HSTS-Enforced

Enhancing HTTP Strict Transport Security through Secure-by-Default Principles

Aaron J. van Diepen

Delft University of Technology

TU Delft

HSTS-Enforced

Enhancing HTTP Strict Transport Security through Secure-by-Default Principles

by

Aaron J. van Diepen

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Monday August 26, 2024 at 10:00 AM.

Student number:	4695720	
Project duration:	November 20, 2023 – August 12, 2024	
Thesis committee:	Prof. dr. ir. Fernando A. Kuipers,	Delft University of Technology, supervisor
	Prof. dr. ir. Georgios Smaragdakis,	Delft University of Technology
	Adrian Zapletal,	Delft University of Technology, advisor

This thesis is confidential and cannot be made public until Wednesday, December 31, 2025.

Cover: Immeuble du Crédit Lyonnais (siège) by Renaud d'Avout under CC BY-SA 3.0 Deed

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft

Abstract

Over the years, the web has slowly been moving towards more security. This is done to ensure integrity, authenticity, and confidentiality of the communication between clients and servers. The most significant improvement to the security on the web has been HTTPS, which provides secure communication using encryption. However, downgrade attacks can bypass HTTPS entirely by reverting the communication to the insecure HTTP protocol. HSTS is the primary defense against such attacks. However, previous research has uncovered numerous vulnerabilities in the HSTS protocol, particularly those that allow attackers to disable HSTS by invalidating its state and a method that uses HSTS headers to enable websites to track users.

In this thesis, we present HSTS-Enforced, an alternative to traditional HSTS. HSTS-Enforced effectively prevents downgrade attacks by employing a Secure-by-Default approach. Website administrators can explicitly opt out of security by specifying an HTTP-Required indicator. We propose two indicators: a new DNSSEC record and the HTTP-Required Preload list.

We demonstrate the effectiveness of HSTS-Enforced, through the creation and validation of a protocol implementation encompassing both client and server-side components. Our evaluation reveals that HSTS-Enforced eliminates the vulnerabilities found in conventional HSTS. Additionally, we show that while enhancing security, HSTS-Enforced imposes a minimal load on all involved components (i.e., client, network, and server).

Acknowledgments

I would like to express my deepest gratitude to all of the people who have contributed to the successful completion of this master thesis.

Foremost, I would like to express my sincere gratitude to my supervisors, Prof. Fernando A. Kuipers and Adrian Zapletal, for their guidance, support, and invaluable feedback throughout the research process.

I would also like to extend my heartfelt gratitude to my family and friends, as well as to my roommates, Simon Tulling and Dion Rovers. A special thanks goes to my friend, Peter Knot, for their generous contributions of time and insights during the early stages of the project, which has significantly enhanced the proposed solution.

Lastly, I would like to express my deepest gratitude to all who work on and facilitate the open-source ecosystem: your work empowers countless developers and projects around the world. This project's success is a testament to your dedication and generosity.

To the open-source communities of BIND9, PowerDNS, Unbound, and Chromium, your tireless efforts in developing and maintaining these invaluable projects were instrumental in making this project possible. Special thanks to everyone who contributed to the detailed documentation of these projects, which has been an indispensable resource throughout this journey.

To the Internet Engineering Task Force (IETF), for their unwavering commitment to maintaining internet standards and ensuring access to them free of cost. Your dedication is a cornerstone of the open internet.

To the maintainers of Git, your tool is essential for version management, which not only helped keep development organized during this project but also facilitates coordination within large open-source communities.

To GitHub, for hosting the majority of open source software codebases or copies of those codebases and for fostering a collaborative environment that accelerates innovation.

Lastly, to all other open-source communities, thank you. Your dedication, selflessness, and openness have been a true inspiration. Thank you for your tireless efforts in advancing technology and for the support and inspiration you provide to developers worldwide.

Aaron J. van Diepen
Delft, August 2024

Nomenclature

ACME Automated Certificate Management Environment. 12
AES Advanced Encryption Standard. 3, 16
API application programming interface. 10, 25, 34, 35

CA Certificate Authority. 7, 8, 12–14, 29
CAA Certificate Authority Authorization. 13, 14
CSP Content Security Policy. 13
CSR Certificate Signing Request. 7, 8

DANE DNS-Based Authentication of Named Entities. 13, 33
DES Data Encryption Standard. 3
DNS Domain Name System. 2, 4–6, 10–13, 21–23, 25, 26, 28–36
DNSSEC Domain Name System Security Extension. 2, 5, 6, 11–13, 22, 23, 25, 26, 28–31, 33–37
DoH DNS-over-HTTPS. 34, 37
DoQ DNS-over-QUIC. 34, 37
DoT DNS-over-TLS. 34, 37
DS Delegation Signers. 5, 6, 28, 30

ECC Elliptic Curve Cryptography. 3
EFF Electronic Frontier Foundation. 33
eTLD effective top-level domain. 4, 10, 23, 25, 30, 35

HSTS HTTP Strict Transport Security. 1, 2, 8–12, 14–34, 36, 37
HTTP Hypertext Transfer Protocol. 1, 2, 6–8, 10–13, 18–37
HTTPS Hypertext Transfer Protocol Secure. 1, 2, 7–16, 18–21, 23–25, 27–30, 32–37

I2P Invisible Internet Project. 15
IANA Internet Assigned Numbers Authority. 20
ICANN Internet Corporation for Assigned Names and Numbers. 4
IP Internet Protocol. 4, 5, 7, 20, 23, 25, 27, 29, 37
IPC inter-process communication. 11
ISP Internet Service Provider. 5

JSON JavaScript Object Notation. 9, 22, 23, 27, 28

KSK key signing key. 5, 6

NS Name Server. 4, 5
NSEC Next Secure. 6, 30
NTP Network Time Protocol. 19

OTR Off-the-Record. 16

PGP Pretty Good Privacy. 16
PKI Public Key Infrastructure. 7
PKP Public Key Pinning. 9, 10, 14, 26, 27

RDATA Resource Data. 22, 26
Root CA Root Certificate Authority. 7
RRSET Resource Record Set. 6
RRSIG Resource Record Signature. 6, 22, 26
RSA Rivest-Shamir-Adleman. 3
RTT Round-Trip Time. 29, 30

SHA Secure Hash Algorithm. 16
SOA Start of Authority. 4, 5, 30
SSL Secure Sockets Layer. 7, 13, 14

TCP Transmission Control Protocol. 6, 7
TLD top-level domain. 4, 5
TLS Transport Layer Security. 7, 13, 14, 37
TLSA Transport Layer Security Authentication. 13, 33
TTL time to live. 4

URL Uniform Resource Locator. 6–8, 12, 24, 27, 30, 34

VPN Virtual Private Network. 16

X3DH Extended Triple Diffie–Hellman. 17
XSS Cross-Site Scripting. 13

ZLTP Zero-Leakage Transfer Protocol. 14
ZSK zone signing key. 5, 6

Contents

1 Introduction

1.1. Motivation

With the rising prevalence of online banking, the increasing digitization of government agencies and critical infrastructure, and rising concerns about privacy, securing web communications is more crucial than ever. While HTTPS [79] has become the standard for secure communication over the web in favor of unsecure HTTP [29], its ability to ensure security and privacy relies heavily on complementary mechanisms like HTTP Strict Transport Security (HSTS) [36]. A significant threat in this context are downgrade attacks, where attackers attempt to force a connection downgrade from HTTPS to HTTP, enabling data interception or manipulation [61, 62]. HSTS addresses this by instructing browsers to exclusively use trusted HTTPS connections for a specified duration, effectively preventing unsecure connections.

HSTS can traditionally be enabled using two methods: HSTS headers and HSTS preloading. HSTS headers are transmitted as part of HTTPS responses and stored in the user agent's HSTS cache. This ensures that once a website is visited using HTTPS, the browser remembers this preference for future visits, reinforcing the restriction to only use trusted HTTPS connections. HSTS preloading goes a step further by allowing websites to be hardcoded into a browser's lists of sites that have HSTS enabled. This ensures these sites are only accessed using HTTPS, even before the first visit. However, both methods have notable limitations. Many websites send incorrectly configured HSTS headers, rendering them invalid such that user agents do not add these websites to their HSTS cache [32, 57, 85]. When using HSTS headers without HSTS preloading, the first connection and connections after the HSTS cache entry has expired remain vulnerable. Additionally, when HSTS preloading is not used, HSTS headers can be leveraged by servers to track users across networks and sessions by exploiting the data stored in the cache [95]. Furthermore, both the HSTS cache and the Preload list are susceptible to attacks that force their entries to expire, nullifying their security benefits [84]. Moreover, the number of websites on the HSTS Preload list is disproportionately small [3, 4, 53, 54, 74, 90]. In particular, banking websites are rarely on the list, even though those websites are among the most crucial to be on it.

While HSTS is an effective tool, these limitations expose vulnerabilities that can be exploited, often due to human error. This thesis aims to address these gaps by proposing a more reliable, user-friendly solution that raises the standard for secure web communications. The new protocol aspires to set a higher standard for secure communications on the web. Through the Secure-by-Default paradigm, it ensures that security is not an optional feature that can be circumvented by attackers, but a fundamental characteristic of all web interactions.

1.2. Secure-by-Default

In the evolving cybersecurity landscape, the concept of Secure-by-Default has emerged as a pivotal strategy to counter the growing threats to digital systems. Advocated by numerous government agencies worldwide [20], Secure-by-Default represents a significant shift in how systems are designed and configured. Secure-by-Default refers to an approach where systems are inherently secure from the

outset, requiring minimal user intervention to achieve and maintain a high standard of security. Unlike traditional security models that rely on users to manually enable security features, Secure-by-Default integrates robust security mechanisms into the core architecture of software and hardware. This proactive approach not only reduces the likelihood of human error, but also ensures that security is a foundational aspect of system design, rather than an afterthought. By embedding security into the default settings, these systems can better safeguard against vulnerabilities, enhancing the overall resilience and trustworthiness of digital infrastructures.

1.3. Contribution

This thesis introduces *HSTS-Enforced*, a new mechanism designed to replace traditional HSTS headers and preloading, addressing their vulnerabilities. HSTS-Enforced ensures that HSTS is enabled by default, with website operators having the option to explicitly opt out using HTTP-Required indicators. Our proposed HTTP-Required indicators are easy to use: we propose a new DNS [65] record type labeled *HTTPREQ*, which leverages the DNSSEC [5] cryptographic chain to prove that a website requires HTTP, and the HTTP-Required Preload List, which functions similarly to the HSTS preloading but "in reverse".

Defaulting to HSTS ensures security by default, preventing erroneous configurations from affecting security, and removes the need for HSTS headers and the HSTS cache, eliminating the possibility of downgrade attacks and user tracking via HSTS headers. Furthermore, HSTS-Enforced makes it easier for website operators to achieve maximum security, for no additional work is required from their side. On the other hand, opting out of security, if desired, is still easy.

1.4. Report Structure

The remainder of the thesis is structured as follows. Chapter 2 provides background information on related technologies, such as symmetric and asymmetric encryption, DNS, DNSSEC, HTTP, HTTPS, downgrade attacks, HSTS, HSTS preloading, and browsers. Next, Chapter 3 outlines related work aimed at increasing security and privacy on the web. Chapter 4 articulates the problems present in HSTS that are addressed by this thesis. Chapter 5 presents the design and implementation details of HSTS-Enforced. Chapter 6 evaluates HSTS-Enforced to see whether it is more secure than HSTS and if it can be adopted as a part of the web. Chapter 7 discusses the limitations of this research, the design choices made, the relationship between HSTS-Enforced, HTTPS-Only Mode, and HTTPS Everywhere, and provides general recommendations for related security protocols. It also provides insights into how a roll-out of HSTS-Enforced can be facilitated and discusses the practical implications of the protocol on the general web. Finally, Chapter 8 concludes the thesis and suggests potential directions for further research.

2

Background

This chapter presents the relevant background knowledge used in this thesis. Section 2.1 provides a high-level explanation of cryptography and how different cryptographic mechanisms facilitate the secure exchange of information. Section 2.2 explains how human-readable names for addresses on the internet - known as domain names - work and how information about them can be accessed. Additionally, it explains how this system was expanded to validate authenticity and ensure the integrity of this information using cryptography. Section 2.3 explains how documents and resources are communicated across the web. Additionally, it explains how cryptography is used to secure these communications. Section 2.4 explains downgrade attacks and the need for a protocol to communicate when only secure connections should be allowed in order to prevent such attacks. Sections 2.4.1 and 2.4.2 explain the current workings of the protocol responsible for communicating this information. Finally, Section 2.5 provides an overview of the distribution of users across browsers and provides more detail about the inner workings of the browser with the largest market share, namely Chromium.

2.1. Cryptography

Cryptography is a broader term that encompasses various techniques used to secure communication and information. Three of the most commonly used cryptographic techniques are encryption, decryption, and digital signatures. These techniques all rely on cryptographic keys. Cryptographic keys provide information about the cryptographic operations that should be applied to a piece of data in order to encrypt or decrypt it. There are two different types of keys: symmetric keys and asymmetric keys. The following sections explain the two types of cryptography they are used in and the different techniques such as encryption, decryption, and digital signatures they facilitate.

2.1.1. Symmetric Cryptography

Symmetric cryptography uses a single key for both encryption and decryption. Common symmetric cryptographic algorithms include Advanced Encryption Standard (AES) [70] and Data Encryption Standard (DES) [71]. These algorithms convert data in a specific manner that can be reversed using the same process as the original steps taken to scramble the data. To securely exchange messages using a symmetric key, both sender and receiver must share a symmetric key. If this key sharing is secure, only the sender and receiver can read the messages. However, if the key sharing is compromised, anyone with the key can decrypt the messages.

2.1.2. Asymmetric Cryptography

Asymmetric cryptography, or public-key cryptography, uses a pair of keys: a public key and a secret (private) key. The secret key is generated randomly, and the public key is derived from it mathematically. The public key is shared openly, while the secret key is kept hidden. Common asymmetric cryptographic algorithms include Rivest-Shamir-Adleman (RSA) [80] and Elliptic Curve Cryptography (ECC) [93]. Public-key cryptography is used in two primary ways: to encrypt data using the public key and decrypt it with the secret key, or to create digital signatures using the secret key.

Encrypting with a public key and decrypting with a secret key ensures content confidentiality; only the secret key owner can decrypt the content. However, this does not authenticate the origin of messages, as anyone with the public key can create such messages.

A digital signature can be used to authenticate the origin of a message and assure its integrity since the secret key is required to create or modify them. To create a digital signature for a piece of data, we pass the data through a hashing algorithm and encrypt the output with the secret key. A hashing algorithm takes an input and returns a fixed-size output, known as a hash. Given a fixed input, the generated hash is always the same. Additionally, it is computationally infeasible to find two different inputs that produce the same hash value or to reverse the hashing operation to receive the original content. Thus, hashes are considered unique identifiers for the original values they represent. When both the digital signature and the original message are received, one can use the digital signature to verify both data integrity and the origin of the data. To verify the origin of the data and its integrity, the receiver decrypts the signature using the public key, applies the same hashing algorithm to the plaintext data, and verifies that they are identical.

2.2. Domain Name System (DNS)

The Domain Name System (DNS) [65] provides human-readable addresses (domain names) for devices on a network. Internet Protocol (IP) addresses, which are numerical and hard to remember, are typically used to identify devices. DNS links these IP addresses to text-based domain names, making it easier for humans to navigate the internet. Domain names are dynamically bound to one or more IP addresses, which allows for the distribution of requests between multiple servers. Section 2.2.1 describes the hierarchical structure of domain names. Section 2.2.2 details how information is specified within DNS. Section 2.2.3 describes how this information is retrieved using the DNS infrastructure. Section 2.2.4 covers how cryptography is used to validate the responses.

2.2.1. Domain Names

Domain names consist of parts called labels, separated by dots and read hierarchically from right to left. For example, in *www.tudelft.nl*, the rightmost label is the top-level domain (TLD) *nl*, followed by the second-level domain *tudelft.nl*, and the third-level domain *www.tudelft.nl*. Subdomains are domains that are below another domain. For instance, *www.tudelft.nl* is a subdomain of both *tudelft.nl* and *nl*.

The Internet Corporation for Assigned Names and Numbers (ICANN) assigns TLDs to different instances responsible for further domain distribution, known as TLD owners. TLD owners often appoint registrars to handle registration of the subdomains of these domains. Domains that can be registered directly are known as main domains. Main domains can be any level of domain, but most are second-level and third-level domains. The domains directly above main domains are known as effective top-level domains (eTLDs) or public suffixes. The public suffix website [66] maintains a list of existing eTLDs. Subdomains of main domains are often used by the same website operator to provide different services under a common name.

2.2.2. Resource Records

Resource records are used to exchange information about domain names and are specified by authoritative name servers, which are one of two types of servers used in DNS. The second type of server is discussed in the next section. Authoritative name servers manage various information for designated domains and their subdomains, collectively called a zone. Resource records are divided into five fields: name, time to live (TTL), class, type, and data. An example of a resource record is *tudelft.nl. 600 IN A 130.161.128.82*. The name field identifies the relevant domain or subdomain, here *tudelft.nl*. The TTL field indicates the duration for which a record should be deemed valid upon receipt, in the example this is 600 milliseconds. The class field specifies the network type that this record is used for. As in the example often this contains the value "IN", which denotes internet-related DNS records, but other classes exist and are used for lesser-known alternatives to the internet. The type field denotes the record type. The main record types that facilitate DNS resolution for the web are *SOA* records, *NS* records, *A* records, and *AAAA* records. The data field of a record contains varying info based on the type of record.

An Start of Authority (SOA) record marks the top of a zone. Its data field contains administrative information about this zone, such as the primary name server responsible for its data, the email of the

domain administrator, and the domain serial number which tracks changes to the zone's record. Name Server (NS) records list name servers responsible for a zone, with the data field containing their domain names. *A* and *AAAA* records link a domain name to an IPv4 or IPv6 address respectively, supporting fallback servers or load balancing by allowing multiple records for the same domain, each with a single IP address. The example specifies that *tudelft.nl* has an IP address of *130.161.128.82*.

2.2.3. Resolving Queries

To resolve a record request to their response data we use a recursive name server, also known as a resolver. Any software that wants to resolve a record can decide which resolver it uses, but generally, the resolver specified by the operating system is used. In most cases, this is a locally hosted resolver, which provides local caching, but forwards any requests it can not answer to another resolver specified in its settings.

Most resolvers responsible for resolving a request are preloaded with the IP addresses of 13 authoritative servers responsible for providing information about the authoritative servers responsible for TLDs. These name servers are known as the root name servers [42]. Next to these servers, a resolver can be preloaded with other authoritative servers that they directly bind to specific domains, this allows for the specification of local DNS information that is not part of the public DNS infrastructure.

Upon receival of a request, a resolver first checks whether it has the requested information. If it has the required record, this record is simply returned. If it does not, it continues by contacting the lowest-level authoritative server it thinks is responsible for the record. If it has not done any recursion before this is one of the preloaded servers. This authoritative server then replies with a response, which contains various sections. In case the server has one or more records matching the request, the response includes an answer section with these records. In case the name is part of its zone but no record matching the request is present, the response only contains an authority section specifying the *SOA* record of the server. This indicates to the requester that no such record exists. In case another authoritative server is responsible for that particular section of the zone, the server responds with an authority section containing the *NS*, *A*, and *AAAA* records pointing to the responsible server. Since authoritative servers only know this information for the servers hosting zones that are their direct descendants, this often requires the resolver to contact the servers specified in such authority sections until the requested record or an authority section pointing to the contacted server is received.

While anyone can host a resolver, there are many publicly available resolvers, for example, Google's Public DNS, Cloudflare DNS, OpenDNS, and Quad9 DNS. Additionally, Internet Service Providers (ISPs) also generally host a resolver.

2.2.4. Domain Name System Security Extensions (DNSSEC)

To achieve security on the web, ensuring the integrity and authenticity of DNS data is paramount. Traditionally, DNS responses lack built-in security measures, leaving them vulnerable to various attacks. Domain Name System Security Extension (DNSSEC) [5] is a set of cryptographic protocols designed to address these vulnerabilities and provide a secure DNS infrastructure. It works by adding a layer of security to DNS responses through digital signatures, thereby enabling DNS clients to verify the authenticity and integrity of the data they receive.

At its core, DNSSEC employs public-key cryptography to sign DNS records, creating a chain of trust that extends from the root DNS servers down to individual domain zones. This cryptographic chain allows DNS resolvers to validate DNS responses by verifying their digital signatures against the corresponding public keys stored in DNSSEC-enabled zones. The information about the cryptographic chain is provided to resolvers using *Delegation Signers (DS)* and *DNSKEY* records. The *DNSKEY* record contains a public signing key, and the *DS* record contains a hash of a *DNSKEY* record signed with a different secret key. Generally, two public-key pairs are used per zone. These two public-key pairs are known as the key signing key (KSK) and zone signing key (ZSK). However, the use of two pairs per zone is not a strict requirement for DNSSEC, one pair of keys could also be used to fulfill the roles of both of these pairs. The KSK is used to verify the authenticity of a zone. A *DNSKEY* record for its public key is published in the zone, while a *DS* record is published in its parent zone. The KSK's secret key is then used only to sign a *DS* record for the ZSK. The *DNSKEY* record for this ZSK is published in the same zone. This approach is often used as it can increase security. The ZSK's *DS* and *DNSKEY* record

being part of the same zone allows the ZSK to be updated more often than the KSK, which requires communication with the parent zone to update its *DS* record. Website operators who want to enable DNSSEC for a domain can request the addition of a *DS* record through their registrar. After providing the desired public key, the registrar then communicates with the authoritative server responsible for the parent domain that a signed hash of the key should be provided in the form of a *DS* record.

To sign a DNS zone, we first group records into *Resource Record Set (RRSET)* records based on their type and name. DNSSEC optionally also provides a way to securely communicate the absence of a domain through an *Next Secure (NSEC)* or *NSEC3* record. These records use the alphabetically next domain to deny the existence of any previous domain. *NSEC3* was introduced as a replacement for *NSEC*. *NSEC* exposed the next domain in plain text, allowing malicious users to quickly discover all domains in a zone. *NSEC3* improves on this by using hashes to hide the name of the next domain. We additionally generate one type of these records for the zone. The ZSK's secret key is then used to create signatures of the *RRSET* and *NSEC* or *NSEC3* records, including the name and type. Using type *Resource Record Signature (RRSIG)* records these signatures are communicated as part of the response whenever a request indicates DNSSEC is required. An *RRSIG* record additionally specifies a label count indicating how many labels of the domain were explicitly specified as part of the signed data. The label count value is used to create signatures for wildcard records. Whenever the label count of the response *RRSIG* record is lower than the label count for which the request was made this indicates that the domain name used during signing started with a wildcard and ended with that label count's value.

Using flags, a requester can indicate whether or not they require DNSSEC validation or not. When validation is requested, the resolver uses the cryptographic chain to verify the signature of a record. To do this it is preloaded with a *DS* record for the root authoritative servers. It uses this record to verify the *DNSKEY* record of the authoritative server, which it then uses to validate its response. In case an authoritative server tells the resolver to contact another authoritative server, the resolver requests the *DS* record for this second server from the first server, which has signed the record with the secret key corresponding to its *DNSKEY* record. This *DS* record validates the *DNSKEY* record of the second server. If all records are successfully validated, the resolver responds with a flag that indicates whether DNSSEC validation was successful.

DNSSEC prevents unauthorized modification or manipulation of DNS data, enhancing the overall security and trustworthiness of the internet's DNS infrastructure. However, despite its security benefits, widespread adoption of DNSSEC has been somewhat limited due to various factors, including the complexity of implementation, interoperability issues, and the need for coordination among DNS stakeholders. However, as awareness of cybersecurity threats continues to grow, DNSSEC adoption is gradually gaining traction, with many root name servers and various other authoritative name servers now supporting DNSSEC validation.

2.3. Hypertext Transfer Protocol (HTTP)

The Hypertext Transfer Protocol (HTTP) [29] is the main protocol used for data exchange on the web. In broad terms, the web is the part of the internet that transfers the information required for the functioning of websites between web servers and web clients. Web clients are also referred to as user agents. The most commonly used type of user agents are web browser. HTTP messages are communicated over the Transmission Control Protocol (TCP). TCP provides a reliable connection between servers over which HTTP requests and responses can be communicated.

HTTP requests and responses typically include information such as the Uniform Resource Locator (URL), the HTTP version, the type, the body or content, and additional information in the form of headers. For modern HTTP messages, the version is often HTTP/1.1, as this is the lowest version of HTTP still supported by the current web standards. The type of a request indicates its intent, the most common types are *GET* and *POST*. The only difference between *GET* and *POST* is that a request of type *POST* allows the specification of data in the body, while a *GET* request does not. As an answer to a request, the body of a response carries the main information, such as an image or a document. HTTP headers are included in both requests and responses. They can be used to provide additional information about the content of the requests or responses. Header fields are identified by a label and can contain any text value. A special type of header field is a cookie field. A cookie allows clients and

servers to specify consistent information about each other. A particular type of cookie is a session cookie. When you log in to a website, the server can return a session cookie. Then, upon requesting another resource, the client includes this cookie. The server uses this information to verify that the user is who they claim to be. Alongside every HTTP request to the same server, all cookies received from and set by the server are included.

HTTP sends data directly over the TCP connection. This data stream is not encrypted. Thus, HTTP offers minimal security. Anyone who intercepts an HTTP message can forge, modify, or read their content.

2.3.1. Hypertext Transfer Protocol Secure (HTTPS)

Hypertext Transfer Protocol Secure (HTTPS) [79] was introduced to enhance security, building on HTTP by adding a Secure Sockets Layer (SSL)/Transport Layer Security (TLS) channel on top of the TCP channel. When a client initiates an HTTPS/1.1 connection to a web server, first a TCP connection is created. After establishing the TCP connection, the server starts by sending an X.509 certificate. This certificate is linked to a domain or IP address and contains a public key, a digital signature, and information about the instance that signed it. After receiving the certificate, the client first verifies the server's certificate using a chain of trust established by the Public Key Infrastructure (PKI) as detailed in Section 2.3.1 to authenticate that the server is authorized to respond for the domain or IP address. The public key contained in the certificate, optionally in combination with a second asymmetric key pair generated by the client, is then used to securely exchange a symmetric key. Using this symmetric key a secure channel is established over which HTTP messages are securely exchanged. This process ensures that data transmitted between the client and server is secure, protecting against eavesdropping and tampering. To distinguish between when a client should contact HTTP or HTTPS, the URL pointing to a resource specifies which protocol should be used.

Authenticating Certificates

To validate the X.509 certificate is from a trusted authority the PKI is used, which is a chain of trust originating from multiple trusted authorities known as Root Certificate Authorities (Root CAs). These instances all have a self-signed certificate, known as a root certificate. Using the secret key associated with the public keys in these root certificates several other certificates have been signed and distributed to Certificate Authorities (CAs). CAs use their certificates to sign other certificates and distribute them to domain owners. When a website is accessed over HTTPS the browser checks the received certificate's signature by validating its authenticity against the parent certificate until it reaches a root certificate. Thus, using the chain of trust, the origin of a message can be authenticated.

It is possible to create HTTPS connections using certificates that are not signed using the PKI. However, if the signing of the certificate does not resolve towards a root certificate, the connection will be labeled as an unsecure HTTPS connection. Such custom certificates are often used when HTTPS is desired for non-public domains or IP addresses since certificates for these addresses cannot be requested from an official CA. However, one can locally extend the certificate chain by adding a custom certificate to the root certificates on a machine. By doing this any machine that includes this custom root certificate, can construct secure HTTPS connections with servers that use certificates signed using the secret key of this custom certificate. When extending the root certificates with a custom certificate, system administrators have to be very careful, since if the secret key of the custom root certificate is obtained by a malicious user, they can create certificates that are trusted by all systems that have included this custom root certificate. The malicious user is then able to sign a certificate for any possible domain and the machines that include the custom root certificate see connections made using these custom signed certificates as secure HTTPS connections, effectively losing all security of HTTPS from the perspective of the malicious user.

Creating Certificates

To create a signed certificate the requesting instance first generates a public-key pair. The generated public key together with information about the requesting organization and domain name forms a Certificate Signing Request (CSR). This CSR is then signed using the generated secret key. After receiving the CSR together with its digital signature, the instance from which a signed certificate is requested can verify the information in the CSR and the digital signature to ensure the legitimacy of

the request. After verifying the legitimacy of the CSR a certificate can be created using the public key and the information from the CSR. After signing the certificate with its secret key the resulting X.509 certificate can be returned to the requester. A certificate for a domain can be acquired from any CA and it is the task of the certificate authorities to confirm ownership of a domain before signing any certificates relating to a domain.

CAs, such as DigiCert and Comodo, often offer various ways to verify ownership of the domain. Examples of this include: sending an email to addresses associated with the domain such as admin@domain.com, placing a file at a specific location on the server associated with the domain, placing a record at a specific location in the name server for the domain, or calling a phone number that has been associated with the domain upon registration. Most certificate authorities require a certificate to be purchased since they provide the service of verifying ownership of the domain and signing the certificate. Often the registrar that you registered your domain with and that hosts the name server listing the records for your domain also offers the service of providing a certificate for the domain. While you have registered ownership of the domain with them, they often still require you to pay for the services of verifying ownership and providing a certificate for the domain you registered with them.

2.4. Downgrade attacks

Downgrade attacks are a type of attack on the web where an attacker forces a connection to use a less secure version of a protocol. In the case of the web, such attacks often attempt to downgrade from HTTPS to HTTP. HTTP Strict Transport Security (HSTS) [36] is a protocol that aims to prevent such attacks. Without HSTS, the only way a client knows which protocol to use is by looking at the URL. However, for user convenience clients often allow users to specify a URL without indicating the protocol. In such cases, a client can historically exhibit varying behavior, some clients always use HTTP, while others first attempt to use HTTPS and fall back to HTTP if HTTPS is unavailable. This behavior leaves the client vulnerable to downgrade attacks.

When access to the HTTPS server is blocked, the browser will automatically attempt to reach the same website over HTTP instead. This effectively downgrades the connection from HTTPS to HTTP. Once downgraded, the attacker can intercept and manipulate the data. In case there is no HTTP server available for the website, or the HTTP server redirects back to the HTTPS server, an attacker can host their own HTTP server tricking the client and user into communicating with it instead. Even when a user does not further interact with the HTTP server, cookies which are used to store session information, are vulnerable when transferred over an unsecure HTTP connection. An attacker can capture these cookies sent when contacting the HTTP server and use them to hijack the session, gaining unauthorized access to user accounts and sensitive information.

Not all types of cookies are vulnerable. For example, cookies, for which the server explicitly states that they should only get sent over HTTPS using the *Secure* directive, do not get sent over HTTP. However, these cookies are vulnerable when a different type of downgrade attack is performed, instead of blocking access to the HTTPS server, an attacker can serve their own HTTPS server using a self-signed certificate. Most browsers will display a warning stating that the HTTPS server is untrusted and indicate that the user's security is at risk. However, users can click through this warning and choose to ignore it, potentially with the idea of being extra cautious but wanting to see if the connection is truly compromised. When a user clicks through, the browser establishes a connection to the HTTPS server and any secure cookies the browser has for the website are sent to the server of the attacker.

2.4.1. HTTP Strict Transport Security (HSTS)

HTTP Strict Transport Security (HSTS) [36] is a protocol that aims to prevent downgrade attacks. The protocol introduces a new HTTP response header labeled *Strict-Transport-Security*. Using this header, servers can indicate as part of a response to a request that any user agent connecting to them should use the HTTPS protocol with a trusted X.509 certificate rather than the HTTP protocol and should not accept an untrusted X.509 certificate. An HSTS header has two directives that are specified by the server. The first directive is *max-age*. This directive is followed by a number and specifies in milliseconds how long a user agent should remember receiving the header. The second directive, *includeSubDomains*, specifies whether a received HSTS header should also apply to the subdomains of the original domain it was received from. This directive helps ensure that all connections, including those to subdomains,

are protected from the start, ensuring that cookies specified at the domain level are secured.

HSTS headers can only be specified as part of a response to an HTTPS request and user agents should remember the receival of an HSTS header to enforce the associated rules to that domain and possibly its subdomains. To remember receiving the HSTS header, a user agent uses a special type of cache called the HSTS cache. Any time an HSTS header is sent to the user agent, an entry is added to the HSTS cache. Each entry in the cache specifies the domain name it was received from, whether it should apply to subdomains as specified by the *includeSubDomains* directive, and lastly the expiration time as calculated from the time of receival and the duration set using the *max-age* directive. Anytime an HSTS header is received, a new entry is added to the HSTS cache replacing an entry for the same domain if one was already present. Overwriting old entries in the HSTS cache, in effect, moves the expiration time of existing entries forward any time an HSTS header is received. When checking if HSTS should be enabled, the HSTS cache is searched for an existing entry. When an entry is found for the domain name, the expiration time of the entry is checked. In case the entry has expired, it is removed from the HSTS cache. If it has not expired, HSTS is enabled for the domain and connections are limited to trusted HTTPS communication. To ensure persistence across sessions, the user agent needs to save the HSTS cache to disk whenever it is modified.

2.4.2. HSTS Preload list

HSTS cache is based on previously received HSTS response headers. Therefore, HSTS is only enabled if the website has previously sent an HSTS header as part of an HTTPS response or a higher-level domain has already been visited and specified that its entry in the HSTS cache should apply to its subdomains. These limitations leave the first visit to a domain vulnerable to all attacks that HSTS aims to prevent. Additionally, the HSTS cache contains a subset of the list of domains visited by the user agent and is thus considered part of the browser history. Therefore upon deleting the browser history or parts of it, the respective HSTS entries are removed. This considerably increases the potential for an attack on such clients.

To address this issue the HSTS specification includes a section labeled HSTS Pre-Loaded List. It introduces the option for user agent vendors to include a list as part of their user agents through which website operators can pre-configure user agents with HSTS enabled for their domain(s). By pre-configuring a domain with HSTS enabled, all connections to that domain - even those before receiving any HSTS headers - are protected from the vulnerabilities addressed by HSTS.

The most commonly used instance of such a list is maintained as part of the Chromium project [97]. Known as the HSTS Preload list [98], it is adopted by most major browsers, including Chrome, Firefox, Opera, Safari, Edge, and all mobile and desktop-based derivatives of those browsers. The official way that Chromium distributes the HSTS Preload List is a JavaScript Object Notation (JSON) file as part of their open source code [1]. The JSON file consists of one object, with a single attribute labeled *"entries"*. This attribute contains a list of all the entries, where each entry is a JSON object, with the attributes *name*, *policy*, *mode*, and *include_subdomains*. The *name* attribute contains a string equal to the domain name that the entry is for. The *policy* attribute is used for the maintenance of the list and indicates the reason for including this specific entry. Options for such reasons are "public-suffix", "custom", "google", and many more. This entry is only used to provide information to the person maintaining the list. The *mode* attribute has only one possible value "force-https", but it is also possible for this attribute not to be specified. The presence of the "force-https" value indicates that this domain should enable HSTS. Entries without a *mode* attribute are used by a secondary function of the HSTS Preload List inside Chromium, which is the static binding of public keys to certain domains using their hashes. This static binding is part of the web standard known as Public Key Pinning (PKP)[28]. Most browsers including Chromium have dropped support for the PKP response header as defined in the web standard. However, Google Chrome still implements the static version of PKP to allow for the binding of certain domains and their X.509 certificates. To do this it stores the hash of the certificate and links it to the desired domain(s). Whenever the browser tries to create a connection with a domain that has a pinned certificate, it compares the hash of the received certificate against the hash of the pinned certificate. In case the hashes do not match, the connection is blocked. Additionally, in case the certificate is signed against the public root certificates, a report is sent to a specified website indicating that an illegitimate

[1] `https://source.chromium.org/chromium/chromium/src/+/main:net/http/transport_security_state_static.json`

certificate is being used for the domain. A deep understanding of PKP is not required to understand this thesis. However, Section 3.1.5 provides more information about the protocol and the mixing of static PKP with the HSTS Preload List is mentioned as part of Section 5.2.3 where we implement our solution in Chromium.

Using the HSTS Preload List Submission website [2] website operators can request to be included in the HSTS Preload List. Before accepting a submission the website first checks the requested domain to see if it fulfills five requirements. The first requirement is that the requested domain is a main domain as defined in Section 2.2.1. The second requirement is that the domain hosts an HTTPS server with an X.509 certificate signed against an official root certificate. The third requirement is that all responses sent by the HTTPS server include an HSTS header. This HSTS header is required to have an additional *preload* directive and the *includeSubDomains* directive set. Additionally, the value set for the *max-age* directive is required to be at least 31536000 seconds (1 year). The remaining two requirements are conditional requirements, meaning that they are only required if a certain condition is met. The first conditional requirement is that if the requested domain hosts an HTTP server, it should redirect all requests to the HTTPS server on the same domain. The last requirement is that if a DNS record for the "www" subdomain exists that subdomain is required to also host an HTTPS server with a trusted certificate. Requirements for addition to the HSTS Preload List have changed over time and some entries currently included in the list would not be accepted if a request was made to include them today. For example, the entry for *wordpress.com* does not have *include_subdomains* set to true, since at the time of the request the *includeSubDomains* directive was not required as part of the submission process.

When requesting an addition to the list, the HSTS Preload List Submission Website automatically checks the requirements and if successful adds an entry for the domain. The latest entries to the list that have been made in this way all have their *policy* set to "bulk-1-year", *mode* set to "force-https", and *include_subdomains* set to true. The policy of "bulk-1-year" indicates that the addition was part of the automatic entries with the requirements of *max-age* being set to a minimum of 1 year. Removal from the list can also be requested using the website. The requirement for a domain to be removed from the HSTS Preload List is that the *preload* directive is no longer specified in the HSTS header.

The latest pending additions to and removals from the HSTS Preload List are published using the HSTS Preload Submission application programming interface (API) available at `https://hstspreload.org/api/v2` under /pending and /pending-removal. Before a new Chromium version is released a maintainer updates the HSTS Preload List using these endpoints and resets them to return two empty lists.

It is also possible for owners of an eTLD to request inclusion. However, to add an eTLD the owner has to manually contact the Chromium team via email. Entries for an eTLD always have *include_subdomains* set to true. In effect, such an entry provides the benefit of the HSTS Preload list to all domains that can be registered under them and their subdomains.

A measurement the Chromium Authors take to prevent blocking valid access to some sites as the Preload data becomes outdated is to store the time at which the HSTS Preload list and static PKP entries were encoded into the browser. These dates are then used to ignore the HSTS Preload and PKP data hard-coded into the browser if they become older than 7 weeks. This successfully prevents outdated PKP entries from blocking connections to domains that have since rotated to a new X.509 certificate. Additionally, it prevents old HSTS Preload entries from blocking legitimate HTTP connections to domains that have dropped HSTS Preload support.

2.5. Browsers

Web browsers are software applications used to access and view information on the web. A browser interprets and displays web content, allowing users to navigate between web pages, interact with multimedia, and access various online services. Browsers use protocols like HTTP and HTTPS to retrieve data from web servers and render it into a readable format for users.

Browsers consist of two parts, the browser engine and the user interface. The browser engine is responsible for fetching, displaying, and allowing interaction with web pages. The user interface is

[2] `https://hstspreload.org/`

responsible for things like navigation, the address bar, bookmarks, the download manager, security indicators, and extensions.

Today there are a total of 3 well-known browser engines being developed, these are Chromium (Google Chrome), Gecko (Mozilla Firefox), and Webkit (Apple Safari). Most other browsers use (a modified version of) these engines. Of the engines Chromium is used most commonly with 77% of the market share [31].

As browsers are the most common way for users to interact with the web, this thesis primarily focuses on improving the security of the browser engine. However, the proposed adjustments can be applied to other clients used to access websites, such as command line tools.

2.5.1. Chromium

Central to Chromium's design is the multi-process architecture and the use of mojom message channels for inter-process communication (IPC). The key processes are the browser, network, gpu, and renderer processes. At most a single browser process can run at the same time. This process is responsible for creating, controlling, and coordinating all other processes. The browser process creates a single network process. This process handles all web requests. Handling network processes separately allows for sandboxing, this prevents malicious network interactions from affecting the rest of the system. The browser process also creates a single GPU process. The GPU process is a dedicated process for handling graphics, which helps in rendering complex visuals and offloading work from the CPU. In general, each new window or tab opens a new renderer process. These processes handle the content of webpages and directly communicate with the GPU process through a command buffer to display websites.

The main focus of this thesis takes place in the network process, which amongst others handles DNS, HTTP, HTTPS, and HSTS communications. Currently, the network process does not have support for handling DNSSEC validation, instead relying directly on the systems' DNS resolver to benefit from the extra security provided by DNSSEC.

3

Related Work

Previous work has been done on the subject of improving security and privacy on the internet. A plethora of work aims to increase web security in ways that are orthogonal to HSTS-Enforced, for instance, by securing certificates [2, 8, 10, 28, 35, 38, 51, 55] or by protecting against attacks such as cross-site-scripting [6, 7, 9, 15, 34, 40, 46–48, 52, 59, 75, 78, 101, 103–107]. Section 3.1 discusses some of these works aimed at increasing security on the conventional web. Section 3.2 discusses several works that increase security and privacy on the internet. Section 3.3 discusses several protocols used for secure and private messaging. This thesis takes inspiration from these works but also uses them as a baseline for the security of the new protocol.

Previous research directly focused on HSTS has primarily measured its adoption rate [3, 13, 18, 25, 32, 33, 39, 53, 54, 56, 60, 63, 67, 68, 74, 82, 83, 89, 90, 92] and identified vulnerabilities [14, 45, 57, 69, 77, 81, 84, 88, 89, 95, 100, 102]. A recent proposal extends HSTS to share data among users to prevent tracking through user-specific responses [22]. However, this proposal does not tackle HSTS's remaining vulnerabilities and is limited to websites with many distinct visitors. Section 3.4 discusses this work.

3.1. Security on the Conventional Web

3.1.1. Automatic Certificate Management Environment (ACME)

In 2014, the non-profit Let's Encrypt[1] CA started offering free certificate signing. They argue that security should be for everyone and want to encourage the widespread adoption of HTTPS. To facilitate domain ownership verification, they make use of the Automated Certificate Management Environment (ACME) [8]. When a domain owner requests a certificate, Let's Encrypt's ACME server automatically challenges them to prove they control the domain for the requested certificate. There are two types of challenges currently in use HTTP-01 and DNS-01. When the HTTP-01 challenge is used, the Let's Encrypt servers ask the requester to place a specific file with specific content at a well-known URL on the web server. Let's Encrypt then checks if it can access that file via HTTP to verify the requester's control over the domain. When checking, the Let's Encrypt servers first attempt to verify existence of the file using HTTP. If HTTP is not available or if the HTTP server redirects to HTTPS, it attempts to verify the file's existence using HTTPS. Since completing the challenge is required to obtain a valid certificate, it does not validate the certificate used to create the connection when HTTPS is used. The DNS-01 challenge requires a requester to create a specific DNS record. Let's Encrypt then checks its existence, thereby confirming domain ownership. Both these challenges rely heavily on the network security of the Let's Encrypt servers since neither DNSSEC nor HTTPS is used. To secure the HTTP-01 and DNS-01 verification, Let's Encrypt takes a lot of extra steps. One example of this is Multi-Perspective Validation [1], which ensures that a single compromised network does not cause the verification process to become compromised. This extra step uses multiple cloud servers to provide different perspectives on whether a challenge is completed, a request is then only accepted if all perspectives agree.

[1] `https://letsencrypt.org/`

3.1.2. Content Security Policy (CSP)

Content Security Policy (CSP) [105] is a web security standard aimed at improving web security. CSP is designed to mitigate various types of attacks, including Cross-Site Scripting (XSS) and data injection attacks. Deploying CSP requires the specification of a set of directives in the form of HTTP headers or using HTML elements. These directives are interpreted by the browser as a whitelist of trusted sources for content loading and execution, thereby reducing the risk of unauthorized code execution and data leakage.

Research on CSP has explored its effectiveness in enhancing web security and mitigating common vulnerabilities. Studies have investigated the impact of CSP deployment on reducing the prevalence of XSS attacks and the ability of attackers to execute malicious scripts on web pages. Several studies, such as [15, 16, 78, 104], analyze the deployment of CSP. They unanimously concluded that CSP is often wrongly configured. However, these papers also highlight the potential for CSP to significantly improve web security when properly configured.

In the context of our research, CSP serves as a complementary security mechanism that helps enforce a secure browsing environment by controlling the behavior of scripts and other resources loaded on web pages. While HSTS-Enforced primarily focuses on securing the transport layer by regulating the use of HTTPS, CSP adds an additional layer of defense by restricting the execution of potentially malicious content within the browser environment. The integration of CSP with HTTPS can further enhance the resilience of web applications against various types of cyber threats, contributing to a more robust and secure web ecosystem.

3.1.3. DNS-based Authentication of Named Entities (DANE)

Developed as an extension to DNSSEC, DNS-Based Authentication of Named Entities (DANE) [38] leverages the hierarchical structure of DNS to authenticate the association between domain names and their cryptographic keys. DANE operates by publishing X.509 certificates using Transport Layer Security Authentication (TLSA) records. These records store the hash of an X.509 certificate. Once the authenticity of such records is verified using DNSSEC they can then be used to validate X.509 certificates presented by servers during the establishment of secure connections using SSL/TLS. By comparing the cryptographic information retrieved using DNS with the certificate presented by the server, the client can verify the authenticity of the server's certificate using DNSSEC in addition to verification using the certificate chain. This offers additional protection in case a high-level CA is compromised.

One of the significant advantages of DANE is its ability to provide domain owners with greater control over the validity of X.509 certificates. Since DANE records are managed alongside DNSSEC records, domain owners can invalidate cryptographic keys and certificate information autonomously, without requiring the involvement of third-party CAs. This decentralization of trust empowers domain owners to promptly respond to security incidents, such as compromised private keys or fraudulent certificates, by revoking and replacing affected keys or certificates in real time.

While DANE offers promising security benefits, its widespread adoption faces challenges related to DNSSEC adoption and deployment complexity. Moreover, the diverse ecosystem of DNS software and implementations introduces interoperability issues that may hinder the seamless integration of DANE into existing systems. Despite these challenges, ongoing efforts to standardize and refine DANE specifications aim to overcome barriers to adoption and promote its role as a complement or even a viable alternative to the traditional CA model.

3.1.4. Certificate Authority Authorization (CAA)

Certificate Authority Authorization (CAA) DNS records are an integral component of the DNS infrastructure, designed to enhance the security of the X.509 certificates. The introduction of CAA records is a response to growing concerns about unauthorized issuance of digital certificates, which can lead to significant security vulnerabilities such as man-in-the-middle attacks.

The concept of CAA records emerged from the need to provide domain owners with greater control over which CAs are permitted to issue certificates for their domains. Before to the adoption of CAA records, any CA trusted by browsers and operating systems could issue a certificate for any domain, posing substantial risks if a CA was compromised or mistakenly issued a certificate.

A CAA record allows a domain owner to specify which CAs are authorized to issue certificates for their domain, thus adding a layer of security and trust. Once specified a CAA record authorizes a particular CA to distribute certificates for it. An example of a CAA record is *example.com. IN CAA 0 issue "letsencrypt.org"*, authorizes Let's Encrypt to issue certificates for *example.com*.

3.1.5. Public Key Pinning (PKP)

PKP [28] is a security measure that aims to enhance the trustworthiness of SSL/TLS connections by associating a specific public key with a particular server or domain. PKP specifically addresses the threat of compromised CAs by ensuring that only specified public keys are trusted for a domain. This method prevents man-in-the-middle attacks that exploit fraudulent certificates issued by malicious or compromised CAs. While PKP provides an additional layer of security by limiting the trust to a subset of keys, it requires careful management to avoid issues such as service outages due to key mismanagement or expiration. Both HSTS and PKP contribute to the robust security posture of modern web applications by addressing different aspects of the HTTPS security landscape.

3.2. Anonymous and Private Network Protocols

3.2.1. Lightweb

Lightweb [21] is an alternative to the conventional web, which uses a relatively new protocol to allow for fully anonymous browsing. The protocol is called the Zero-Leakage Transfer Protocol (ZLTP). When using ZLTP a server serves fixed size-pages of data. There are two modes of operation in which a server can provide access using ZLTP. The first mode of operation uses private information retrieval methods. This enables a client to fetch a key-value pair while hiding all information from the server about which pair the client fetched. To enable a fast response time, the authors use a fast well-known information retrieval method. A downside to this method is that it requires the client to communicate with two servers to request a page. The second mode of operation uses hardware enclaves and obfuscation of the memory used by the server to directly obscure the request information from the server. Hardware enclaves are secure and isolated regions within a processor's architecture that safeguard sensitive data and computations from unauthorized access or tampering. This decreases the computational cost and improves the response rate. However, there are several attacks possible on hardware enclaves, which expose weaknesses in the ZLTP protocol if used. Therefore, the authors suggest to use the first mode of operation if security is needed. By using the ZLTP protocol a client can browse fixed-size Lightweb pages, without revealing to anyone which pages it is reading. This includes both network interceptors and the servers hosting the pages.

Due to limitations of the ZLTP protocol, the Lightweb only allows servers to serve static content. Due to the inability to serve dynamically generated content, the Lightweb does not allow for the authentication of users on the server side. To provide authentication the authors suggest the use of public-key encryption to restrict access to certain pages. By storing the encrypted page on the server, and handing out the public key only to authenticated users, the access to pages is limited to authenticated users. For the distribution of these keys, the authors suggest using the conventional web. Thus, to securely facilitate authenticated requests, the Lightweb still relies heavily on the security of the conventional web.

3.2.2. Onion Routing

Onion routing [24] is a privacy-preserving communication technique used to enhance anonymity and security on the internet. It operates by obscuring the origin and destination of data packets as they traverse through a network of intermediary nodes, called onion routers. Each node provides a layer of encryption to conceal the path and content of messages. This encapsulation using multiple layers of encryption is akin to the layers of an onion, hence the name. Each layer is encrypted such that only the designated intermediary node can decrypt its respective layer. This layering ensures that no single node in the network knows both the source and destination of the data. Thus, it thwarts attempts to trace the communication back to its origin.

Upon receiving an encrypted message, each onion router sequentially peels off a layer of encryption, revealing the address of the subsequent node to which the message should be forwarded. This process continues until the message reaches its final destination, at which point the outermost layer of encryption is removed, and the plaintext message is read. Throughout this journey, the identity and location of the

sender remain concealed from all intermediate nodes, preserving anonymity.

In comparison to HTTPS, onion routing additionally provides resistance to traffic analysis attacks. Since messages are relayed through multiple nodes in the network, adversaries attempting to monitor communication patterns face significant challenges in correlating data flows with their corresponding endpoints. However, if an attacker controls all intermediate nodes or can coerce them into revealing sensitive information, the anonymization of the data flow is still compromised. Additionally, the overhead associated with encrypting and decrypting multiple layers of data can introduce latency and overhead, impacting the performance of applications.

Nonetheless, onion routing remains a cornerstone of modern privacy-enhancing technologies, forming the basis for popular anonymity networks such as Tor (The Onion Router). By providing a robust framework for anonymous communication, onion routing continues to play a crucial role in safeguarding privacy and freedom of expression on the web.

3.2.3. Garlic Routing

Garlic routing [99] is a variant of onion routing designed to improve efficiency and scalability. Instead of each node in the network peeling off only one layer of encryption as in onion routing, in garlic routing, multiple layers of encryption are bundled together. These bundled layers of encryption are called cloves. Each node in the garlic routing network removes one such clove and then forwards the content as a bundle. This bundling of data allows for more efficient use of network resources, reducing the overhead associated with individual onion layers.

Garlic routing is generally considered more efficient than onion routing because it reduces the number of individual encryption and decryption operations required per message. With garlic routing, nodes can process multiple messages in a single operation, leading to lower latency and higher throughput compared to onion routing. This increased efficiency of garlic routing does not compromise security, it is designed to maintain the same level of anonymity as onion routing. Additionally, garlic routing offers more flexibility in terms of how messages are bundled and processed compared to onion routing. For example, garlic routing allows for more sophisticated routing strategies, such as mixing multiple messages together to further obscure traffic patterns. The Invisible Internet Project (I2P) uses the garlic routing protocol to provide a secure and private browsing experience for its users.

3.2.4. Mixed Networks

Mixed networks [96], also known as mix networks or mixnets, operate by routing messages through a series of intermediate nodes called mixes or mix nodes. Each mix node re-encodes or mixes the messages it receives before passing them on to the next node, making it difficult to trace the original sender and receiver. The mixing process typically involves encryption and reordering of messages to break the link between input and output. This reordering and encryption of the messages hides additional information about the data being communicated, such as timing and size. Mixed networks differ from onion routing as they introduce additional delays by holding messages for a certain amount of time before forwarding them, further complicating traffic analysis.

3.2.5. Forward Proxies

A forward proxy can act as an intermediary between clients and servers on the internet. Any messages from the client to the server and vise versa are sent through the forward proxy. The goal of forward proxies is to facilitate private browsing. By using a forward proxy, the server never sees the client's address. Additionally, forward proxies can provide functionalities such as caching, content filtering, and load balancing.

To provide additional anonymity multiple forward proxies can be chained together into a ProxyChain. Each proxy in the chain forwards the traffic to the next proxy in the chain. This makes it more difficult for outside parties to trace the origin and destination of the data.

A forward proxy can optionally enforce the use of secure communication channels between the client and the proxy server. However, it does not enforce a secure connection between the proxy server and the server that the client is requesting data from. Instead, securing the communication between the client and the destination server relies on the default protocols available on the web such as HSTS and

HSTS preloading. Additionally, the forward proxy is in a position to intercept any data sent through it. Thus, while aimed at anonymizing communication, the use of a forward proxy has a downside. Using a forward proxy places inherent trust in the proxy, since it is placed in the perfect position to perform various attacks. This attack vector is enlarged by the vulnerabilities present in HSTS and the low adoption of HSTS preloading.

3.2.6. Virtual Private Network (VPN)

Virtual Private Networks (VPNs) [58] are also aimed at securing and privatizing internet traffic. Over the last few years, VPNs have become very popular. VPNs are very similar to forward proxies. However, a VPN works for all network traffic and not only web traffic. Commonly, VPNs are also used to provide access to devices that are generally only accessible from the network on which the VPN server is hosted.

However, VPNs have the same weaknesses as forward proxies. Similarly to forward proxies, a malicious VPN can intercept and potentially tamper with any data sent through it.

3.3. Private messaging protocols

Private messaging protocols provide a secure and private way of communication across the internet. These protocols do not rely directly on the use of HTTPS. Instead, these protocols often favor more advanced methods of providing secure and private communication. In recent years, private messaging platforms have garnered significant attention and for most people, they have become part of their daily lives. This section discusses several of these protocols and how they facilitate secure and private communication.

3.3.1. Pretty Good Privacy (PGP)

Pretty Good Privacy (PGP) [30] defines data encryption and decryption methods that provide cryptographic privacy and authentication for data communication. It was originally meant for securing mail services but has been adapted to other messaging services. To provide secure communication PGP uses a single symmetric key to encrypt a message. This symmetric key is different for each message. To securely communicate the symmetric key it is encrypted using two asymmetric key pairs. One of these pairs is associated with the sender and the other with the receiver. The encrypted symmetric key is attached to the symmetrically encrypted message. The use of the asymmetric keys allows users to authenticate each other using each other's public keys. However, its security is limited as a user always authenticates using the same asymmetric key pair.

PGP is quite similar to the encryption used to facilitate HTTPS. It offers the same level of security. However, due to the binding of the same asymmetric key to a user involved in multiple conversations, it can be considered cryptographically weaker than an HTTPS connection created using two asymmetric key pairs, one of which is randomly generated by the client for a specific connection.

3.3.2. Off-the-Record (OTR)

Off-the-Record (OTR) [12] is a protocol designed to provide encryption for instant messaging conversations. The OTR protocol uses a combination of the AES symmetric-key algorithm with 128 bits key length, the Diffie–Hellman key exchange with 1536 bits group size, and the Secure Hash Algorithm (SHA)-1 hash function. Despite its age, OTR remains relevant due to its strong security guarantees and ease of implementation.

Compared to HTTPS standards the OTR protocol provides additional benefits, such as deniable authentication, which certainly have their use-cases in private messaging systems. Deniable authentication means that both parties involved in an OTR message exchange can confirm the origin of a message, while at the same time, any third parties cannot do the same. However, this additional benefit is generally not required for the exchange of documents or resources on the web.

3.3.3. Signal Protocol

The Signal Protocol [86], utilized by the Signal messaging app [87], is renowned for its end-to-end encryption mechanism, which guarantees that only the communicating users can read the messages. This protocol employs forward secrecy, ensuring that even if the encryption keys are compromised, past

communications remain secure. To do this it uses a combination of several cryptographic algorithms. These algorithms are all combinations of simpler cryptographic protocols. One of these algorithms is Extended Triple Diffie–Hellman (X3DH) key agreement. X3DH is a modification of Diffie-Hellman, where the key exchange process is repeated three times, hence the "triple" in its name. This allows for the authentication of both parties while generating a unique key for each conversation. This unique key is then used to create a key generator at each of the parties. Through highly mathematical operations these key generators allow for the generation of the halves of a key pair at each party. The keys these generators produce are then used to encrypt and decrypt the messages sent between both parties. Signal's commitment to privacy and open-source development has made it a benchmark for secure messaging protocols.

Applying the cryptographic standard introduced by the Signal Protocol to the web would be possible. However, the additional overhead induced by the X3DH key agreement and the use of a shared key generator would likely induce cryptographic overhead for which the resources are not currently available as some web servers handle a lot of connections to different clients.

3.4. CoStricTor

Due to its potential for user tracking, Tor Browser typically disables HSTS, prioritizing user anonymity. CoStricTor [22] addresses this issue by leveraging a crowdsourcing mechanism that allows Tor Browser users to share HSTS data in a secure and privacy-preserving manner.

At its core, the HSTS headers are still requested directly from websites. However, CoStricTor shares the headers received by a single client with the others. The protocol uses Bloom filters to anonymously and efficiently share this data. Bloom filters are a probabilistic data structure that uses hashing algorithms. Bloom filters use these algorithms to create a very compact representation of a set of elements. Despite the authors efforts to minimize them, Bloom filters inherently have a non-zero false positive rate. This means that legitimate connections may occasionally be incorrectly blocked due to the probabilistic nature of the filter. By carefully tuning the parameters of the Bloom filter, the protocol can strike a balance between memory efficiency and accuracy. However, not all false positives can be prevented, which the authors accept as HSTS wrongfully being enabled results in safer behavior than not using HSTS at all. To show the correct working of the protocol the authors analyze it on 150,000 websites of which 10,000 have HSTS enabled.

While CoStricTor increases the anonymity of HSTS, its approach has several downsides and potential issues. To prevent false entries into the bloom filter the protocol depends on a high number of users visiting a single website, requesting its HSTS headers, and sending them to their CoStricTor instance. In case the number of visitors to a domain is low, a single attacker can easily insert a false entry. However, given enough resources an attacker can always insert false entries. As the number of websites on the web is much larger than 150,000 websites, the parameters of the Bloom filter need to be adjusted to keep the number of false negatives to a minimum. This can drastically affect the real performance of the protocol. Additionally, as the percentage of websites that have adopted HSTS increases, the chance increase for a website that does not support HSTS to be incorrectly identified as having HSTS enabled. Finally, CoStricTor does not address the low adoption of HSTS, low awareness of its existence by website operators, and misconfigured HSTS headers leading to diminished security.

4

Problem

There are several issues present in the existing protection against downgrade attacks, in the first section of this chapter, we outline these issues. In the second section, we put forward the requirements that the solution to the problem outlined in this thesis should meet.

4.1. Issues with HSTS

4.1.1. Header Misconfigurations

Misconfigurations of HSTS headers happen relatively often [32, 57, 85]. These misconfigurations range from misplaced characters and negative *max-age* values to sending the headers only over HTTP. Operators of these domains might falsely expect HSTS to be enabled. However, browsers that receive such misconfigured HSTS headers do not enter the domains into the HSTS cache. Negative *max-age* values even actively remove any existing entries for the domain. The effect of a misconfigured HSTS header is that HSTS is disabled for a certain domain.

4.1.2. State-Based

For HSTS headers to prevent unsecure connections, the browser must have previously received the header from the HTTPS server and stored it in the HSTS cache. This means that the HSTS cache stores a history of which websites have been previously visited using HTTPS and provided HSTS headers. Therefore, a history deletion in the browser using the "clear all history" window includes clearing the HSTS cache. Thus, the first connections made to websites after clearing a user agent's history do not prevent unsecure connections.

4.1.3. Cross-Network & Cross-Session Tracking

As long as the browser history is not deleted, the HSTS cache can be used to track devices across different networks and sessions [95]. Using several subdomains a server can establish a distinct identifier for devices using HSTS as a binary encoding. By setting HSTS entries for different subsets of these domains for each new visitor, they become uniquely identifiable. On subsequent visits, the server can extract the identifier by prompting the browser to request the full range of subdomains via HTTP. HSTS then causes the user agent to visit the subset of subdomains for which HSTS entries were set over HTTPS instead of HTTP. The server can then transform the pattern of subdomains visited via HTTP and HTTPS back into the binary identifier. This is one of the reasons that the Tor browser does not support HSTS. As discussed in 3.4, attempts have been made to prevent this tracking by introducing an extension of HSTS which uses collaborative information sharing to automatically share HSTS entries amongst users, preventing servers from setting client-specific entries [22]. However, the effectiveness of this extension is limited to domains that have a relatively high amount of frequent visitors.

4.1.4. Low adoption of Preload Status

HSTS Preloading has a relatively low adoption rate, especially websites that should use it, such as government websites and banks have historically shown to lack support [3, 4, 53, 54, 74, 90]

The lack of adoption of HSTS preloading is not entirely known, but the consensus is that the low adoption rate is directly caused by two factors. Firstly, some developers do not know that HSTS preloading exists. The second cause is slightly more complicated. It is a direct result of the hard requirement of HSTS preloading to include subdomains. A historically common practice is to use subdomains for additional services. When these services are only internally accessible, they are often served over HTTP. The use of such services causes many websites to not enlist for HSTS preloading as it would require them to upgrade all these services to use HTTPS. Although this seems to be a simple fix, large organizations have often applied this practice for years. They often have undocumented services used by a small subset of the employees. This situation causes these large organizations, to often not register for HSTS preloading. However, these organizations often have the largest user bases. Therefore, opting into HSTS preloading is often even more crucial for these organizations.

4.1.5. Hard to Preload

As discussed in 2.4.2 there are a lot of requirements before a website operator can successfully request an addition to the HSTS Preload list. Additionally, inclusion on the Preload List enforces HSTS for all subdomains. These requirements and restrictions make it hard for website operators to adopt HSTS preloading.

4.1.6. Time-Based Validity

HSTS headers have an expiration date as indicated by the *max-age* directive. To prevent unwanted blocking of connections, browsers additionally expire the HSTS Preload List. For example, Chromium releases an update every six weeks, therefore they can expire the Preload List after six weeks (with a grace period), as by then users will have likely updated to the new version. By forwarding the system time, an attacker can exploit this behavior, causing the browser to clear all entries in the HSTS cache and ignore the HSTS Preload List. Using Network Time Protocol (NTP) [64], such an attack is shown to be possible [84].

4.2. Requirements

This section lists the requirements that the solution to the problem highlighted in this thesis should fulfill.

4.2.1. Maximum Security

The solution should provide full protection against downgrade attacks from attackers who do not have full access to the system. This requirement addresses the issues highlighted in Sections 4.1.2 and 4.1.6.

4.2.2. Minimum Configuration

To ensure a high level of adoption of this new maximum level of protection, requirements and configuration steps for users and website owners to protect against all vulnerabilities should be kept to a minimum. This requirement addresses the issues highlighted in Sections 4.1.4 and 4.1.5.

4.2.3. Protect against Misconfiguration

To prevent misconfigurations from exposing vulnerabilities, any misconfiguration should not degrade the protection against downgrade attacks. This requirement addresses the issue highlighted in Section 4.1.1.

4.2.4. No Tracking

Tracking of user agents as currently present in the HSTS protocol should be prevented. At the same time, the solution should not introduce new methods of tracking the user agent. This requirement addresses the issue highlighted in Section 4.1.3.

4.2.5. No Strict Enforcement

Due to some older servers having limited performance in cryptographic operations or the unwillingness of some website operators to acquire an X.509 certificate, there should be a way for operators to use unsecure communication protocols such as HTTP when required.

4.2.6. Exceptions

A selection of domains is reserved by the Internet Assigned Numbers Authority (IANA) for policy or technical reasons [43]. Since domains on this list are not available for public registration, they will never be released to a registrar and no publicly trusted X.509 certificate can be received for them or their subdomains. Additionally, publicly trusted X.509 certificates can not be requested for any IP address. Therefore, it is impossible to establish a valid HTTPS connection to websites under these hostnames without adding self-signed certificates to the list of root certificates. To not block all traffic to servers using these domains or IP addresses by default, connections using these hostnames should be allowed to use HTTP. However, by adding self-signed certificates to the list of root certificates on a device and using these to sign an X.509 certificate for a reserved domain name or IP address, a trusted HTTPS connection can be established to servers using them. Therefore, users should be able to re-enable the maximum level of protection offered by HSTS for these reserved domains and all IP addresses.

4.2.7. Internal Use

Lastly, the public visibility of HTTP being used for internal purposes could indicate to an attacker that there is a vulnerability in the system. Therefore, there should be an option for website operators to hide HTTP being used on internal networks from the outside network.

5 Solution

In this chapter, we present HSTS-Enforced, our solution to the aforementioned problem. We describe how we conceptualized its design, and translated this design into a working implementation. This chapter serves as a roadmap for readers to understand the thought process behind the solution's development and to provide guidance during its further practical realization.

5.1. Design

The initial step in designing HSTS-Enforced is to establish HTTPS as the sole communication protocol for the web by default. In other words, by default, we disable the use of the HTTP protocol from within user agents. Additionally, we limit the use of HTTPS to only secure HTTPS connections, meaning that any certificate used to create HTTPS connections should be verified against the root certificates as discussed in Section 2.3.1. This step is very similar to **enabling HSTS for all websites** and directly disables all possibilities for a downgrade attack since any attempt at downgrading an HTTPS connection as described in Section 2.4 are effectively reduced to a denial of service attack instead, which is an attack the attacker could already perform. This step, in effect, correctly prevents any data from being leaked through the use of unsecure communication channels.

Disabling the use of unsecure connections by default allows us to **remove HSTS header processing, HSTS cache, and HSTS preloading** without losing any of their security benefits. Removing all these features has three objectives. Firstly, by removing HSTS header processing, HSTS cache, and HSTS preloading, we also remove their issues. By effectively enabling HSTS for all connections, we ensure full protection against downgrade attacks without requiring any configuration from both the user and website operator, thereby, fulfilling the requirements detailed in Sections 4.2.1 to 4.2.3. Secondly, since we no longer require a state to be stored, we have removed the tracking capability inherent to cache-based HSTS. Therefore, given we do not introduce new tracking methods, we have fulfilled the requirement detailed in Section 4.2.4. Lastly, we create a clean starting point from which a new protocol can be implemented to fulfill the remaining requirements.

To fulfill the requirement detailed in Section 4.2.5, we need to allow connections to certain websites using unsecure connection protocols such as HTTP when required. To do this, we introduce HTTP-Required indicators as part of HSTS-Enforced. Using the HTTP-Required indicators, website operators can indicate that their domain should not have HSTS enabled and thus their website should be accessible over HTTP. Before a user agent allows the use of the HTTP protocol and disables HSTS, it should always verify that a valid HTTP-Required indicator is present. Before an HTTP-Required indicator is deemed valid, it should be verified to originate from the website operators of the domain they apply to. Additionally, we specify the order in which the user agent checks the various types of HTTP-Required indicators as this allows us to minimize the load it puts on the DNS network while keeping the time taken to resolve indicators to a minimum. In case a user agent cannot validate the legitimacy of an indicator, such indicators should be deemed as non-existent.

As part of HSTS-Enforced, we provide two HTTP-Required indicators for website operators to indicate

an opt-out of having HSTS enabled and thus indicate they desire user agents to not block the use of HTTP. These two methods are the DNSSEC-verified *HTTPREQ* DNS record and the HTTP-Required Preload List. In the following two sections, we elaborate on the working of these indicators. In the two sections thereafter we provide information on two additional specifications that are a part of HSTS-Enforced. Section 5.1.3 describes a change in the way in which connections are instantiated and HSTS status is checked. Section 5.1.4 describes the addition of an interface aimed to improve the accessibility of setup panels provided by devices on the local network.

5.1.1. DNSSEC-validated HTTPREQ record

Using the DNSSEC system is the recommended way to indicate that HTTP is required to access a website. By specifying a DNS record of type *HTTPREQ* with a valid DNSSEC signature, a website can indicate through a verifiable channel that it allows clients to use the HTTP protocol to connect to their domain. User agents can validate the origin of an *HTTPREQ* DNS record through the DNSSEC key-chain as explained in Section 2.2.4. An *HTTPREQ* record should only be deemed valid if the corresponding RRSIG record is validated. This validation should be performed locally, as otherwise, the recursive DNS server can fake an indicator by spoofing the security flag. Since negative responses for an *HTTPREQ* record are treated the same regardless of their DNSSEC-validation status, we skip DNSSEC-validation of negative responses to reduce the load of validating *HTTPREQ* records.

Specifying an *HTTPREQ* record is done as follows. Similarly to other DNS records, the domain should be specified, CLASS should be set to "IN", TYPE should be set to "HTTPREQ", and Resource Data (RDATA) can be left empty. For example, to set an *HTTPREQ* record for "internal.example.com" using a bind style configuration file for the authoritative server of "example.com", we add the line "internal IN HTTPREQ". In case we also want to include all subdomains of "internal.example.com", we add a second line "*.internal IN HTTPREQ". After specifying the records, the DNS software can statically sign these records by generating their corresponding RRSIG records. By using the built-in wildcard to specify that all subdomains are included, a user agent can use a single DNSSEC request to check the presence of an *HTTPREQ* record for a particular domain.

While the *HTTPREQ* record does not require any RDATA, to allow for the possibility to extend the *HTTPREQ* record any RDATA contained in a response should still be forwarded by DNS servers. In case the protocol is extended, this ensures less effort is required to implement the required changes.

Hosting a recursive DNS server with restricted availability and linking a domain to an authoritative DNS server with the same restricted availability allows operators to specify an *HTTPREQ* record that is only available to people using that recursive DNS server. This allows the use of HTTP for specific local domains without directly exposing this information to the rest of the internet, this directly fulfills the requirement outlined in Section 4.2.7.

5.1.2. HTTP-Required Preload List

The HTTP-Required Preload List is a JSON file that contains domains that have indicated that they should be accessible over HTTP. An example of this list can be seen in Appendix A. This list is very similar to the HSTS Preload List, except entries on the HTTP-Required Preload List explicitly allow the use of HTTP for cases in which HTTP is required. Every entry on the list has three fields *"name"*, *"include_subdomains"*, and *"policy"*. The *"name"* field contains a direct reference to the domain name for which a HTTP-Required Preload List entry is present. The *"include_subdomains"* contains a boolean value. If true it indicates that an entry should also indirectly allow HTTP to be used on all subdomains of the domain specified in the *"name"* field and is done to keep the size of the HTTP-Required Preload List to a minimum. The *"policy"* field is used as an indication for how an entry got appended to the list. This field has three possible values: "evaluation", "reserved", and "requested". These denote the different reasons as to why an entry is included on the list.

Entries marked as "evaluation" are used to verify the correct working of HSTS-Enforced. These entries are discussed as part of Section 6.1.

To guarantee the accessibility of reserved entries as discussed in Section 4.2.6, these domains are entered on the HTTP-Required Preload List. To distinguish them from other entries on the list, they are marked with the *"reserved"* policy. To guarantee full accessibility of subdomains of these reserved domains, they

all have a true value for the *"include_subdomains"* field. An exception to this is "example.com", which is not included as "reserved" but instead used as part of the evaluation. All user agents should provide an option to ignore the entries marked as "reserved", in order to allow users to re-enable HSTS for these domains.

The secondary option for the policy field is *"requested"*. This policy value indicates that an entry was requested by the operator for this particular domain. To register a website for the HTTP-Required Preload List anyone can request the addition of a domain. Requesting addition to the list is done through the HTTP-Required Registration Website. Upon receiving a request, verification that the website operators responsible for the domain indeed wants the domain to be included in the list is done by validating that an HTTP *"HTTP-Required"* header is being sent by the corresponding server. This HTTP header indicates to the registration site that the relevant domain is to be included in the HTTP-Required Preload List. An *"HTTP-Required"* header can have two possible values, "includeSubdomains; preload" or simply "preload". When the value is "includeSubdomains; preload" this indicates that an HTTP-Required Preload List entry should apply to subdomains as well. Similarly to HTTP-01, which we discuss in Section 3.1.1, the security of the underlying network that the HTTP-Required Registration Website uses to validate the *"HTTP-Required"* header is vital to ensure the security of the HTTP-Required Preload List. Therefore, all security measures that Let's Encrypt takes to ensure HTTP-01 verification is secure should be applied to the HTTP-Required Registration Website verification.

To limit the size of the list, we limit the domains for which an addition can be requested. Any domain that can deploy DNSSEC should use the DNSSEC-based HTTP-Required indicator instead. This means that if the DNS server responsible for the eTLD supports DNSSEC we do not accept requests for the underlying domains.

Similarly to the HSTS Preload List, the JSON file generated from the database of the HTTP-Required Registration Website is statically encoded into user agents. To keep the search time taken for an entry to a minimum a data structure that allows for fast look-ups should be used. Combined with an encoding algorithm the size of such data structure can be further reduced. Examples of a data structure and encoding are provided in Section 5.2.3.

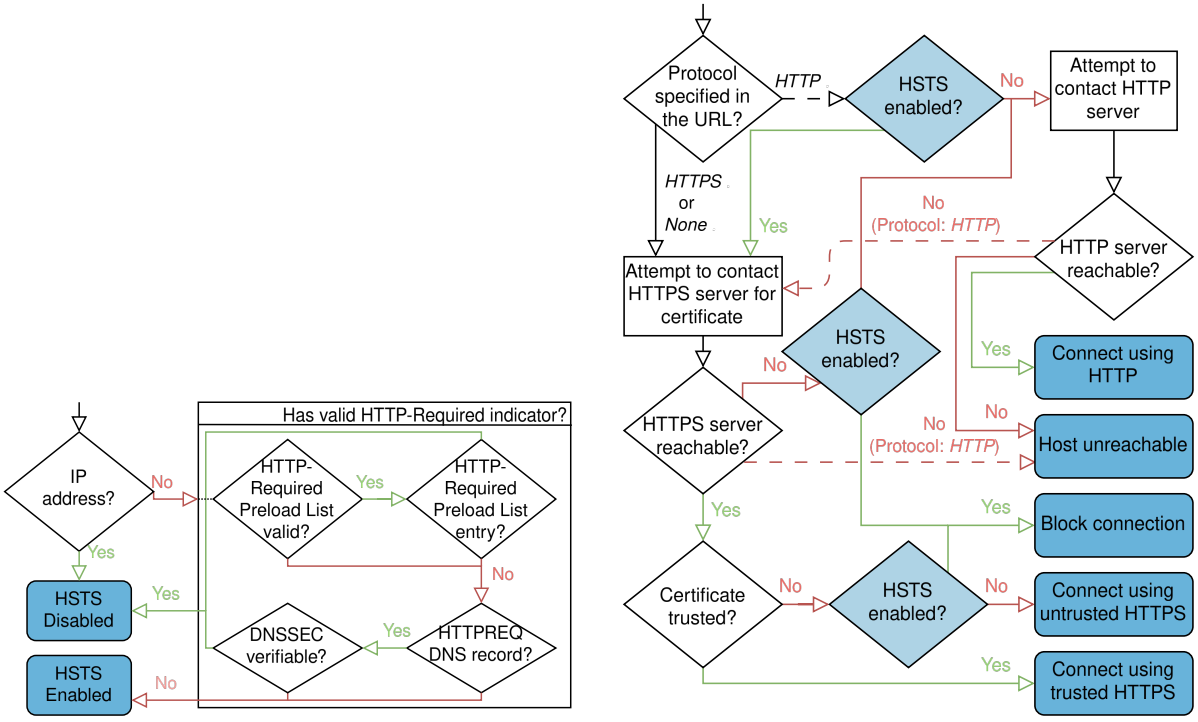
To decrease the risk inquired by using an old HTTP-Required Preload List, a timestamp is included with each new version of the HTTP-Required Preload List. The recommended duration for which the list should be deemed valid is 7 weeks. Each version of the list contains an expiration time and date, this time indicates when all entries on the list should be deemed as invalid HTTP-Required indicators. This protects users that do not frequently update their browser against downgrade attacks on domains that were previously on the list. Every 6 weeks a new version of the HTTP-Required Preload List is generated and published. This is around the same frequency as the current HSTS List is currently being updated by the Chromium team. Before publishing a new version, the HTTP-Required Preload List Website first checks all existing entries. If an entry does no longer fulfill the previously stated requirements it is removed from the list. This protects against downgrade attacks on domains that have recently changed ownership or simply no longer require HTTP in order to be accessed. Additionally, the list is minimized such that no domain is covered by two entries, for example through an entry for a parent domain that has the *include_subdomains* directive.

5.1.3. Connection Process

When HSTS-Enforced creates connections from the user agent to a website, HSTS-Enforced ensures the following things.

Firstly, HSTS-Enforced only allows unsecure connections if HSTS is disabled. HSTS is disabled if a valid HTTP-Required indicator is present, and it is always disabled when accessing a domain directly using an IP address because publicly signed certificates are only distributed for domain names and not for IP addresses. Since non-publicly signed certificates can be created for IP addresses by extending the root certificates, trusted HTTPS connections can be created to IP addresses. Therefore, user agents should provide an option to disable this bypass of HSTS for IP addresses. Together with the "reserved" entries of the HTTP-Required Preload List this fulfills the requirement highlighted in Section 4.2.6.

Second, HSTS-Enforced ensures minimum checks to the HSTS state are made. This change is included as every check to the HSTS status requires checking HTTP-Required indicators, which induces a small



(a) The process of checking whether HSTS is enabled for a domain with HSTS-Enforced.

(b) The connection process with HSTS-Enforced. Nodes labeled "HSTS enabled?" perform the process shown in Figure 5.1a.

Figure 5.1: The connection process in HSTS-Enforced

load on the network. As it is simply not necessary to check the HSTS status before attempting to create a secure HTTPS connection, we delay the checking until after a potential certificate error is detected or when an HTTP connection is attempted. This ensures the network load induced by HSTS-Enforced is kept to a minimum.

Lastly, to increase accessibility, when HSTS is disabled we allow the user agent to switch between HTTP and HTTPS when either is unavailable. Since the website operator has specifically indicated that the website should be accessible over HTTP, we can automatically switch the protocol to HTTP if HTTPS is unavailable. This differs from existing behavior, which commonly only switches between HTTP and HTTPS in two situations. The first situation is to prevent HTTP connections when HSTS is enabled. The second situation is when falling back to HTTP if a connection defaulted to using HTTPS because the URL did not specify the required protocol.

The process of deciding which type of connection should be allowed if any depends on which protocol is specified in the URL: *HTTP*, *HTTPS*, or *no protocol*. Figure 5.1b details the decision process.

When *no protocol* or *HTTPS* is specified, HSTS-Enforced first attempts an HTTPS connection. If this HTTPS connection fails because the certificate is not trusted, HSTS-Enforced checks whether HSTS is enabled or not and allows the untrusted HTTPS connection only if HSTS is disabled. If the HTTPS server cannot be reached, HSTS-Enforced checks whether HSTS is enabled or not and attempts an HTTP connection only if HSTS is disabled. In either case, if HSTS is enabled, HSTS-Enforced blocks the connection. In the second case, if the HTTP server cannot be reached, HSTS-Enforced displays that the website is unreachable.

When *HTTP* is specified, HSTS-Enforced directly checks whether HSTS is enabled or not and attempts an HTTP connection only if HSTS is disabled. If HSTS is enabled or it is disabled but the HTTP server cannot be reached, HSTS-Enforced attempts an HTTPS connection. If this HTTPS connection fails because the certificate is not trusted and HSTS is enabled, HSTS-Enforced blocks the connection. If HSTS is disabled, HSTS-Enforced allows the untrusted HTTPS connection. If the HTTPS server cannot be reached, HSTS-Enforced displays that the website is unreachable.

To reduce the latency induced by checking HTTP-Required indicators, we use a simple order of checking the presence of the indicators. By first checking the HTTP-Required Preload List, we can skip the time taken to resolve DNS records if an entry has been found on the list which can be searched relatively quickly.

5.1.4. Local Device Setup Overview Panel

Local routing devices often use domain names under public top-level domains to simplify access to their setup panels. These setup panels do not support the HTTPS protocol. However, these panels can also be accessed using the direct IP address of the devices. To facilitate ease-of-configuration for these devices we introduce an internal browser-specific page that offers an overview of such devices. The browser scans for devices that offer such panels over the HTTP protocol after which it displays a list containing all devices found. This approach offers a relatively easy-to-access interface for users to configure local routing devices. Section 5.2.3 provides an example of how such interfaces can be created.

5.2. Implementation

In this section, we detail our implementation, which encompasses various essential components for HSTS-Enforced. This includes an implementation of the HTTP-Required Registration Website, an extension of the DNS system with the *HTTPREQ* record, and an implementation of HSTS-Enforced in Chromium. The full implementation is published in the form of a Git repository, available at `https://github.com/AaronvDiepen/HSTSEnforced`.

5.2.1. HTTP-Required Registration Website

The full implementation for the HTTP-Required Registration Website, including the REST-API and user interface was created using Rust. To facilitate the development of the server we use the axum framework[1]. To store the entries entered on the Registration Website we use an underlying SQLite3 database. For future proofing, sqlx[2] was used to interface with the database, as it allows for easy replacement of the database with other more scalable SQL-based databases.

To gather information about DNSSEC-enabled eTLDs, a combination of the trust-dns-resolver[3] and publicsuffix[4] crates is used. Before the addition of a domain to the HTTP-Required Preload List, the eTLD, which can be extracted using the publicsuffix crate, is checked for its DNSSEC-capability using the trust-dns-resolver. To reduce the amount of DNS requests required, this result is cached in the database. Every 6 weeks all entries on the HTTP-Required Preload List are checked for their DNSSEC-capability. At the same time, the cached data about DNSSEC-capability is cleared so that newly DNSSEC-capable domains can be removed and informed about the changes before the next release of the list.

To facilitate access to the HTTP-Required Preload List information and easy submission of new domains, we provide access through the API. The API has several endpoints such as */preloadable*, */submit*, */status*, and */list*. Respectively, these endpoints: provide information about whether a domain can be preloaded, allow requesting an addition to the list, provide information about the current status of a domain, and allow fetching the different versions of the list. In order to successfully integrate the API into projects that depend on it, the website provides the full API specification. Additionally, to facilitate updates of user agents, */list/latest* always redirects to the latest version of the list. We publish two different styles of each list, one with additional comments describing the list, similarly to the HSTS Preload List, and one which is as small as possible, examples of these lists can be seen in Appendices A and B. Screenshots of the HTTP-Required Registration Website and its API can be seen in Appendices C and D.

5.2.2. DNSSEC Implementation

To allow testing of the *HTTPREQ* record, we need to extend existing DNS servers with support for the *HTTPREQ* record. We create two DNS server implementations: BIND9 [44] and PowerDNS [76]. By offering implementations in both BIND9 and PowerDNS, we address diverse requirements and preferences within the DNS management ecosystem. BIND9, with its robust features, is widely adopted

[1] `https://github.com/tokio-rs/axum`
[2] `https://github.com/launchbadge/sqlx`
[3] `https://github.com/msoxzw/trust-dns`
[4] `https://github.com/rushmorem/publicsuffix`

and trusted. On the other hand, PowerDNS excels in scalability and flexibility, easily integrating into various deployment scenarios through its direct interaction with SQL-based databases for record information. To add support for HSTS-Enforced to these DNS server implementations, we add a new record type, labeled *HTTPREQ* with the number 66. We add the code to process this record from both SQL-based databases and bind-style config entries. To ensure correct functioning, this code mostly relies on functions that are already used for the processing of other records with some minor adjustments where necessary. To allow optional RDATA, we write the code in such a way that it expects a string of any size.

To enable operators to easily verify the correct setup of DNSSEC-validated *HTTPREQ* records, we also integrate implementations of the *HTTPREQ* record in both the Dig and Delv utilities as part of the BIND9 toolchain [44]. To test the presence of a DNSSEC-validated *HTTPREQ* record, Delv can be called from the command line using "$ delv [domain name] HTTPREQ". If a DNSSEC-verified *HTTPREQ* record is correctly specified for the provided domain name, Delv indicates that the result is fully validated using the DNSSEC-chain and returns the *HTTPREQ* and its *RRSIG* record to the command line. In case DNSSEC is not correctly enabled, Delv indicates that it received an unsigned answer. In case an *HTTPREQ* record was not correctly specified, Delv indicates that a negative response was received.

5.2.3. Chromium Implementation

To allow full testing of the proposed solution we create an implementation by modifying the Chromium source code [5]. This implementation includes all proposed design changes such as the HTTP-Required indicators, the new connection process, the device configuration panel, and all settings specified as part of the design. Additionally, we add an internal page to Chromium that allows diagnosing of network-related issues related to HTTP-Required indicators under chrome://http-required.

Removal of Traditional HSTS Methods

First, we remove the code responsible for HSTS header processing, its cache, and the persisting of its state. To cleanly remove the HSTS Preload List, we need to remove its information from the Chromium. However, since most domains with static PKP enabled also have HSTS preloaded, the data structure that stores HSTS preload information also stores the data for static PKP. To leave the information required for PKP, we modify the code that generates this data structure such that it no longer encodes the HSTS Preload List information. We leave the code responsible for processing connections when HSTS is enabled intact and temporarily always enable HSTS.

Addition of HTTPREQ Record Processing

DNSSEC support is lacking from Chromium. Therefore, to validate *HTTPREQ* records through DNSSEC resolution, we integrate libunbound [73] into Chromium. Leveraging libunbound, created by NLnet Labs, allows us to depend on its cryptographic validation of DNSSEC signatures. By default, libunbound's return structure only has a field indicating if RDATA was received as a response to a DNS query. In order for libunbound to correctly work with *HTTPREQ* records, which can be specified without any RDATA, we modify the return structure to include a field that simply indicates whether any positive response has been received. We then use this field together with the field indicating if DNSSEC-validation was successful, to check for the DNSSEC-validated *HTTPREQ* record. To reduce the time spent validating the authenticity of negative responses, which is not required for HSTS-Enforced, we also slightly modify libunbound's resolution process, such that it performs an early return in case a negative response is detected. Inside Chromium's network process, we create a single libunbound context, which can asynchronously handle multiple requests for *HTTPREQ* records.

To test the correct working of custom DNSSEC root keys, we provide a user interface inside Chromium that allows for the customization of the DNSSEC root keys used by libunbound. This interface is provided as part of the security settings and can be seen in Appendix E. Whenever, apply is hit in this interface the libunbound context is recreated with the new DNSSEC root keys. The new root keys are stored as part of the Chromium settings such that they are persistent across sessions.

[5] `https://source.chromium.org/chromium/chromium/src`

Addition of HTTP-Required Preload List Processing
The HTTP-Required Registration Website outputs the HTTP-Required Preload List in the form of a JSON file as shown in A. To minimize the cost of deployment we want to limit the effect of the list on the size of the browser. The used approach is similar to what Chromium is already using for the HSTS List. First, a simple Huffman-encoding is generated based on the number of times a character occurs in all the entries of the JSON file. A Huffman-encoding assigns a variable-length code to each character, with shorter codes assigned to more frequent characters and longer codes assigned to less frequent characters, meaning that the overall representation is reduced. Each entry in the list is character-wise reversed and encoded to their Huffman representation, this representation is added to the trie. Adding an entry to the trie means that a path is created from the root node of the trie, along this path, every transition is labeled with a single character in order of occurrence in the representation. The final node of this path is added to a list of valid exit nodes. Each entry in the list of valid exit nodes additionally stores a marker indicating whether this entry also includes its subdomains. This way whenever inclusion in the list has to be verified, a traversal of the tree using an input sequence that reaches a valid exit node is considered as included. Moreover, whenever a node is passed over that is labeled as including its subdomains, we can immediately conclude that it is included in the HTTP-Required Preload List. Developers are recommended to use a similar data structure to integrate the list into a user agent in order to reduce memory and resolution time.

Modification of Connection Process
Using the created functions and an existing function to check if a domain is an IP address, we modify the functions that check if HSTS is enabled to match Figure 5.1a and postpone the time at which these functions are called to the positions at which they are called in Figure 5.1b. The second change involves moving the call to the HSTS status check from the creation of a connection attempt to the point at which the attribute used to store the status is actually used. When the attribute is not used, we simply remove the call to the HSTS status check.

Additionally, in case HSTS is disabled, we need to facilitate the secure fallback between HTTP and HTTPS. To do this we use a custom network throttle which is created whenever a new request is made using HTTP or HTTPS. Whenever the network process detects that it is unable to contact an HTTP or HTTPS server, we first create a tab helper for the current navigation process which is responsible for storing the current URL and a boolean that indicates we are attempting to do a protocol switch. After storing this information, we redirect to a new URL, for which we swap the protocol with the new required protocol. This new connection also creates an instance of the same custom network throttle. Since the tab helper stores an indicator that we are attempting to do a protocol switch, the throttle will not redirect again. In case the redirect is successful, the tab helper is cleared and the received content is displayed. However, in case the connection fails again, we indicate a failed switch as part of the tab helper and redirect back to the URL we had previously stored. This ensures that if a server is temporarily offline and hosts two different sites across each of the protocols, then once back online a connection will be attempted to the one that was originally specified in the URL.

Addition of Settings and Interfaces
To show that full integration of HSTS-Enforced is possible, we provide several user settings and interfaces that provide extra accessibility for users. We replace the existing *chrome://net-internals#hsts* page, which provides an overview of static and dynamic HSTS and PKP information when queried with a domain, with two separate pages. The first of these pages displays PKP information for domains and is available at *chrome://net-internals#pkp* as depicted in Appendix F. The second one displays information about both types of HTTP-Required indicators for domains and is available at *chrome://net-internals#http-required* as depicted in Appendix G. We add two flags (available at *chrome://flags*), labeled hsts-for-ip-addresses and hsts-for-reserved-domains as depicted in Appendix H. These flags respectively allow the user to re-enable HSTS for IP addresses and HTTP-Required Preload List entries marked as reserved.

Finally, we add the overview panel at the *chrome://configure-devices*, which scans the network interfaces on the device and extract a list of default gateways. We display a button for each default gateway, that links to their configuration panel using their IP address. Additionally, in case only one default gateway is found, which is the most common situation for devices with a single network card, we automatically redirect to the configuration panel, since there is nothing else a user can do on the page.

6

Evaluation

In Chapter 4, we identified significant problems in the implementation of the HSTS protocol. Key requirements established for the solution included ensuring maximum security with minimum configuration and removal of the tracking capabilities.

The proposed solution flips the current opt-in approach to an opt-out approach. This approach allows us to ensure a default secure state for domains against downgrade attacks.

First, we describe in detail how the different parts of the implementation of HSTS-Enforced were configured to conduct the evaluation. Then, we use this setup to asses the proposed solution based on several performance criteria. The purpose of these criteria is to verify the correct working of the protocol and to assess its overhead, intending to prove that HSTS-Enforced is secure and can be adopted by the web without causing significant issues.

6.1. Setup

To correctly evaluate HSTS-Enforced, we need to simulate a small part of the web. We use Docker to simulate the required DNS/HTTP/HTTPS servers and run Chromium directly on the system.

To evaluate the design, we simulate all combinations of configurations that affect HTTP-Required indicators as shown in Table 6.1. As shown, some combinations of configurations correctly specify a valid HTTP-Required indicator. These domains are either present on the HTTP-Required Preload List or have both an *HTTPREQ* DNS record and DNSSEC enabled. We use eight subdomains of *hstsenforced.example.com*, for which the label corresponds directly to its combination of configurations. We include the domains listed as part of the HTTP-Required Preload List in the JSON file with the "reserved" policy. We create a single authoritative server running in Docker using the PowerDNS implementation and define nine zones as part of this server. Each subdomain listed in the table has a seperate zone. This allows us to sign some of the zones using DNSSEC, while others remain unsecured. We also create a zone for *hstsenforced.example.com* and sign it using DNSSEC. This allows us to create a small locally verifiable DNSSEC chain starting at *hstsenforced.example.com*. To ensure the DNSSEC signed zones are accepted as valid, we add the DS record corresponding to the DNSKEY record of

Subdomain label	HTTP-Required Preload List	*HTTPREQ* DNS record	Zone signed using DNSSEC	Valid HTTP-Required indicator
none.	✗	✗	✗	✗
none-sec.	✗	✗	✓	✗
dns.	✗	✓	✗	✗
dnssec.	✗	✓	✓	✓
list.	✓	✗	✗	✓
list-sec.	✓	✗	✓	✓
list-dns.	✓	✓	✗	✓
list-dnssec.	✓	✓	✓	✓

Table 6.1: The subdomains of hstsenforced.example.com, which provide all possible combinations of configurations, and whether they provide a valid HTTP-Required indicator

Web server availability	HTTP	untrusted HTTPS	HTTP + untrusted HTTPS		HTTPS	HTTP + HTTPS	
Specified protocol	HTTP/HTTPS/None	HTTP/HTTPS/None	HTTP	HTTPS/None	HTTP/HTTPS/None	HTTP	HTTPS/None
No valid HTTP-Required indicator	🛡	🛡	🛡	🛡	✓	✓	✓
∃ valid HTTP-Required indicator	!	⚠	!	⚠	✓	!	✓

Table 6.2: Results of connection attempts to websites hosting different combinations of web servers, depending on the protocol specified in the URL and whether there is a valid HTTP-Required indicator or not. ✓ indicates a connection using trusted HTTPS. 🛡 indicates a blocked connection to an unsecure server. ! indicates a connection using HTTP. ⚠ indicates a connection using untrusted HTTPS.

hstsenforced.example.com to Chromium.

Using Docker we then create three containers hosting different combinations of servers, one that listens only to HTTP, one that listens only to HTTPS, and one that listens to both. For the HTTPS servers, we sign X.509 wildcard certificates using a custom CA that we create especially for this evaluation. To switch between trusted and untrusted HTTPS connections, we add or remove the certificate of this CA to/from the root certificates used by Chromium as required. When started, these Docker containers all listen on the same IP address.

We add an *A* record to each DNS zone pointing to the IP address of the Docker containers hosting the HTTP(S) servers. Additionally, we add a Docker container that hosts a recursive DNS server. For *hstsenforced.example.com* and its subdomains we point the recursive server to the authoritative server we previously created. For all other zones, we redirect the requests to an external recursive server, specifically the recursive server hosted by Quad9.

To do a more realistic evaluation of a real world scenario, we register a public domain, create and register the required DNSSEC keys, request an HTTPS certificate using Let's Encrypt, and set up a fully working website for this domain with an *HTTPREQ* record. We set up the servers using a similar approach as was used for the eight subdomains. To simulate an external recursive server we add a Round-Trip Time (RTT) of 20 ms between the user agent and the recursive DNS server by adding a 10 ms delay in each direction to the Docker container hosting the recursive server using netem.

6.2. Criteria 1: Comprehensiveness

The comprehensiveness of HSTS-Enforced is evaluated based on the extent to which it ensures secure communications and prevents downgrade attacks.

By default, HSTS-Enforced requires HTTPS from the outset, eliminating the possibility of a downgrade. Additionally, it adequately addresses all identified side-channel attacks of HSTS by using validatable opt-outs in the form of valid HTTP-Required indicators. Since valid HTTP-Required indicators cannot be spoofed, the only way for an attacker to bypass HSTS protection under HSTS-Enforced is to have full access to the system, as full access to the system would allow an attacker to directly modify the browser or system settings required to validate unofficial indicators. Noteworthy, is that almost nothing can prevent an attacker once they have full system access, so this should be prevented by any means possible, but this protection is outside of the scope of this work.

Since HSTS-indicators are by design unspoofable, we assess that HSTS-Enforced correctly ensures communication is secure and that it provides robust protection against downgrade attacks.

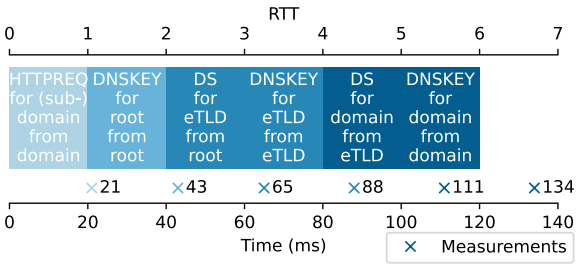
6.3. Criteria 2: Effectiveness

To assess the effectiveness of HSTS-Enforced, we test the correct working of its implementation.

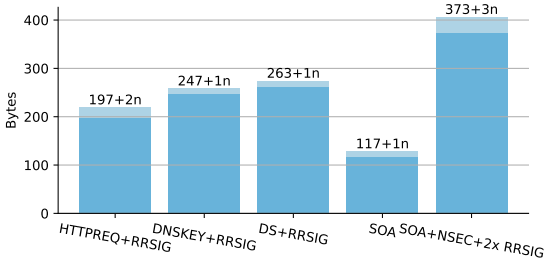
By starting the different DNS servers for *hstsenforced.example.com* and its subdomains, overriding the system's recursive DNS server with the IP address of the Docker container hosting the recursive server, adding the CA certificate to Chromium as required, and navigating to all subdomains from Chromium, we verify that for each different combination of configurations, the behavior is as expected.

Additionally, we verify that when the custom DNSSEC key for *hstsenforced.example.com* is not registered, none of the underlying *HTTPREQ* records are deemed as valid indicators.

Table 6.2 shows the results of this experiment. These results are in accordance with the design, meaning



(a) Resolution times for verifying the *HTTPREQ* DNS record, depending on which records are not cached and must be requested. Equal colors denote linked records, which are typically both cached if one of them is.



(b) Packet sizes of DNS records used during resolution of a DNSSEC-based HTTP-Required Indicator. n denotes the length of the requested domain name.

Figure 6.1: Results of the overhead analysis.

that for each subdomain that does not have a valid HTTP-Required indicator, only connections using trusted HTTPS are created, while connections using HTTP or untrusted HTTPS are blocked. In case no valid HTTP-Required indicator is present, we verify that connections using HTTP are correctly created when the server only listens to HTTP, or when the use of HTTP is explicitly specified as part of the URL. We also verify that untrusted HTTPS connections are correctly created when a valid HTTP-Required indicator is present and either untrusted HTTPS is the only protocol the server listens to or the protocol indicated as part of the URL is HTTPS or None.

6.4. Criteria 3: Overhead

To access the overhead of HSTS-Enforced, we calculated the expected and measured the average time taken to resolve an DNSSEC-based HTTP-Required indicator, measured the size of the DNS packets, measured an upper bound for the time taken to check the HTTP-Required Preload List and measured the memory and disk space usage of Chromium with and without HSTS-Enforced.

Indicator Resolution Time

Using WireShark, we measured the time taken to resolve all records required to validate the *HTTPREQ* DNS record using DNSSEC from an empty cache. Figure 6.1a illustrates the expected number of RTTs the process takes depending on which records must be retrieved and the actual times it took in our measurements, which are slightly longer than multiples of the RTT due to the processing time on the DNS server. In the worst case, the resolution could take six RTTs – in our measurements, the worst case took 134 ms with an RTT of 20 ms. However, this worst case should rarely occur in actuality because DNS servers cache records and in the case of a negative response, the verification process stops immediately. For positive responses, cache hits are likely because the DNS response packets are small (a few hundred bytes at most) and verification attempts often share required records such as those for root and eTLDs. Furthermore, DS and DNSKEY pairs are always used in tandem. Thus, if one of them is cached, usually both are. In a typical scenario, the DNSKEY for root is already cached and so are the DS and DNSKEY for the eTLD. Hence, the typical verification process should take around three RTTs.

To create an upper-bound for the time taken to check if a domain is included in the HTTP-Required Preload List, we measure the time taken to find varying domains on the HSTS Preload List. Since the HTTP-Required Preload List has no requested entries yet, the HSTS Preload List has around 144 thousand entries, and we represent both using the same data structure, we can assume that the time taken to search the HTTP-Required Preload List is at most equal to the time taken to search the HSTS Preload List. All checks for domains on the list using the Huffman-encoded trie data structure finish in less than 1 ms.

Network Load

HSTS-Enforced causes additional network traffic when verifying an *HTTPREQ* record. However, its impact is minuscule because it induces but a few DNS packets, which are small in size. Figure 6.1b shows the packet sizes of HSTS-Enforced DNS packets. They are at most around 400 bytes (for a signed negative response with a signed SOA and NSEC record), but most packets are even smaller. The size of a signed *HTTPREQ* response is around 220 bytes. Signed DNSKEY and DS responses used to verify

the signature are around 260 and 270 bytes. However, such a response does not require signature verification as described in Section 5.1.1.

Disk Usage
To estimate the additional disk size used by HSTS-Enforced, we create a Debian software package for both Chromium with and without HSTS-Enforced. The size of the Debian software package before implementing HSTS-Enforced is 89 MB. The size of the Debian software package for Chromium with HSTS-Enforced is 88 MB. The decrease of around 1 MB can be attributed to the reduced number of entries on the HTTP-Required Preload List as compared to the HSTS Preload List. Due to not all domains requiring HTTP-Required indicators, the alternative DNSSEC-based indicator, and the restriction on which domains can request addition, the HTTP-Required Preload List will likely never become as large as the HSTS Preload List. Additionally, the Huffman-encoded trie data structure representing the list will only grow by a few bytes for each newly added entry.

Memory Usage
Using the system monitor window, we measure the memory usage of the network process to be 67.6MB and 64.6MB respectively for Chromium with and without HSTS-Enforced. Other processes are unmodified and thus their memory usage does not differ. This difference is explained by the version of Chromium with HSTS-Enforced implementing two separate libraries for DNS resolution. The first of these DNS resolvers is the one originally used by Chromium that lacks full DNSSEC support. The second resolver is Unbound for validation of *HTTPREQ* records, as implemented in Section 5.2.3.

7 Discussion

This chapter discusses the limitations of the work provided in this thesis, particular choices made during the design phase, the relation to HTTPS-Only mode and HTTPS Everywhere, general recommendations to improve security on the web, recommendations for the roll-out of the HSTS-Enforced, and indirect effects of HSTS-Enforced on the web once fully adopted.

7.1. Limitations

HSTS-Enforced mainly focuses on improving the security of HSTS by removing the existing vulnerabilities and using the gained security to increase the accessibility of the web. It does not encompass protection against attacks on the HTTPS protocol, such as the HTTP/2 Rapid Reset DDoS Vulnerability [17] or QUICsand [72]. HSTS-Enforced does also not protect users against visiting malicious domains, whether visited through links received in an email, accessed by clicking a hyperlink on another website, or anywhere else. Neither does it protect a user against a compromised system. For example, it does not protect against an unofficial root certificate registered on a system. In this case, only allowing HTTPS offers no protection against the holder of the private key associated with that root certificate. Locally registering a custom root certificate means they can generate X.509 certificates for any domain, and the system will deem them valid. While HSTS-Enforced defends against the effects of a compromised network on the final security of connections, it does not directly protect the local network against being compromised. Instead, it assumes that the local network is potentially compromised and protects against attacks on the communication channels used for the web. Lastly, though HSTS-Enforced prevents websites from tracking user agents through HSTS cache exploits, it does not protect against any additional tracking that might take place elsewhere in the network.

Due to the scale of the web, this thesis is limited to a proof of concept. Focusing on common DNS software and limiting the client-side implementation to Chromium on Ubuntu ensures a manageable scope and allows for a controlled environment to validate the core concepts. This approach balances feasibility with the need to demonstrate practical application, though it does not cover the full spectrum of possible configurations or platforms. More effort is required to implement HSTS-Enforced across the full spectrum. Additionally, the Chromium implementation can be improved. For example, by switching the normal DNS resolution to also use Unbound, we can reduce the memory used by the network process.

7.2. Alternatives to DNSSEC

Only relying on the HTTP-Required Preload List to indicate that unsecure connections are required to access a website would cause the list to become relatively long and directly expose all domains that allow unsecure connections. This would directly affect the size and memory usage of user agents. Therefore, we need a validatable way to communicate that unsecure connections are required to access a particular website. Since this data relates directly to a domain name, we chose DNS to communicate this information.

We require a way to validate the information conveyed by the new DNS record. For this validation, we did not want to rely directly on a single cryptographic key, as this would cause a relatively high dependency on the security of such a key. Therefore, we required a cryptographic chain to create a robust validation method. Create of a new cryptographic chain would increase the complexity of the protocol and by extension the web, which would hinder the adoption of HSTS-Enforced. The only other widely adopted cryptographic chain besides the DNSSEC chain is the X.509 chain.

We experimented with the use of the X.509 chain to validate information conveyed by DNS messages. However, this method requires websites that want to only use HTTP connections to still request a certificate, which can also be used to create HTTPS connections. Requiring users to obtain an X.509 certificate to use HTTP directly contradicts the conventional purpose of X.509 certificates. Additionally, since HTTP servers do not communicate X.509 certificates, it requires the X.509 certificate of a domain to also be included as part of information distributed by DNS. Lastly, in case this method is used, this X.509 certificate would still need to be verified against the X.509 root certificates causing additional overhead.

There is already an existing record for conveying information about a domain's certificate, namely the TLSA record type which is part of DANE. However, this record only serves a hash of the certificate intended to be used to validate an X.509 certificate sent by the HTTPS server after it has been validated using DNSSEC. For secure validation of an opt-out, we instead require the entire X.509 certificate. Due to the size of X.509 certificates, conveying them across DNS leads to fragmentation of responses, which causes additional overhead. Additionally, since each domain would require a single record containing their full certificate, these records would take up relatively much space in the DNS cache of recursive servers. These two factors cause specifying a full X.509 certificate as part of the DNS system to put a significant load on the DNS infrastructure. Additionally, the relative complexity as compared to DNSSEC, for which most DNS server configurations have a simple command or toggle to turn on automatic key management, causes this method to be simply more complex than needed.

Instead, while its adoption is relatively low, DNSSEC is purpose-built to secure DNS records, directly addressing the needs of the project to validate the authenticity of data conveyed by the DNS infrastructure through an existing cryptographic chain.

7.3. Relation to HTTPS-Only Mode/HTTPS Everywhere

HTTPS Everywhere [26] was a collaborative project between the Electronic Frontier Foundation (EFF) and the Tor Project. Introduced in 2010, it was an extension for browsers that automatically switched the browser from unsecure HTTP to more secure HTTPS connections whenever possible, and showed a warning when HTTPS was unavailable. However, if HTTPS was not available, it would silently fall back to HTTP to access the website instead. After noting the security benefit of HTTPS Everywhere, browsers started to adopt its features natively into the browser. In 2021, with Firefox update 91 [49], private browsing tabs would upgrade to HTTPS whenever possible, later this feature was also enabled for normal tabs. In 2023, the same feature was also added to Chromium under the title HTTPS Upgrades [19], and to Safari as part of an unnamed update [27]. The warning before falling back to HTTP from HTTPS was later incorporated into Firefox under the name HTTPS-Only mode [50], and into Chromium under the name HTTPS-First mode. By default, these modes are disabled. Safari has not yet incorporated an option to show a warning between the fallback to HTTP.

Chromium has also started to work on HTTPS-First Mode V2 For Engaged Sites. This new version of HTTPS-First Mode recognized that users would not like it if Chromium enabled HTTPS-First Mode by default since it might show them a big red error screen on some of their websites that only provide HTTP connections. Thus, the Chromium team started experimenting with automatically enabling the protection based on how likely the website was to only provide HTTP connections. To estimate this likelihood, they look at how often the user typically accesses that particular website over HTTPS, but they also automatically enable the features for users that only very rarely use HTTP. To provide this feature, the browser internally keeps track of how often a user uses a particular protocol to visit individual websites and how many total visits have been made using HTTP and HTTPS.

Since these modes are not enabled by default, a user has to opt-in to this protection. Additionally, if HSTS is not enabled for a domain, users can still click through the warnings and continue to an unsecure website, which is a behavior that users commonly exhibit [91, 94]. Therefore, these modes alone are not

enough to protect against downgrade attacks.

7.4. Relation to Secure DNS Communication

An alternative to local DNSSEC validation is to rely on an external resolver to perform validation and communicate with this resolver over a secure communication channel. Examples of such protocols are DNS-over-HTTPS (DoH) [37], DNS-over-TLS (DoT) [23], or DNS-over-QUIC (DoQ) [41], which use X.509 certificates to validate and secure the resolver similarly to HTTPS. Using these protocols would reduce the requests needed to validate an *HTTPREQ* record. However, a resolver can spoof the DNSSEC flag that indicates that validation was successful without compromising the communication channel. An option is to limit the resolvers that can be used to validate the *HTTPREQ* record to a select group of resolvers that are trusted to not spoof DNSSEC flags. However, this would restrict users to a select group of resolvers and prevent the use of local resolvers, making it impossible to limit the *HTTPREQ* record to a limited set of users.

7.5. Inherent Trust of HSTS-Enforced

Since HSTS-Enforced relies on DNSSEC and the HTTP-Required Registration Website to validate HTTP-Required indicators, it places trust in the parent DNSSEC zones and the HTTP-Required Registration Website. Therefore, it is very important that these instances take their roles seriously and maintain a high level of security and reliability. Any compromise in the parent DNSSEC zones or the HTTP-Required Registration Website could undermine the trustworthiness of the entire HSTS-Enforced system.

Specifically, the parent DNSSEC zones must ensure that their cryptographic keys are securely managed and regularly rotated to prevent unauthorized access or tampering. Any breach in these zones could result in attackers manipulating DNS records, leading to false validation of HTTP-Required indicators, which could have serious security implications.

Similarly, the HTTP-Required Registration Website must implement robust security measures to protect against threats such as man-in-the-middle attacks, data breaches, or website spoofing. The integrity of the registration process and the data it handles is crucial, as any vulnerability could allow malicious actors to falsely register domains as HTTP-Required, potentially leading to the inadvertent trust of unsecure or malicious websites.

In summary, the security and reliability of HSTS-Enforced are heavily dependent on the integrity and security practices of the parent DNSSEC zones and the HTTP-Required Registration Website. It is imperative that these entities adhere to the highest standards of security to maintain the overall trust in the system.

7.6. Security Recommendations

While browsers have already robustly implemented HSTS, and can thus adopt HSTS-Enforced with relative ease, there are still other user agents, such as command-line utilities, HTTP libraries, and packages that still exhibit shortcomings in this area. For example, curl and wget partially support HSTS. However, they do not support HSTS preloading. Additionally, curl does not enable HSTS by default, instead requiring additional arguments. We would like to stress the importance of secure connections in all web traffic-generating software. The current approach taken by these user agents relies heavily on developers to correctly specify the correct protocol as part of the URL, as explained in [11], this can cause leaking of API authentication and other sensitive information. Currently, these user agents also still default to HTTP as the default protocol when none is specified. This methodology always requires developers to explicitly specify that HTTPS should be used. However, human errors, such as typing a URL without an S after HTTP, or omitting the protocol entirely, are relatively simple mistakes, that should not compromise security.

These user agents, have often not implemented HSTS as it would require persisting the cache across requests. Since HSTS-Enforced no longer requires the persistence of HSTS data and provides a way to securely switch between HTTP and HTTPS, we stress the importance for these user agents to adopt HSTS-Enforced, which is relatively easier to adopt than traditional HSTS. We do note that this might increase the size of libraries when compiled as part of software that should never use unsecure

connections, and thus would never need the use of HSTS-Enforced. Therefore, we suggest that by default libraries only allow secure connections, and HSTS-Enforced can be enabled by developers if unsecure connections are required.

7.7. Adoptability

To assess how easy it is for the public internet to adopt HSTS-Enforced, we look at its underlying protocols. HSTS-Enforced ensures ease of adoption by using existing protocols, such as HTTP and DNSSEC. Most DNS management software already supports DNSSEC and generating a key and signing a record has often been fully automated by registrars. Extending these platforms with the new *HTTPREQ* record requires some changes. However, extending DNS with new types has been done many times and is relatively easy, as shown in Section 5.2.2. Additionally, to prevent any hurdles caused by the relatively low adoption rate of DNSSEC by eTLDs, the HTTP-Required Registration Website provides a way to opt out without relying on DNSSEC. This second indicator ensures that if an HTTPS-Required indicator is required, it is still possible to specify one for a domain. Additionally, the API of the HTTP-Required Registration Website allows the developers of user agents to acquire the latest version of the HTTP-Required Preload List with relative ease, thereby allowing them to easily adopt the indicator and update to the latest version of the list on its release.

7.8. Roll-out of the Protocol

We envision a gradual adoption period during which website operators can deploy the HTTP-Required indicators for domains that require unsecure connections. The transition can be assisted by a user-based collaborative that collects a list of websites that need to adapt their configurations, allowing us to contact these websites if necessary.

In case all eTLDs eventually adopt DNSSEC, all domains can use *HTTPREQ* records instead of the HTTP-Required Registration Website. The website can then be retired and the list can be reduced to only the entries marked as "evaluation" or "reserved".

7.9. Practical Implications

The widespread adoption of HSTS-Enforced would significantly enhance the overall security posture of the web. By mandating that clients only interact with sites over HTTPS unless indicated by the website operators, the protocol successfully mitigates the risk of downgrade attacks. This ensures the confidentiality and integrity of user data during transit, providing a secure browsing experience and fostering greater trust in online services. Additionally, since the default behavior is to communicate using only trusted HTTPS, there would no longer be a requirement for HTTP servers that only serve redirects to their corresponding HTTPS servers. Overall, adopting HSTS-Enforced would significantly enhance web security, streamline client and server behavior, and remove the effects of human error on security.

8

Conclusion

In this thesis, we introduced a novel protocol called HSTS-Enforced aimed at enhancing web security by fundamentally rethinking the way HSTS is communicated. Our protocol leverages Secure-by-Default principles, ensuring that websites are inherently robust against downgrade attacks, without requiring additional steps from website operators or end-users.

Our protocol addresses multiple key limitations of the traditional HSTS mechanism. First, it eliminates the initial vulnerability period associated with the first non-secure request by pre-emptively enforcing HTTPS connections for all websites. Secondly, it allows website operators to opt out of this security and allow unsecure connections through the specification of HTTP-Required indicators. We introduce two such indicators, one of which is preloaded into the browser and the second of which uses DNSSEC resolution and validation. To remove the potential to spoof such indicators, we ensure that the authenticity of these indicators can be validated.

Furthermore, our approach simplifies the deployment and maintenance of HSTS for website operators. Traditional HSTS requires specific header configurations and careful management of *includeSubDomains* and *max-age* directives, which are easy to misconfigure causing them to become invalid. In contrast, our protocol only requires operators to configure an opt-out if desired, removing the potential for human error and configuration oversights to unintentionally decrease security. Additionally, the Secure-by-Default paradigm allows us to consistently apply the highest level of security across all web properties without additional administrative burden.

Performance evaluations and security analyses presented in this thesis demonstrate that our protocol not only matches but surpasses the security guarantees provided by traditional HSTS. The implementation details, including the integration with existing DNS infrastructure and web browsers, showcase the practicality and feasibility of our approach. By leveraging existing technologies and protocols, we ensure backward compatibility and smooth transitions for web services adopting HSTS-Enforced.

In summary, HSTS-Enforced represents a significant advancement in web security. By adhering to secure-by-default principles and relying on existing protocols, HSTS-Enforced mitigates initial connection vulnerabilities, simplifies administration, becomes more error-resistant, and enhances overall web security. This makes HSTS-Enforced a user-friendly alternative to traditional HSTS. We believe that the HSTS-Enforced protocol will play a critical role in shaping the future landscape of web security, making the internet a safer place for everyone.

8.1. Future work

Future work could focus on further optimizing the protocol, expanding its adoption, and continuously evaluating its effectiveness against emerging security threats. Additionally, we want to highlight three specific areas of improvement.

8.1.1. Unified HSTS Interface

Creating a shared HSTS checking library or service presents a valuable opportunity to standardize and streamline security checks across different applications and systems. This interface could come in many forms. For example, it could just be a library that can be imported as required but it could also be a service designed to run as part of the operating system which can be interacted with as required. By centralizing HSTS enforcement, redundancy can be minimized and the consistency of security practices across software that relies on HSTS to secure connections can be guaranteed. Future research could focus on developing this library, ensuring it is robust and interoperable with various systems, and evaluating its performance and security benefits. Additionally, exploring how to integrate this library with existing security protocols and tools would be a critical step.

8.1.2. Patch-based updates to the HTTP-Required Preload List

Patch-based updating of the HTTP-Required Preload List is another promising area for future research. Automating this process, thereby removing the requirement for developers to manually update the list, can ensure that the latest security policies are consistently enforced. However, there are potential downsides, such as the risk of introducing errors during automated updates, the need for reliable validation mechanisms to prevent malicious entries, and missed updates causing the HTTP-Required Preload List of a client to become outdated. Future work should address these challenges by developing secure and efficient distribution methods, exploring rigorous testing and validation frameworks for automated updates, and assessing the overall impact on system security and reliability.

8.1.3. Improved Browser Interfaces

Improving browser interfaces with a focus on increasing the user's awareness of security and privacy is another avenue for improving internet security. For example, currently, when the lock icon is clicked in a browser, very little information about the connection is provided to the user. Most browsers simply show that HTTPS is used and allow the user to click through to view more details of the connection such as TLS version and the certificate that is used. Extending these information panels with comprehensive details about other protocols used to secure connections helps increase the user's awareness of their security and privacy. Examples of additional information that can be shown are the state of HSTS, what type of HTTP-Required indicator was received, whether the IP address of the server was resolved using DNSSEC, the usage of DoH [37], DoT [23], or DoQ [41], and many more security-focused information. Such detailed information could empower users to make informed decisions about their online security and make web developers more aware of the available security protocols. Future work could involve designing user-friendly interfaces, measuring which security information is the most important for end-users, and assessing the impact on the behavior and awareness of users.

Bibliography

[1] Josh Aas, Daniel McCarney, and Roland Shoemaker. *Multi-Perspective Validation Improves Domain Validation Security*. Let's Encrypt. https://letsencrypt.org/2020/02/19/multi-perspective-validation.html. 2020.

[2] Abdelrahman Abdou and P. C. Van Oorschot. "Server Location Verification (SLV) and Server Location Pinning: Augmenting TLS Authentication". In: *ACM TOPS* 21.1 (2017).

[3] Giorgi Akhalaia et al. "Secure Encrypted Connection on Georgian Website". In: *Cybersecurity Providing in Information and Telecommunication Systems*. 2023.

[4] Johanna Amann et al. "Mission accomplished?: HTTPS security after diginotar". In: *ACM IMC*. 2017.

[5] Roy Arends et al. *Resource records for the DNS security extensions*. RFC 4034. 2005.

[6] Davide Balzarotti et al. "Multi-Module Vulnerability Analysis of Web-based Applications". In: *ACM CCS*. 2007.

[7] Davide Balzarotti et al. "Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications". In: *IEEE S&P*. 2008.

[8] Richard Barnes et al. *Automatic Certificate Management Environment (ACME)*. RFC 8555. 2019.

[9] Adam Barth. *The Web Origin Concept*. RFC 6454. 2011.

[10] David Basin et al. "ARPKI: Attack Resilient Public-Key Infrastructure". In: *ACM CCS*. 2014.

[11] Jerry Bell. *Your API Shouldn't Redirect HTTP to HTTPS*. @jviide. https://jviide.iki.fi/http-redirects. 2024.

[12] Nikita Borisov, Ian Goldberg, and Eric Brewer. "Off-the-record communication, or, why not to use PGP". In: *ACM WPES*. 2004.

[13] William Buchanan, Scott Helme, and Alan Woodward. "Analysis of the Adoption of Security Headers in HTTP". In: *IET Information Security* 12 (2017).

[14] Franco Callegati, Walter Cerroni, and Marco Ramilli. "Man-in-the-Middle Attack to the HTTPS Protocol". In: *IEEE S&P*. 2009.

[15] Stefano Calzavara, Alvise Rabitti, and Michele Bugliesi. "Content Security Problems?: Evaluating the Effectiveness of Content Security Policy in the Wild". In: *ACM CCS*. 2016.

[16] Stefano Calzavara, Alvise Rabitti, and Michele Bugliesi. "Semantics-Based Analysis of Content Security Policy Deployment". In: *ACM TWEB* 12.2 (2018).

[17] CERT-EU. *HTTP/2 Rapid Reset Ddos Vulnerability*. CERT-EU. https://cert.europa.eu/publications/security-advisories/2023-074/. 2023.

[18] Ping Chen et al. "Security Analysis of the Chinese Web: How well is it protected?" In: *CCS SafeConfig*. 2014.

[19] Chrome Security team. *Towards HTTPS by Default*. Chromium. https://blog.chromium.org/2023/08/towards-https-by-default.html. 2023.

[20] Cybersecurity and Infrastructure Security Agency (CISA) and Federal Bureau of Investigation (FBI) and National Security Agency (NSA) and Australian Cyber Security Centre and Canadian Centre for Cyber Security and New Zealand's Computer Emergency Response Team and United Kingdom's National Cyber Security Centre and Germany's Federal Office for Information Security (BSI) and Netherlands' National Cyber Security Centre. *Shifting the Balance of Cybersecurity Risk: Security-by-Design and Default Principles*. Cybersecurity and Infrastructure Security Agency. https://www.cisa.gov/news-events/alerts/2023/04/13/shifting-balance-cybersecurity-risk-security-design-and-default-principles. 2023.

[21] Emma Dauterman and Henry Corrigan-Gibbs. "Lightweb: Private Web Browsing Without All The Baggage". In: *ACM HotNets*. 2023.

[22] Killian Davitt et al. "CoStricTor: Collaborative HTTP Strict Transport Security in Tor Browser". In: *PETS*. 2024.

[23] John Dickinson et al. *DNS Transport over TCP - Implementation Requirements*. RFC 7766. 2016.

[24] Roger Dingledine, Nick Mathewson, and Paul Syverson. "Tor: The Second-Generation Onion Router". In: *USENIX Security*. 2004.

[25] DeJean Dunbar, Patrick Hill, and Yu-Ju Lin. "Survey of United States Related Domains: Secure Network Protocol Analysis". In: *International Journal of Network Security & Its Applications (IJNSA)* 14 (2022).

[26] Electronic Frontier Foundation (EFF). *HTTPS Everywhere*. Electronic Frontier Foundation. `https://www.eff.org/https-everywhere`. 2010.

[27] Electronic Frontier Foundation (EFF). *HTTPS: It's Actually Everywhere*. Electronic Frontier Foundation. `https://www.eff.org/deeplinks/2021/09/https-actually-everywhere`. 2021.

[28] Chris Evans, Chris Palmer, and Ryan Sleevi. *Public Key Pinning Extension for HTTP*. RFC 7469. 2015.

[29] Roy Fielding et al. *Hypertext Transfer Protocol – HTTP/1.1*. RFC 2616. 1999.

[30] Hal Finney et al. *OpenPGP Message Format*. RFC 4880. 2007.

[31] Global Stats. *Browser Market Share*. Statcounter. `https://gs.statcounter.com/browser-market-share/`. 2024.

[32] Tom van Goethem et al. "Large-Scale Security Analysis of the Web: Challenges and Findings". In: *Trust and Trustworthy Computing (TRUST)*. 2014.

[33] Hélder Gomes et al. "Secure Browsing in Local Government: The Case of Portugal". In: *Journal of Web Engineering (JWE)* 20.4 (2021).

[34] Matthew Van Gundy and Hao Chen. "Noncespaces: Using Randomization to Enforce Information Flow Tracking and Thwart Cross-Site Scripting Attacks". In: *NDSS*. 2009.

[35] Phillip Hallam-Baker, Rob Stradling, and Jacob Hoffman-Andrews. *DNS Certification Authority Authorization (CAA) Resource Record*. RFC 8659. 2019.

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

[37] Paul E. Hoffman and Patrick McManus. *DNS Queries over HTTPS (DoH)*. RFC 8484. 2018.

[38] Paul E. Hoffman and Jakob Schlyter. *The DNS-Based Authentication of Named Entities (DANE) Transport Layer Security (TLS) Protocol: TLSA*. RFC 6698. 2012.

[39] Hsu-Chun Hsiao et al. "An Investigation of Cyber Autonomy on Government Websites". In: *ACM WWW*. 2019.

[40] Yao-Wen Huang et al. "Web Application Security Assessment by Fault Injection and Behavior Monitoring". In: *ACM WWW*. 2003.

[41] Christian Huitema, Sara Dickinson, and Allison Mankin. *DNS over Dedicated QUIC Connections*. RFC 9250. 2022.

[42] Internet Assigned Numbers Authority (IANA). *Root Servers*. Internet Assigned Numbers Authority. `https://www.iana.org/domains/root/servers`. 2023.

[43] Internet Assigned Numbers Authority (IANA). *Special-Use Domain Names*. Internet Assigned Numbers Authority. `https://www.iana.org/assignments/special-use-domain-names/special-use-domain-names.xhtml`. 2023.

[44] Internet Systems Consortium (ICS). *BIND*. Internet Systems Consortium. `https://www.isc.org/bind/`. 2024.

[45] Yaoqi Jia et al. "Man-in-the-browser-cache: Persisting HTTPS attacks via browser cache poisoning". In: *Computers & Security* 55.1 (2015).

[46] Trevor Jim, Nikhil Swamy, and Michael Hicks. "Defeating Script Injection Attacks with Browser-Enforced Embedded Policies". In: *ACM WWW*. 2007.

[47] Nenad Jovanović, Christopher Krügel, and Engin Kirda. "Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities". In: *IEEE S&P*. 2006.

[48] Florian Kerschbaum. "Simple Cross-Site Attack Prevention". In: *International Conference on Security and Privacy in Communications Networks*. 2007.

[49] C. Kerschbaumer et al. *Firefox 91 Introduces HTTPS by Default in Private Browsing*. Mozilla. `https://blog.mozilla.org/security/2021/08/10/firefox-91-introduces-https-by-default-in-private-browsing/`. 2021.

[50] C. Kerschbaumer et al. *HTTPS-Only: Upgrading all connections to https in Web Browsers*. Mozilla. `https://research.mozilla.org/files/2021/03/https_only_upgrading_all_connections_to_https_in_web_browsers.pdf`. 2021.

[51] Tiffany Hyun-Jin Kim et al. "Accountable key infrastructure (AKI): a proposal for a public-key validation infrastructure". In: *ACM WWW*. 2013.

[52] Engin Kirda et al. "Client-Side Cross-Site Scripting Protection". In: *Computers & Security* 28.7 (2009).

[53] Emre Kısa and Emin Tatlı. "Analysis of HTTP Security Headers in Turkey". In: *International Information Security and Cryptography Conference*. 2016.

[54] Michael Kranch and Joseph Bonneau. "Upgrading HTTPS in Mid-Air: An Empirical Study of Strict Transport Security and Key Pinning". In: *NDSS*. 2015.

[55] Ben Laurie et al. *Certificate Transparency Version 2.0*. RFC 9162. 2021.

[56] Arturs Lavrenovs and F. Jesús Rubio Melón. "HTTP Security Headers Analysis of Top One Million Websites". In: *International Conference on Cyber Conflict (CyCon)*. 2018.

[57] Xurong Li et al. "HSTS Measurement and an Enhanced Stripping Attack Against HTTPS". In: *European Alliance for Innovation SecureComm*. 2017.

[58] Dr. Arthur Y. Lin et al. *A Framework for IP Based Virtual Private Networks*. RFC 2764. 2000.

[59] Mike Ter Louw and V. N. Venkatakrishnan. "Blueprint: Robust Prevention of Cross-site Scripting Attacks for Existing Browsers". In: *IEEE S&P*. 2009.

[60] Meng Luo et al. "Time Does Not Heal All Wounds: A Longitudinal Analysis of Security-Mechanism Support in Mobile Browsers". In: *NDSS*. 2019.

[61] Moxie Marlinspike. *More Tricks For Defeating SSL In Practice*. Black Hat USA. `https://www.blackhat.com/presentations/bh-dc-09/Marlinspike/BlackHat-DC-09-Marlinspike-Defeating-SSL.pdf`. 2009.

[62] Moxie Marlinspike. *New Tricks For Defeating SSL In Practice*. Black Hat DC. `https://www.blackhat.com/presentations/bh-dc-09/Marlinspike/BlackHat-DC-09-Marlinspike-Defeating-SSL.pdf`. 2009.

[63] Abner Mendoza, Phakpoom Chinprutthiwong, and Guofei Gu. "Uncovering HTTP Header Inconsistencies and the Impact on Desktop/Mobile Websites". In: *ACM WWW*. 2018.

[64] David L. Mills et al. *Network Time Protocol Version 4: Protocol and Algorithms Specification*. RFC 5905. 2010.

[65] Paul Mockapetris. *Domain names - implementation and specification*. RFC 1035. 1987.

[66] Mozilla Foundation. *Public Suffix List*. Public Suffix List. `https://publicsuffix.org/`. 2022.

[67] J. Mtsweni. "Analyzing the security posture of South African websites". In: *Information Security for South Africa (ISSA)*. 2015.

[68] Yogesh Mundada, Nick Feamster, and Balachander Krishnamurthy. "Half-Baked Cookies: Hardening Cookie-Based Authentication for the Modern Web". In: *ACM Asia CCS*. 2016.

[69] Yogesh Mundada et al. *Half-Baked Cookies: Client Authentication on the Modern Web*. 2014.

[70] National Institute of Standards and Technology (NIST). "Advanced Encryption Standard (AES)". In: *Federal InformationProcessing Standards Publication* 46.3 (2001).

[71] National Institute of Standards and Technology (NIST). "Data Encryption Standard (DES)". In: *Federal InformationProcessing Standards Publication* 46.3 (1999).

[72] Marcin Nawrocki et al. "QUICsand: Quantifying QUIC Reconnaissance Scans and DoS Flooding Events". In: *ACM IMC*. 2021.

[73] NLnet Labs. *Unbound*. NLnet Labs. `https://unbound.docs.nlnetlabs.nl/en/latest/`. 2024.

[74] Ivan S. Petrov et al. "Measuring the Rapid Growth of HSTS and HPKP Deployments". In: *IET CANS*. 2017.

[75] Tadeusz Pietraszek and Chris Vanden Berghe. "Defending Against Injection Attacks Through Context-Sensitive String Evaluation". In: *RAID*. 2005.

[76] PowerDNS Team. *PowerDNS*. PowerDNS. `https://www.powerdns.com/`. 2024.

[77] Bareño Gutierrez Raúl and Alexandra María Lopez Sevillano. "Services cloud under HSTS, Strengths and weakness before an attack of man in the middle MITM". In: *Congreso Internacional de Innovacion y Tendencias en Ingenieria (CONIITI)*. 2017.

[78] Mengxia Ren and Chuan Yue. "Coverage and Secure Use Analysis of Content Security Policies via Clustering". In: *IEEE EuroS&P*. 2023.

[79] Eric Rescorla. *HTTP Over TLS*. RFC 2818. 2000.

[80] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems". In: *Communications of the ACM* 21.2 (1978).

[81] J. V. Roig and Eunice Grace Gatdula. *HSTS Preloading is Ineffective as a Long-Term, Wide-Scale MITM-Prevention Solution: Results from Analyzing the 2013 - 2017 HSTS Preload List*. 2019.

[82] Sebastian Roth et al. "The Security Lottery: Measuring Client-Side Web Security Inconsistencies". In: *USENIX Security*. 2022.

[83] Sergio de los Santos and José Torres. "Analysing HSTS and HPKP implementation in both browsers and servers". In: *IET Information Security* 12.4 (2018).

[84] Jose Selvi. *Bypassing HTTP Strict Transport Security*. Black Hat Europe 2014. `https://www.blackhat.com/docs/eu-14/materials/eu-14-Selvi-Bypassing-HTTP-Strict-Transport-Security-wp.pdf`. 2014.

[85] Hendrik Siewert et al. "On the Security of Parsing Security-Relevant HTTP Headers in Modern Browsers". In: *IEEE S&P Workshops*. 2022.

[86] Signal Foundation. *Signal Documentation*. Signal. `https://signal.org/docs/`. 2024.

[87] Signal Foundation. *Signal Home*. Signal. `https://signal.org/#signal`. 2024.

[88] Suphannee Sivakorn, Angelos D. Keromytis, and Jason Polakis. "That's the Way the Cookie Crumbles: Evaluating HTTPS Enforcing Mechanisms". In: *ACM WPES*. 2016.

[89] Suphannee Sivakorn, Iasonas Polakis, and Angelos D. Keromytis. "The Cracked Cookie Jar: HTTP Cookie Hijacking and the Exposure of Private Information". In: *IEEE S&P*. 2016.

[90] Suphannee Sivakorn, Patsita Sirawongphatsara, and Nuttaya Rujiratanapat. "Web Encryption Analysis of Internet Banking Websites in Thailand". In: *Joint Conference on Computer Science and Software Engineering (JCSSE)*. 2020.

[91] Andreas Sotirakopoulos, Kirstie Hawkey, and Konstantin Beznosov. "On the Challenges in Usable Security Lab Studies: Lessons Learned from Replicating a Study on SSL Warnings". In: *ACM SOUPS*. 2011.

[92] Edi Stambuk, Stjepan Gros, and Marin Vukovic. "Analyzing Web Security Features using Crawlers: Study of Croatian Web". In: *International Conference on Telecommunications (ICT)*. 2021.

[93] Standards for Efficient Cryptography Group (SECG). *SEC 1: Elliptic Curve Cryptography*. Standards for Efficient Cryptography Group. `https://www.secg.org/sec1-v2.pdf`. 2009.

[94] Joshua Sunshine et al. "Crying Wolf: An Empirical Study of SSL Warning Effectiveness". In: *USENIX Security*. 2009.

[95] Paul Syverson and Matthew Traudt. "HSTS Supports Targeted Surveillance". In: *USENIX FOCI*. 2018.

[96] Sanaa Taha and Xuemin (Sherman) Shen. “Mixed Network”. In: *Encyclopedia of Wireless Networks*. Springer, 2020.

[97] The Chromium Authors. *Chromium*. Chromium. `https://www.chromium.org/home/`. 2024.

[98] The Chromium Authors. *HSTS Preload List Submission*. HSTS Preload List Submission. `https://hstspreload.org/`. 2012.

[99] The I2P Project. *Garlic Routing*. I2P. `https://geti2p.net/en/docs/how/garlic-routing`. 2024.

[100] Matthew Traudt and Paul F. Syverson. “Does Pushing Security on Clients Make Them Safer?” In: *HotPETs*. 2019.

[101] Philipp Vogt et al. “Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis”. In: *NDSS*. 2007.

[102] Congli Wang et al. “Analyzing the Browser Security Warnings on HTTPS Errors”. In: *IEEE ICC*. 2019.

[103] Gary Wassermann and Zhendong Su. “Static Detection of Cross-Site Scripting Vulnerabilities”. In: *International Conference on Software Engineering (ICSE)*. 2008.

[104] Lukas Weichselbaum et al. “CSP Is Dead, Long Live CSP! On the Insecurity of Whitelists and the Future of Content Security Policy”. In: *ACM CCS*. 2016.

[105] Mike West. *Initial Assignment for the Content Security Policy Directives Registry*. RFC 7762. 2016.

[106] Peter Wurzinger et al. “SWAP: Mitigating XSS Attacks using a Reverse Proxy”. In: *ICSE Workshop on Software Engineering for Secure Systems*. 2009.

[107] Yichen Xie and Alex Aiken. “Static Detection of Security Vulnerabilities in Scripting Languages”. In: *USENIX Security*. 2006.

A

Commented HTTP-Required Preload List

```
// This file contains the HTTP-Required list in a machine readable format.

// The top-level element is a dictionary with one key: "entries", which contains
// the details for each domain.
//
// "entries" is a list of objects. Each object has the following members:
//   name: (string) the nameserver name of the domain in question.
//   policy: (string) the policy under which the domain is part of the
//     preload list. This field is used to better understand the list.
//       - evaluation: domains used for evaluation.
//       - reserved: domains which can not be purchased from a registrar
//           but should be http compatible to allow local usage.
//       - requested: domains which requested to be entered on the list
//           through the HTTP-Required list website.
//   include_subdomains:
//       - true: Also allow subdomains of this domain to be accessed over HTTP.
//       - false: Do not allow subdomains of this domain to be accessed over HTTP.
//

{
  "entries": [
// These are the entries that are used for evaluation of HSTS-Enforced
    {
      "name": "list.hstsenforced.example.com",
      "policy": "evaluation",
      "include_subdomains": true
    },
    {
      "name": "list-sec.hstsenforced.example.com",
      "policy": "evaluation",
      "include_subdomains": true
    },
    {
      "name": "list-dns.hstsenforced.example.com",
      "policy": "evaluation",
      "include_subdomains": true
    },
    {
      "name": "list-dnssec.hstsenforced.example.com",
      "policy": "evaluation",
```

```
      "include_subdomains": true
    },
// These are the entries that are reserved by the IANA
    {
      "name": "alt",
      "policy": "reserved",
      "include_subdomains": true
    },
    {
      "name": "example",
      "policy": "reserved",
      "include_subdomains": true
    },
    {
      "name": "example.net",
      "policy": "reserved",
      "include_subdomains": true
    },
    {
      "name": "example.org",
      "policy": "reserved",
      "include_subdomains": true
    },
    {
      "name": "invalid",
      "policy": "reserved",
      "include_subdomains": true
    },
    {
      "name": "local",
      "policy": "reserved",
      "include_subdomains": true
    },
    {
      "name": "localhost",
      "policy": "reserved",
      "include_subdomains": true
    },
    {
      "name": "test",
      "policy": "reserved",
      "include_subdomains": true
    },
    {
      "name": "xn--kgbechtv",
      "policy": "reserved",
      "include_subdomains": true
    },
    {
      "name": "xn--hgbk6aj7f53bba",
      "policy": "reserved",
      "include_subdomains": true
    },
    {
      "name": "xn--0zwm56d",
      "policy": "reserved",
      "include_subdomains": true
    },
    {
      "name": "xn--g6w251d",
      "policy": "reserved",
```

```
      "include_subdomains": true
    },
    {
      "name": "xn--80akhbyknj4f",
      "policy": "reserved",
      "include_subdomains": true
    },
    {
      "name": "xn--11b5bs3a9aj6g",
      "policy": "reserved",
      "include_subdomains": true
    },
    {
      "name": "xn--jxalpdlp",
      "policy": "reserved",
      "include_subdomains": true
    },
    {
      "name": "xn--9t4b11yi5a",
      "policy": "reserved",
      "include_subdomains": true
    },
    {
      "name": "xn--deba0ad",
      "policy": "reserved",
      "include_subdomains": true
    },
    {
      "name": "xn--zckzah",
      "policy": "reserved",
      "include_subdomains": true
    },
    {
      "name": "xn--hlcj6aya9esc7a",
      "policy": "reserved",
      "include_subdomains": true
    },
// These are the entries that were requested by domain owners
    {
      "name": "http.badssl.com",
      "policy": "requested",
      "include_subdomains": false
    }
  ],
  "valid_until": 1730073600
}
```

B

Raw HTTP-Required Preload List


```
{"entries":[{"name":"list.hstsenforced.example.com","policy":"evaluation","include_subdomains":true},{"name":"list-sec.hstsenforced.example.com","policy":"evaluation","include_subdomains":true},{"name":"list-dns.hstsenforced.example.com","policy":"evaluation","include_subdomains":true},{"name":"list-dnssec.hstsenforced.example.com","policy":"evaluation","include_subdomains":true},{"name":"alt","policy":"reserved","include_subdomains":true},{"name":"example","policy":"reserved","include_subdomains":true},{"name":"example.net","policy":"reserved","include_subdomains":true},{"name":"example.org","policy":"reserved","include_subdomains":true},{"name":"invalid","policy":"reserved","include_subdomains":true},{"name":"local","policy":"reserved","include_subdomains":true},{"name":"localhost","policy":"reserved","include_subdomains":true},{"name":"test","policy":"reserved","include_subdomains":true},{"name":"xn--kgbechtv","policy":"reserved","include_subdomains":true},{"name":"xn--hgbk6aj7f53bba","policy":"reserved","include_subdomains":true},{"name":"xn--0zwm56d","policy":"reserved","include_subdomains":true},{"name":"xn--g6w251d","policy":"reserved","include_subdomains":true},{"name":"xn--80akhbyknj4f","policy":"reserved","include_subdomains":true},{"name":"xn--11b5bs3a9aj6g","policy":"reserved","include_subdomains":true},{"name":"xn--jxalpdlp","policy":"reserved","include_subdomains":true},{"name":"xn--9t4b11yi5a","policy":"reserved","include_subdomains":true},{"name":"xn--deba0ad","policy":"reserved","include_subdomains":true},{"name":"xn--zckzah","policy":"reserved","include_subdomains":true},{"name":"xn--hlcj6aya9esc7a","policy":"reserved","include_subdomains":true},{"name":"http.badssl.com","policy": "requested","include_subdomains":false}],"valid_until":1730073600}
```

C

HTTP-Required Preload List Website

D

HTTP-Required Preload List API

HTTP Required Preload

httprequired.org/api/v1/

HTTP Required Preload List API Version 1 Overview

This is version 1 of our API. Below you'll find examples of different api calls.

Endpoints

- **/list**:

  Returns a list of all published versions of the HTTP Required Preload List.

  Example:

  /api/v1/list

- **/list/:version**:

  Returns that version of the HTTP Required Preload List, latest can be used to specify that you want the latest version, list can optionally be prettied and documented with comments.

  Example:

  /api/v1/list/{{version}|latest}?pretty={prettied}

- **/preloadable**:

  Returns whether a domain can at this point in time be added to the HTTP Required Preload List.

  Example:

  /api/v1/preloadable?domain={name}

  ```
  {
      "name": "example.com",
      "include_subdomains": true
  }
  ```

  Possible HTTP response codes:

  - 200 - OK (if domain can be preloaded)
  - 409 - Conflict (if domain is invalid or already preloaded)
  - 500 - Internal Server Error (for other errors)

- **/submit**:

  Submits a domain for entry to the HTTP Required Preload List.

  Example:

  /api/v1/submit?domain={name}

  ```
  {
      "name": "example.com",
      "policy": "requested",
      "include_subdomains": true
  }
  ```

  Possible HTTP response codes:

  - 200 - OK
  - 409 - Conflict (if domain is invalid or already preloaded)
  - 500 - Internal Server Error (for other errors)

- **/status**:

  Returns the status of a domain on the HTTP Required Preload List.

  Example:

  /api/v1/status?domain={name}

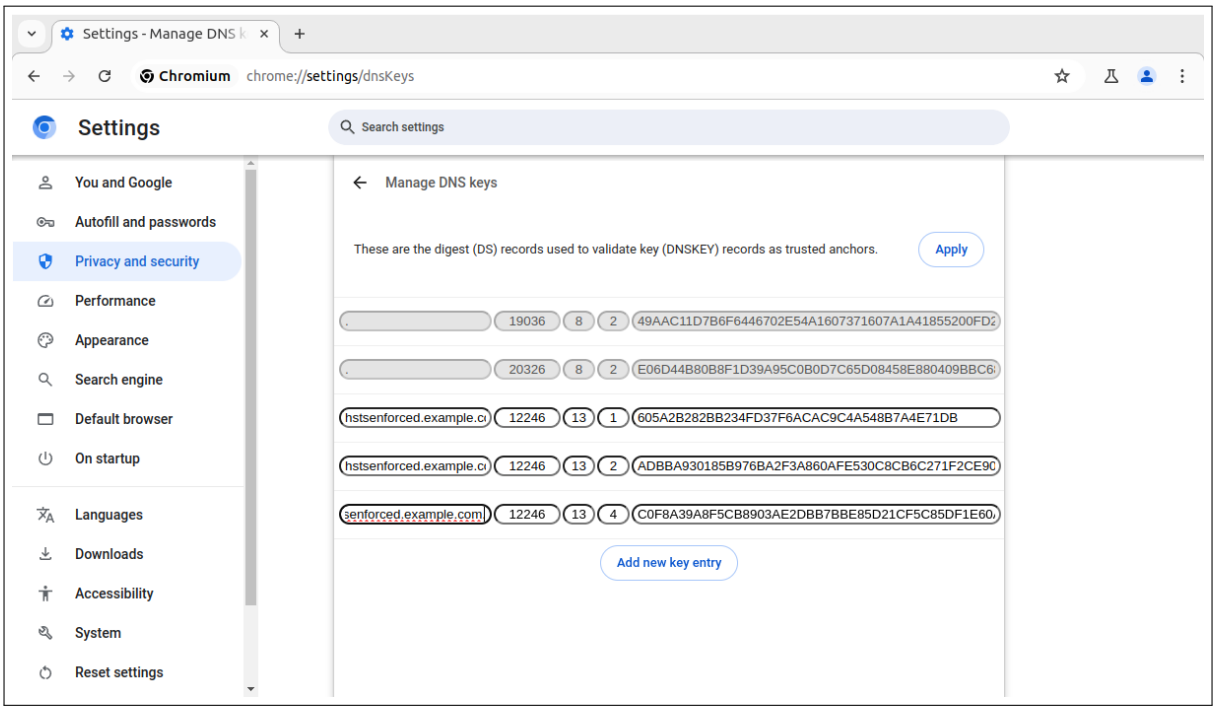
  ```
  {
      "name": "example.com",
      "policy": "requested",
      "include_subdomains": true
  }
  ```

  Possible HTTP response codes:

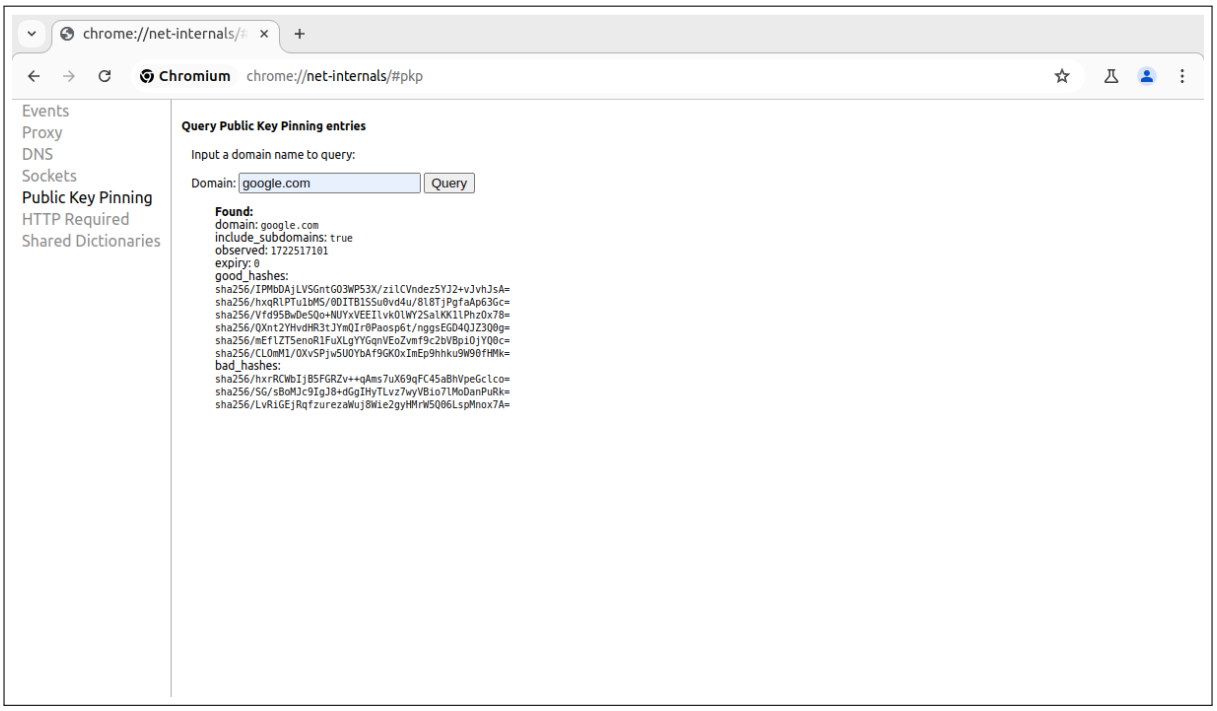
  - 200 - OK
  - 404 - Not Found (if domain not found)
  - 500 - Internal Server Error (for other errors)

E

DNS Key Manager Screenshot

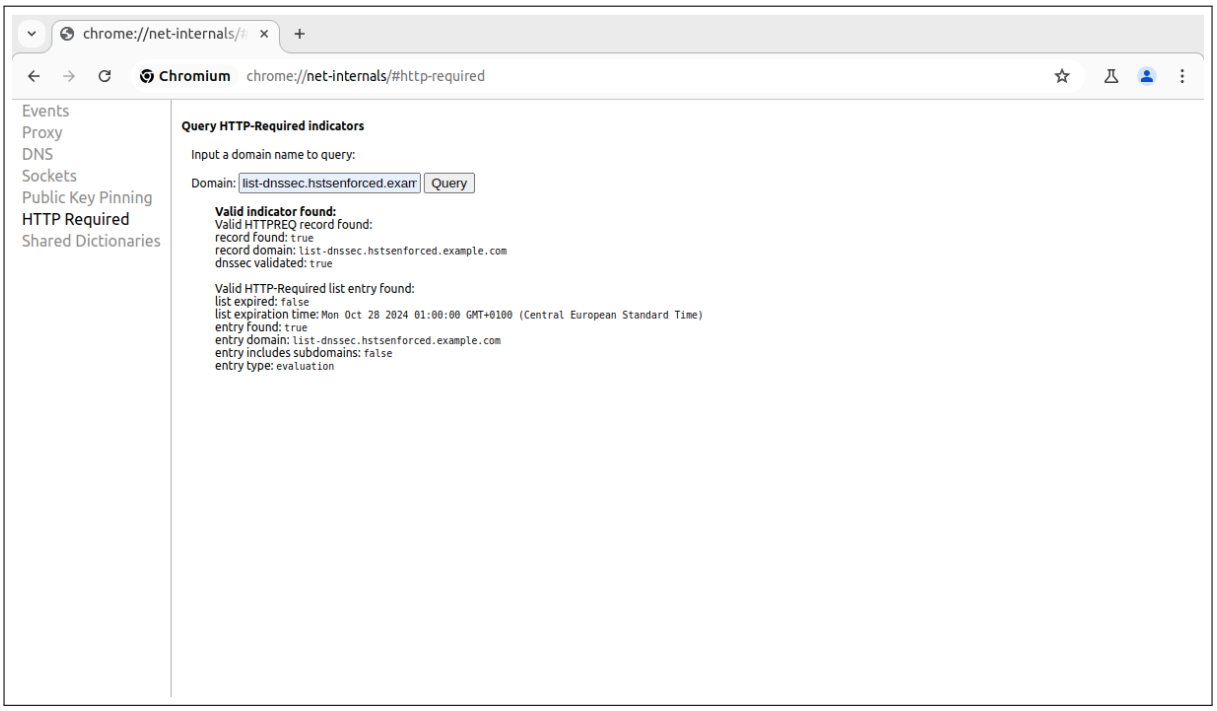


Public Key Pinning Debug Screen



G

HTTP-Required Debug Screen



H

HSTS Flags Screenshot