to any assumptions, since it can mitigate a passive and active attack, such as a man in the middle data manipulating attacks. The navigation starts and ends over a secure channel;

- Tier B: The site complies HSTS best practice. In this scenario a site security can mitigate an attack from any malicious intent. The site is subjected to the assumption of "trust on first use", since the navigation starts over an insecure channel for every first time;

- Tier C: The site response over a secure connection but it has some HSTS bad implementation, like *max-age* less than one year in seconds or there is no *includeSubDomains* directive.
  In this scenario, the site can mitigate an attack on the home page, but could be vulnerable on its subdomains.
  In this type of scenario we also check the presence of "cookies prefixes"

because in case of missing *includeSubDomains* directive there can be a potential cookies leakage. The site is subjected to both assumption previously mentioned.

- Tier D: In this scenario a site responds over an insecure channel and there is no presence of HSTS header field. Hence, a site is vulnerable to both active and passive attacker.

There are other tier in which we insert multiple scenarios. This is because the sites that enter in those scenarios are very vulnerable to any type of attack either active or passive.

- Tier E: In this tier we need not to do any assumption since the entire domain is subject to any attack. We put those site that have either *max-age* directive less than or equal to zero or do not have HSTS header.
  - **Scenario 1:** The site responds over a secure connection and has no HSTS header. In this scenario a site is always vulnerable on every first connection since the connection begins over HTTP and HSTS is not presents. Thus, the successive connection cannot start over a secure channel;
  - **Scenario 2:** The site responds over a secure connection and has a malformed HSTS header. With malformed header we intend an header with multiple directive declaration or unacceptable values like "*[...]; includeSubDomains = true*";
  - **Scenario 3:** The site responds over a insecure connection and has HSTS. This is probably the worst case because in case that HSTS header is not ignored by the browser when the site is accessed using HTTP, an attacker may intercept HTTP connections and inject, modify or remove the header.

Chapter 5

Web Measurement

In this chapter we illustrate the experiments conducted with the tool and the results that came out. In particular, we replicated the experiments taken from the articles [8] [13] on which this thesis is based.
We start our analysis starting from:

- http://example.com;
- https://example.com.

where "example.com" is replaced by the current domain present in the list. Thus, we give an overview of HSTS deployment among the sites and the current settings of this security mechanism. Moreover, we collect the number of those site that declare the *preload* directive in the response header. Finally, we divide the sites crawled into the tiers previously described (section 4.2).

5.1 HSTS Settings On The First Thousand Hundred Sites

We crawl the top n-sites of Tranco list [14] in order to carry out the state of HSTS over the net.
We counted the number of sites that respond over a secure channel and collect headers from the home page of these sites. From the headers, we focused on the field *strict-transport-security* and then collect the data of directives. Note that we collect the data of those site that set a *max-age* equals to zero,

that is, those that deliberately deactivated it.

In the table 5.1, we correlate the incidence of sites that respond over HTTPS. In the average, the percentage of the sites is never below the 80%. The study conducted in 2013, registered an incident on the average of about 60%. Thus, we can say that with respect to this study (table 5.3) we have a great improvement.
In the next step of this study, we registered the state of HSTS of the top thousand hundred sites. Once again we have an improvement of the data, mostly in the top ten sites of the list. In this level, the half of the sites set HSTS more or less correctly, none of them deactivate it with the directive max-age equals to zero. Moreover, two out of four declare the *preload* directive in the response header. This is satisfactory in terms of the fact that, in 2013, none of the sites in the top ten had a proper HSTS setup. Notice that, when we approach on the first thousand sites, the percentage of settings decreases. This is due to the fact that there are sites not so visited, sites that do not collect sensitive data or static sites and therefore, the need for HSTS is not a requirement.

	HTTPS	
# TOP sites	#	%
10	8	80%
100	92	92%
1.000	889	89%
10.000	8.417	84%
100.000	79.393	79%

Table 5.1: Number of sites that responds over secure channel

Finally, we conclude this part of the study with the evident proof that, in these years, the deployment of strict transport security has significantly increased its use. In addition, the introduction and maintenance of a preload list helps the sites developer to aim for a correct settings of this security mechanism.
In the next section we give a more detailed overview of HSTS.

HTTPS		HSTS		max-age = 0		includeSubDomains		preload	
#	%	#	%	#	%	#	%	#	%
8	80%	4	50%	0	0%	3	75%	2	50%
92	92%	48	52%	2	4%	29	60%	14	29%
889	89%	295	33%	9	3%	164	56%	87	29%
8.417	84%	2.057	24%	103	5%	1.156	56%	502	24%
79.393	79%	15.308	19%	824	5%	8.375	55%	2.948	19%

Table 5.2: HSTS settings on the first hundred thousand sites

# top sites	HTTPS	HSTS	incl.sub.	max-age=0
10	8	1	0	1
100	76	3	0	2
1000	629	11	3	4
10000	5.402	56	11	10
100000	46.943	277	66	25

Table 5.3: HSTS deployment in 2013

5.2 Overview of HSTS

In this section, we collect as much headers as possible from insecure and secure version of the sites present in the list and for the latter one we crawled its subdomains for a correct inclusion in the tables.
For the first part, we collect:

- which sites do not redirect to a secure connection;
- which sites do redirect but always to an insecure connection;
- which sites sends HSTS header only through HTTP;
- which sites redirect to HTTPS.

For the sites that responds over secure version of HTTP, we focused on:

- sites that do not have HSTS or deactivated it by setting a *max-age* value to zero;

- sites that have bad implementation of HSTS such as lack of the presence of *includeSubDomains* directive;
- sites that follow the best practices described in chapter 3;
- sites that follow the best practices and declare the *preload* directive in the header.

Let's focus for a moment only on the sites that rely on HTTP. Even if there are some sites that still declare the policy header only over HTTP, we have a comfortable scenario in which the number of insecure sites has decreased over the years.
Moreover, we notice that there are a high percentage of sites that redirect correctly to the HTTPS. This is a comfortable data since there is a secure approach in sites development, however, on the other hand these sites are subjected to the *man-in-the-middle* attack on every first visit from the users since the first contact to the site happens through an insecure channel.

In the second part of the test, we focused on possible HSTS settings scenarios. A point of particular concern is the number of sites that have HSTS not present, deactivated or with a *max-age* directive less than or equal to zero, which is equivalent to deactivate it. This is due to the fact that probably the crawled sites are static, that is, those sites that do not collect user's sensitive data or do not have a private area for the user. In this cases the only HTTPS is sufficient.

A particular assumption is that we cannot do a strict comparison between our study and the study conducted in 2015, reported in the table 5.5 because of the number of headers collected, the difference between the lists used and the type of study conducted. In conclusion, observing the fields of sites that follow the best practices in the table 5.4 we can state that about 10% of the sites scanned set correctly HSTS but this is not sufficient to assume that the sites are secure to both a passive or active attack.
In the next section, we try to provide an overview of the security of the scanned sites.

	Tranco Top 1M	
	Domains	%
Total sites	**164.747**	**-**
total HTTP site	**86.354**	**52,42%**
Doesn't redirect HTTP->HTTPS	14.069	16,29%
Redirect to HTTP	11.729	13,58%
Set STS only over HTTP	3.189	3,69%
Redirect to HTTPS	57.130	66,16%
total HTTPS site	**78.393**	**47,58%**
HSTS not present or max-age<=0	63.074	80,46%
HSTS with malformed header	7.608	9,70%
HSTS Best Practice	4.752	6,06%
HSTS Best Practice + Preload	2.948	3,76%

Table 5.4: Overview of HSTS

	Alexa top 1M	
	Domains	%
Attempts to set dynamic HSTS	12,593	-
Doesn't redirect HTTP ->HTTPS	85,554	44.1%
Sets HSTS header only via HTTP	517	4.1%
Redirects to HTTP domain	774	6.1%
HSTS Redirects to non-HSTS	74	0.6%
Malformed HSTS header	322	2.6%
max-age = 0	665	5.3%
0 <max-age <= 1 day	2,213	17.6%
Sets HSTS securely w/o errors	5,099	40.3%

Table 5.5: HSTS deployment in 2015

5.3 Are sites secure?

In this section, starting from the previously headers collected, we divided the sites in tiers. The tiers which we refer to are described in section 4.2,
A worrying fact is that, in each subdivision, the tier E is populated by about

50% of the sites. We remind that we do not make any distinction from dynamic and static site and furthermore, the latter falls into this scenario. However, this is not a valid excuse to those sites that respond over an insecure channel, do not redirect to a secure version of the site or set HSTS only through HTTP.
Viceversa, tier A is the less populated with respect to the lower tiers. A possible explanation is that even if the home page of the site sets correctly HSTS, probably its subdomains do not require a reference to the parent and due to this the policy is not enforced. Thus these sites that have the requirements to enter in the higher tiers, unfortunately, are placed to the lower tiers.
Finally, we observe that the tier B is the least populated. This can be in part to the fact previously described for tier A and in part that the subdomains collected do not present the policy in the response header even if the parent declares in the policy the *includeSubDomains* directive.

	TIER									
	A		B		C		D		E	
top n-sites	#	%	#	%	#	%	#	%	#	%
8	0	0,00%	0	0,00%	3	37,50%	1	12,50%	4	50,00%
96	4	4,17%	1	1,04%	24	25,00%	22	22,92%	45	46,88%
912	26	2,85%	1	0,11%	131	14,36%	159	17,43%	595	65,24%
8.861	107	1,21%	6	0,07%	989	11,16%	1.376	15,53%	6.383	72,03%
90.038	563	0,63%	49	0,05%	7.088	7,87%	19.019	21,12%	63.319	70,32%

Table 5.6: Sites divided in tiers

Overall, we can say that the developers assume that the navigation starts from the home page when actually the user can access the site from any of its subdomains. It is clear that in table 5.6, that the sites protect only their home page and only few sites are fully protected thanks to correct HSTS configuration.

5.4 Common Settings and Clerical Mistakes

In this section, we show some significant common HSTS header setting and the mistakes that we encountered during our tests. In addition, we also

collect the Set-cookie and CSP directives in order to point out possible vulnerabilities from HSTS inconsistencies.
Once again, we collect about thousand hundred headers from parent domains and about five times as many headers from subdomains and count the frequency of these occurrences. For study reason we illustrate in the tables below significant values of the headers collected.

The most encountered value during our test is the "undefined" value, which means that the sites do not present the field in the response header, with a frequency of 62.260 times. Moreover, another fact is the number of those headers that deactivate the policy by setting the *max-age* value equals to 0. The sites that presents only max-age occur 409 times, those which set the *includeSubDomains* occur 98 times and those which declare the *preload* directive occur 113 times. We expect this data since in the previous sections we have an high number of sites that are placed in tier E in table 5.6 (last row) or in table 5.4 were the number of sites with a HSTS deactivated is over the 80%. These data are highlighted in red in the table 5.7.
Moreover, we present an HSTS header value which is borderline, which means that is accepted by the UA, but not compliant the best practice for be included in the preload list.
Finally, the parent domains that present a valid header has a frequency of occurrence of approximately 6.500 times. Thus, we have a consistency of data concerning what did in the previous sections and what we present here in the table 5.7.

The value that recurs the most for *max-age* directive is 31.536.000 seconds (1 year in seconds). This value confirms what described in chapter 6 where in their tests, they recorder this value as the most recurrent. Additionally, we refer this value in order to follow the best practices in chapter 3. We also registered 832 occurrences of value equals to 0. This data can also be seen in table 5.2 (last row) under the *max-age* heading.
An interesting fact is the number of different value for *max age*, in the table 5.8 we present the first 10 occurrences, but with our tool we collect about 170 different values.

Now we focus on the collected subdomains HSTS header.
Also here, we collect with a frequency of 122.659 times the value "undefined" which we remind that means that there are not the field in the response

HSTS header value	# of occurrences
undefined	62.260
max-age=31536000	2.800
max-age=31536000; includeSubDomains; preload	1.683
max-age=31536000; includeSubDomains	1.488
max-age=63072000; includeSubDomains; preload	547
max-age=0	409
max-age=31536000; preload	215
max-age=0; includeSubDomains; preload	113
max-age=0; includeSubDomains	98

Table 5.7: Most occurring values of HSTS header on parent domains

max-age value	# of occurrences
31.536.000	7.216
63.072.000	1.931
15.768.000	1.135
15.552.000	1.049
0	832
300	622
2.592.000	423
7.889.238	416
15.724.800	328
86.400	308

Table 5.8: Common max-age value on parent domains

header. Furthermore, we registered two cases in which the directive is deactivated and in one case defines also the *includeSubDomains* and *preload* directives.
We also recorded two borderline cases with two different *max-age* values which is accepted by the UA, but not compliant with being included in the preload list as we described in chapter 3.
Finally, in the most significant HSTS header presented in the previous tables, we have four cases of correct setting of this policy, all of them with an expiration time of over 1 year in seconds.

For what concerns the *max-age* value here too we registered as the most common value 1 year with a frequency of over 33.500 times. The most significant data is the *max-age* value equal to 0, which is more frequent respect to the parent domains, which means that the subdomains are more vulnerable to an attack.
The other values range from few minutes to about six months in seconds with an occurrence of over thousands times.

HSTS Header value	**# of occurrences**
undefined	122.659
max-age=631138519	18.662
max-age=31536000	12.573
max-age=31536000; includeSubDomains	8.063
max-age=31536000; includeSubDomains; preload	7.989
max-age=15552000; preload	6.133
max-age=0	1.738
max-age=31536000; preload	538
max-age=0; includeSubDomains; preload	186

Table 5.9: Most occurring values of HSTS header on subdomains

max-age value	**# of occurrences**
31.536.000	33.580
631.138.519	18.691
15.552.000	10.517
63.072.000	5.174
15.768.000	3.723
0	2.360
7.889.238	1.363
259.200	1.253
15.724.800	1.209
300	1.123

Table 5.10: Common max-age value on subdomains

In conclusion, we present in the tables only significant values of HSTS setting but, for what concern the parent domains, we registered about 650 different headers and over 170 different values for *max-age* directive. Furthermore for subdomains we collect over thousands different setting for HSTS policy and about 250 different *max-age* values. This means that there are a lot of confusion for declaring the policy at its best.
Moreover, the RFC [9] gives some loose guidelines to declare this policy, and in particular states two guidelines that gives to the user the possibility to declare the same policy in various way:

- The order of appearance of directives is not significant;
- The names of directives are case-insensitive.

This creates a lot of confusion and moreover, the *includeSubdomains* directive is declared in it as a optional field than a required field leaving the subdomains open to attacks from malicious users.
What we expected is that RFC [9] gives some stakes to declaring the policy with the possibility to add or not other directives without compromising its effectiveness.

5.4.1 Cookie leakage and CSP setting

To better understand the security implications of inconsistent HSTS deployment, we performed an analysis to estimate the number of cookies which can be exposed in the clear against network attackers. Thus, we focus on cookies, that rely on HSTS to protect their cookies rather than setting the Secure attribute. Specifically, we replicated the tests performed in "Reining in the web's inconsistencies with site policy" [1] and identify three categories of cookies at danger on sites that have HSTS enabled:

1. Cookies set by a page that correctly deploys HSTS, but another page on the same origin (i.e. example.com/foo) has configured *max-age* to a non-positive value. The attacker can retrieve these cookies after disabling HSTS;
2. Cookies bound to domains such that *includeSubDomain* is not declared in HSTS response header, nor at least one of its parents. The attacker can steal these cookies by forcing HTTP requests to a non-existing subdomain (of the set "Domain" value);

3. Cookies bound to domains such that any of their subdomains deactivates HSTS by setting *max-age* directive to a non-positive value. After HSTS deactivation, these cookies can be leaked by forcing the victim to visit the now downgraded HTTP subdomain.

For the first case, we find out that 2.084 cookies out of 5.552 collected can be retrieved by deactivating HSTS in the same domain.
Then, for the second category 12 cookies bounded to those sites that do not extends HSTS to their subdomains, can be leaked through referring to a non-existing subdomain.
Finally, for the last case, 7.064 collected cookie are vulnerable because an attacker can contact the site over the now downgrade HTTP protocol.
Overall, sites risk exposing cookies in the clear due to inconsistencies in their HSTS configuration. It is clear that the cookie leakage comes from the subdomains, highlighting the fact that those are left unprotected.

For what concern *upgrade-insecure-request* of CSP almost the totality of the collected headers set this directive both on domains and relative subdomains that have HSTS disabled. Therefore, the sites rely on this directive rather than choose HSTS that protect their site from third party visits.

Summarizing what we did in our study, in the tables 5.1 and 5.2 we have a significant increment on sites that respond over HTTPS and in declaring of HSTS. Thus, we can imagine that alike the security on those sites increase as well. Unfortunately, table 5.4 shows that a low number of sites declare HSTS without errors. A particular observation is that we cannot do a strict comparison between our table 5.4 and the table 5.5 of the study conducted in 2015. This observation arise from the fact that the number of headers collected, the difference between the lists (Tranco list versus Alexa list) used and the type of study conducted are different. So, we cannot infer nothing about HSTS configuration over HTTPS, but what can we say is that there is a drastic decrease on non-redirecting sites, 16,29% registered on our study and 44,1% on the study of 2015, and on those site that declare HSTS only over HTTP, 3,69% against 4.1% in 2015. This might means that sites developer pay much more attention setting up HSTS policy.
Although this data can be comfortable, in the table 5.6 we show that the sites still have to face with some HSTS inconsistencies, in particular, with policy declaration on subdomains. Given the low number of sites in the first

two tiers, it is clear that the policy declaration presents some inconsistencies and leave the sites open to attacks that HSTS is meant to mitigate.
Finally, in the section 5.4, we collect the headers and define the common mistakes encountered during the tests. The last four tables [5.7, 5.8, 5.9, 5.10] indicates the significant values for HSTS declaration and the common values for *max-age* bot on parent domains and their relatives subdomains.
The data are inline on what did in the previous tests. Thus, we have that the common HSTS value is "undefined" which means that the policy is not declared. Other significant mistakes came from the subdomains crawled. The number of inconsistencies are much more large respect to the parent domains, thus the sites that presents these mistakes are more vulnerable on subdomains rather than to their parent domains.
We collect also cookies that are at risk due to HSTS inconsistencies. We find out that the major chance to cookie leakage is from sites' subdomains. Once again we have confirmation that the potential entries for an attacker are the left vulnerable and unprotected subdomains.

Chapter 6

Related Work

The next two sections describe the two papers that were taken in consideration as a starting point for this thesis. They briefly review the use of HSTS among sites, and thus are useful in order to better understand the topic at stake.
The first paper is "The State of HSTS Deployment"; it concerned about the study conducted by Lucas Garron, Andrew Bortz and Dan Boneh, in 2013. They make a scan of the top visited sites of the Alexa top 1 million list in order to understand the correct set of HSTS header and reveal the common pitfalls. In particular, this case study is the first analysis on HSTS.
The second paper taken into account, "Upgrading HTTPS in Mid-Air", introduces an in-depth study of HSTS and public key pinning. The study was conducted by Michael Kranch and Joseph Bonneau from Princeton University.

6.1 The State of HSTS Deployment: A Survey and Common Pitfalls

As briefly introduced, the authors of this paper focus on HSTS (HTTP Strict Transport Security). They present the state of deployment and explain the common mistakes and drawbacks with HSTS configuration. As a matter of fact, the describe it in the IETF specification and browser implementation, which was the basis for their survey on top sites.

As it is specified in the article, HSTS mechanism is effective enough against

both active and passive attacks, it can help ensuring the security avoiding plain HTTP, which instead is still accepted by some sites. Indeed, three out of five of the major browsers have improved their security, such as Google, Twitter, iCloud, etc. The main advantage of HSTS is that it permits to specify a Strict-Transport-Security header, thus creating and entry that induces all future loads to be sent over HTTPS. These entries ease a wide range of attacks, such as the eavesdropping, injection/redirection, etc...
Later, they consider "example.com" to be typed and to be redirected to the "https://www.example.com" domain: if the site only sends HTTPS header over the latter, the future requests without www will not be secure, and therefore it is possible to state that parts of the domain are still vulnerable. They also propose some countermeasures in order to improve HSTS, such as the includeSubDomains directive in order to increase the subdomains protection.

Now, it will be presented the general overview that they present of HSTS. It is a mechanism introduced as ForceHTTPS in 2008 and later accepted by the IETF Standards. It is known to be an HTTP header that may be sent to browsers in order to require improved security for the domain at stake. HSTS is commonly described as a mechanism "to transform insecure URI references... into secure URI references". That means, it improves strict security in the mechanisms.
They present three ways to send HSTS header and these are:

- **Strict-Transport-Security: max-age=31536000**
  It defines the effect of the HSTS Policy for one year and it is applicable only to the domain of HSTS Host issuing it;

- **Strict-Transport-Security: max-age=15768000 ; includeSubDomains**
  It defines the effect of HSTS Policy for approximately six months and that it applies to the domain of the issuing HSTS Host and all of its subdomains;

- **Strict-Transport-Security: max-age=0**
  It specifies that the UA must delete the entire HSTS Policy associated with HSTS Host that sent the header.

Moreover, HSTS has to only be accepted and sent over HTTPS, and includeSubDomains is ignored for max-age=0.

HSTS is supported by: (1) Chrome, (2) Opera and (3) Firefox; but not by Internet Explorer and Safari. (1) Chrome: considers HSTS to be part of the cache and employs the following list directly, so that it enforces the security measures.

- force-https - permits HTTPS for a site by default (without expiration);
- includeSubDomains;
- pins - particular pinned certificates.

(2) Opera: has the same behaviour as Chrome above. It is closed-source, and thus does not provide documentation on HSTS. (3) Firefox: performs its straining in order to maintain the list fresh and to notice the latest host settings. A site is in the Firefox preload list if:

- it is present in the Chromium list;
- it sends an HSTS header;
- and the max-age is at leas 18 weeks.

As a matter of fact Firefox implements HSTS for all the sites in the preload list. The latter, includes also a boolean flag for includeSubDomains and this flag is se to TRUE when the includeSubDomains in HSTS header of the site. Moreover, it has an expiration time of 18 weeks from the time it is run the script, after that time Firefox will ignore it. Indeed, Firefox refers to HSTS as a part of "Site Preferences" in its "Clear Recent History" attribute.

As we introduced at the beginning of this section, the study of these authors was on the top sites. In particular, they surveyed the top 100k sites in the Alexa global rankings, requesting four root URL's each:

- http://example.com;
- http://www.example.com;
- https://example.com;
- https://www.example.com.

They assumed that the front pages were sufficiently of HSTS behaviour and then recorded how many sites: used HSTS with max-age > 0; used includeSubDomain; used HSTS with max-age = 0; and which sites responded over HSTS at all.
They only considered well-formed HSTS header and sent over HTTPS, however they did not categorize which site uses HSTS. All of the data collected were found using python script and were taken in October 2013. They only considered initial responses, being their goal about securing initial landing. (Tabelle)
What is enhanced is that the use of HSTS is to secure a domain, at least and thus it is usually preferable to secure all the visited (sub)domains; secure both "example.com" and "www.example.com"; and secure all the (sub)domain of "example.com".
They observed that a significant number of them sent an HSTS header, while others do not secure as many of their (sub)domains as they should.

Sites, moreover, should be aware of the issues connect to the use of includeSubDomain because it is present at least one subdomain that can break over the HSTS.
Therefore, to improve HSTS deployment, a site should operate considering that an attacker can run a MITM attack over HTTP and the strongest way to be protected is to redirect to HTTPS and have well-formed HSTS header with: every HTTPS request; a long max-age; includeSubDomains. It should also never send a HSTS header over HTTP, and once the site is engaged with HSTS then it should also be into the preload list of the browser. All of this is centred on the selection of the most safe options available and on making sure that the browser receives them. Sites must be aware of HSTS details, because it can only secure the current domain and the survey described in this section shows that, indeed, sites may expire weaknesses.

6.2 Upgrading HTTPS in Mid-Air: An Empirical Study of Strict Transport Security and Key Pinning

HTTPS is a stratification of HTTP traffic over the TLS/SSL encrypted transport protocols in order to achieve confidentiality and integrity; and it is the

dominant protocol used to ensure web traffic. The most considerable issue has been inconsistent and incomplete deployment of HTTPS: indeed, browsers must sustain both HTTPS and HTTPS connections.

Firstly, the main countermeasure to HTTPS stripping is *strict transport security* (HSTS), which is useful to browsers to ascertain a particular domain that can only be accessed by HTTPS.
Other that this, there are also concerns about certificate authority (CA) system weaknesses. Many protocols have been developed in order to keep the security against an attacker with a certificate signed by a trusted CA, and one of them is the key pinning. The latter is actually distributed as a preloaded policy with Chrome and Firefox.

Now then, we shall describe these two protocols in depth.
HTTPS: it is also known as "HTTP over TLS" and combines HTTP traffic with TLS (Transport Layer Security) indeed. TLS aims are confidentiality, integrity and authenticity, respectively against eavesdropper, manipulation by a network adversary, and by identifying the certificates. As a matter of fact one of its famous attackers is the Man-In-The-Middle attack (MITM).
Strict Transport Security: due to the fact that many web sites support only insecure HTTP, many domains serve traffic over both HTTP and HTTPS. Doing so, it enables a possible attack to take place and downgrading the security to plain HTTP by, for example, rewriting URLs to change the protocol. This type of attack is known as stripping attack. To contrast this attack it was proposed the "ForceHTTPS" to permit servers to request clients only communicate over HTTPS. This proposal was then called "HTTP strict transport security".

1. HSTS security model: HSTS works as a binary security policy; indeed by default it is claimed for a particular domain name, even if there is the optional includeSubDomain directive that applies to all subdomains of the policy;

2. HSTS headers: the main method o establish HSTS is setting HTTP header Strict-Transport-Security; in addition to the includeSubDomain explained before, there must be specified the max-age directive;

3. HSTS preload: to acknowledge the vulnerability of HTTPS stripping, both Chrome and Firefox are provided with a hard-coded list of do-

main which receive a preloaded HSTS policy. This reduced security for preloaded domains in order to maintain the authenticity of the browser installation;

4. HTTPS Everywhere: it is a browser extension that provides analogous protection but for a greater list.

Key Pinning: it is useful in order to force traffic to use HTTPS. Nevertheless, it has no effect with an attacker that can obtain a signed certificate and use it in a MitM attack. These issues show that requiring HTTPS via HSTS is not enough due to the rogue certificate threat.

1. *Pinning security model:* Key pinning indicates a limited set of public keys which a domain can use when it is creating a TLS connection;

2. *Pinning preloads:* Chrome has deployed preloaded pinning policies since 2011, despite only a handful of non-Google domains currently participate;

3. *Pinning headers (HPKP):* By HTTP Public Key Pinning (HPKP), sites may proclaim pinning policies via the Public-Key-Pins HTTP header. The syntax is very much like to HSTS, with an optional directive and includeSubDomains max-age directive.

Therefore, as the authors of this study explain, HSTS is still in early stages and pinning is too. Both of them already gained security for a lot of websites. Their research to observe how HSTS and pinning are deployed was developed through an automated web measurement platform. They used OpenWPM for their testing and performed a HTTP and HTTPS header-only crawl of the domain in the www subdomain of all the top million scale of Alexa sites.

Firstly, they present an overview of the current deployment of HSTS and pinning; they crawled and inspected the preload lists of Chrome and Firefox and they report:

a) Preload implementations;

b) Preloaded HSTS;

c) Preloaded pinning;

d) Dynamic HSTS;

e) HSTS errors;

f) Dynamic pinning (HPKP) deployment.

They then introduce the topic of mixed content: they explain that when an HTTP page loads sources from an HTTP origin, we talk about mixed content. It is considered dangerous due to the fact that an attacker may change the source delivered over HTTP and thus undermining both confidentiality and integrity of HTTPS, consequently undermining also the advantages of deploying HTTPS. Anyway, not all mixed content is dangerous in the same way and those that they take in consideration are:

- Pinning and mixed content;
- HSTS Mixed Content.

Despite the terminology is not standardized, mixed content is actually widely split into: active content (i.e. scripts, stylesheets, iframes, and Flash objects) that can totally change the content of the page's DOM or resupply data; and passive/display content (i.e. images, audio, or video) that can only change a constrained portion of the provided page and can not steal data. As said before for the terminology, this distinction between active and passive content is not standardized.

For what concern cookies, we explain them in depth in section 4.1.2, however the authors briefly explain this topic connected with cookie theft.
In particular, they state that an enduring issue with the web has been the inconsistency that there is between the same-origin policy that s defined for the majority of the web content and the one specified for cookies. In fact, for the latter, cookies are isolated bu the host and not by port or scheme. Consequently, cookies sent by HTTPS domain will be sent back to the same HTTP domain. Mainly, they are concerned with:

- secure cookies;
- interaction of secure cookies and HSTS;
- and, interaction of cookies and pinning.

Moving on, a reasonable amount of researchers focused on empirical mistakes with HTTPS and TLS. The most common works regard: cryptographic vulnerabilities including key revocation; factorable RSA keys; elliptic curve deployment errors in TLS; forged TLS certificates, etc.
In the paper though, they propose two new features of HTTPS: pinning and transport security, which are weaknesses at the HTTPS level more than the TLS level.
In order to improve HTTPS, they also propose some suggestions mainly regarding limiting the risk of rogue certificates:

1. **DANE (DNS-based Authentication of Named Entities):** it is a proposal that concerns the inclusion of the corresponding of public key pins in DNS records. DANE does not have any support for declaring policies applicable to all subdomains;

2. **Out-of-chain key pinning:** it is a good alternative to define a separate self-managed public key that has to sign all end-entity public keys. This is added to the request of a certificate chain that leads to a trusted CA;

3. **Public logging:** this proposal aims to demand that all valid certificates are publicly registered to assure rogue certificates are detected. Certificate Transparency (CT) is the most outstanding of these endeavours and it would avoid the majority of issues in which the strain on web developers is very low.

To conclude, the paper at stake explains that HSTS is still in its early stages of both adoption and pinning, even though both technologies already improved security for many websites. Pinning in particular is to account for the detection of CA arrangements since 2010; indeed the research shows that amounts of mistakes are undermining the potential security. Moreover, their paper is useful to spread that plainness is an important feature for developers: looking at some of the errors they studied, it is possible to state that better defaults might have helped. This is the example they report: "we would advocate for a default value of perhaps 30 days for HSTS policies set without an explicit max-age. Forcing all developers to choose this value has probably led to unwise choices on both ends of the spectrum in addition to malformed headers. Setting sensible defaults for pinning is far more challenging—there is no clear way to choose a default "backup pin" besides the

values currently being used".

This problem implies that pinning will probably never be an easy, as they say, "on switch" for developers. It may be, though, if TLS certificate management can be totally extracted.
Moreover, if both HSTS and pinning had policies that apply to subdomains by default and an option for disabling or turning it off for particular subdomains, they would be more secure. Even expanding cookies' attribute to require pinning as secure as HTTPS, seems to be a step closer to the technology matching developer expectations.
Indeed, in their paper the authors promote simplifying HTTPS security features in order to have an easier configuration. HSTS and pinning are surely not the last improvement, as a matter of fact there are two structures that are being standardized to be set in HTTP headers.
Combining dynamic HSTS and pinning declaring in a more flexible and extendable structure may be beneficial in order to permit developers to declare once, rather than expect them to learn new syntaxes each time a new patch is employed.

Chapter 7

Conclusion

Nowadays, the topic of security has become a popular issue due to the contemporary digital society in constant development. The HTTP protocol does not give the necessary security avoiding attacks from malicious attackers. There were many security mechanism implemented for internet security like SSL, TLS and later HSTS.
The TLS (Transport Layer Security), and its predecessor SSL (Secure Socket Layer), are cryptographic protocols designed to ensure a secure communication over the network. These protocols prevent tampering, falsification and interception of data. However, the HTTP over TLS used alone is not enough against malicious intent. In 2009, was presented an elaborate type of attack against HTTPS called SSL stripping.

In order to avoid SSL stripping, websites that receive HSTS header over HTTP automatically redirect the connection over HTTPS (even if the specification does not require it) or they can reject the connection. These best practices can be resumed as follows:

- Hosts should declare HSTS policy at their top level domain name. In other words, the policy should be enabled for https://example.com;
- Have the *max-age* directive set up to 1 non leap year in seconds (31.536.000 seconds);
- *includeSubDomains* should be present.

The HSTS policy must never be send over plain HTTP.

HSTS also can have the *preload* directive. Being preloaded means that the browsers are already informed that before the communication begins, the host requires the use of a secure connection (over SSL/TLS or other encryption protocol). Being preloaded also removes any malicious intent from an attacker to intercept and tamper any communications that are redirected from HTTP.
A possible vulnerability is the Bootstrap MITM (Man-In-The-Middle). This type of vulnerability shows up when the user manually enters, or clicks on a link to an unknown host that uses an HTTP connection instead of HTTPS. Because of this, the user agent is subjected to various attacks

In order to study HSTS, we used NodeJS and Puppeteer and collect as much headers as possible both from insecure and secure request to the sites present in the Tranco[14] list and with our study we aimed to inform users about the correct configuration and gave an overview about the current deployment of this essential security mechanism. Then we defined tiers, from A to E, that corresponds to a possible scenario that an attacker may encounter in order to exploit the site's vulnerabilities. In some scenarios we integrated other mechanism in order to verify the integrity of the scenario such as upgrade-insecure-request and cookies prefix.

First of all, we replicated the experiments taken from the articles on which this thesis is based. We added some extra control, like the preload condition of the sites. In our study we take into consideration the first hundred thousand sites of Tranco list [14]. In the first test we illustrate the incidence of HSTS over HTTPS sites. Then, we divide the sites into insecure and secure version and, for the first ones, we test "worst practices" and for the latter we show the current deployment of HSTS among the HTTPS version of the sites. Finally, we show the percentage of sites able to mitigate an attack more or less elaborate from an attacker.

The purpose of this study is to inform the final user about the risk that it may encounter while surfing the net and assert the state of HSTS. Extending the previous researches, we can state that the current deployment of this security mechanism has increase its adoption over the years, and the number of sites that implemented it incorrectly has significantly decreased on the their parent domains.

Due to both legal and ethical reasons we have not tested the effective vulnerabilities of the crawled sites. The proposed scenarios, came out from analysis of most common situations in the wild and not from our technical analysis. Thus, our study must be considered as a possibility of attacks by evil users. A limitation of STS is the dimension of the preload list. It is reasonable to think that every site in the net could not be part of the preload list and consequently any conversation among the user and the site cannot start securely. A further work could be thinking about how to implement a mechanism that helps HSTS in order to avoid that every first communication to a site begins over plain connections.

Finally, we encountered many possible configuration for HSTS, from those that set a lower max-age directive to those that are present in the preload list. Moreover, the most common mistakes that we encountered are from sites subdomains: a clerical mistake is that there is confusion in the declaration of HSTS. To make the configuration easier to use, it might be useful to establish a standard configuration for declaring the strict transport security header with the possibility of being part of the preload list, or alternatively force all server to respond over a secure channel.

Bibliography

[1] Stefano Calzavara, Tobias Urban, Dennis Tatang, Marius Steffens, and Ben Stock. Reining in the web's inconsistencies with site policy. In *The 2021 Network and Distributed System Security Symposium*, NDSS '21, 2021.

[2] Google Developers. `https://developers.google.com/web/tools/puppeteer`.

[3] Mozilla Developers. `https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Strict-Transport-Security`.

[4] Mozilla Developers. `https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies`.

[5] Mozilla Developers. `https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP`.

[6] Alexis Deveria. `https://caniuse.com/stricttransportsecurity`.

[7] Roy T. Fielding and Julian Reschke. Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. RFC 7231, June 2014.

[8] Lucas Garron, Andrew Bortz, and Dan Boneh. The state of hsts deployment:a survey and common pitfalls. `https://garron.net/crypto/hsts/hsts-2013.pdf`.

[9] Jeff Hodges, Collin Jackson, and Adam Barth. HTTP Strict Transport Security (HSTS). RFC 6797, November 2012.

[10] D. Hubbard. Cisco Umbrella 1 million. [Online]. `https://umbrella.cisco.com/blog/2016/12/14/cisco-umbrella-1-million/`, December 2016.

[11] Amazon Web Services. Inc. Alexa Top Sites. https://aws.amazon.com/alexa-top-sites/, March 2018.

[12] D. Jones. Majestic Million CSV now free for all, daily. [Online]. https://blog.majestic.com/development/majestic-million-csv-daily/, October 2012.

[13] Michael Kranch and Joseph Bonneau. Upgrading https in mid-air: An empirical study of strict transport security and key pinning. In *The 2015 Network and Distributed System Security Symposium*, NDSS '15, February 2015.

[14] Victor Le Pochat, Tom Van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczyński, and Wouter Joosen. Tranco: A research-oriented top sites ranking hardened against manipulation. In *Proceedings of the 26th Annual Network and Distributed System Security Symposium*, NDSS 2019, February 2019.

[15] Bodo Möller, Thai Duong, and Krzysztof Kotowicz. This POODLE Bites: Exploiting The SSL 3.0 Fallback [PDF]. https://https://www.openssl.org/~bodo/ssl-poodle.pdf, September 2014.

[16] NodeJs. https://nodejs.org/it/.

[17] Quantcast. Open internet ratings service. https://web.archive.org/web/20070705200342/http://www.quantcast.com/, July 2017.

[18] Eric Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, August 2018.

[19] Preload submission form. https://hstspreload.org/.

[20] w3counter. https://www.w3counter.com/globalstats.php.

[21] Wikipedia. Content security policy. https://en.wikipedia.org/wiki/Content_Security_Policy.

[22] Wikipedia. Http cookie. https://en.wikipedia.org/wiki/HTTP_cookie.

[23] Wikipedia. Http strict transport security. https://en.wikipedia.org/wiki/HTTP_Strict_Transport_Security.