Survey of Web Security Mitigations in Modern Browsers

1st Ludvig Öberg
Linköping University
Linköping, Sweden
ludob894@student.liu.se

2nd Tommy Johansson
Linköping University
Linköping, Sweden
tomjo891@student.liu.se

3rd Ulf Kargén
Department of Computer and Information Science
Linköping University
Linköping, Sweden
ulf.kargen@liu.se

***Abstract*—Web applications have a growing importance in the modern society. These applications are built on the HTTP protocol. Many web applications have vulnerabilities that modern browsers aim to mitigate through security extensions to HTTP. In this paper a survey of these mitigations are presented. This report prioritizes a subset of the mitigations based on the effectiveness of the mitigation, the ease of implementation as well as the severity and prevalence of the vulnerability it mitigates, and describe these in detail.**

In this report the selection of vulnerabilities was based on *The Ten Most Critical Web Application Security Risks* report published by OWASP. The vulnerabilities are ranked based on the four criteria presented in OWASPs report: Exploitability, Prevelance, Detectability and Technical Impact. Similarly mitigations were selected from *Mozilla's Web Security Guidelines* which ranks web application mitigations based on security benefits and ease of implementation.

I. Introduction

The number of websites has grown rapidly since the birth of the modern internet in 1991, and there are currently 1,7 billion websites[1]. These websites provide services ranging from streaming movies and music, to reading newspapers and downloading applications. These are all different types of web application and it is a common occurrence that they are insecure. As the internet becomes a larger part of many companies operations, web applications is becoming an important infrastructure which needs to be defended against attackers. For example in 2019 156 billion USD was spent on cyber-security[2]. The industry is set to grow at an annual rate of 10%[2]. This is partly due to stricter regulations around customer data such as GDPR.

In 1994 Netscape created SSL 1.0 which was a secure protocol by with two networked peers could encrypt communications[3]. The adoptions of HTTPS has been slow but in recent years it has increased in popularity, currently 71,5% of all websites have HTTPS as their default protocol[4]. HTTPS is built on HTTP, which means that many of the same security problems that are present in HTTP are also present in HTTPS. It is up to the developer to implement mitigations to these HTTP Web Application vulnerabilities. In order to remedy common vulnerabilities many modern browsers have also implemented security extensions to HTTP.

This paper aims to survey these HTTP/1.1 mitigations. Selecting the most important mitigations based on the effectiveness of the mitigation as well as the severity and prevalence of the vulnerability, and describe the highest prioritized mitigations in more detail. The selection of vulnerabilities were based on *The Ten Most Critical Web Application Security Risks* report which is published by OWASP, a non-profit for better IT-Security. Mitigations were selected using *Mozilla's Web Security Guidelines* which ranks web application mitigations and is created by the Mozilla Foundation, which create and maintain the Mozilla Firefox web-browser.

A. Delimitation

This paper does not cover mitigations implemented in the web application source code, instead this paper will focus on security extensions to HTTP. The type of vulnerabilities that can be mitigated by HTTP security extensions are those that interact directly with the HTML document and resource policies for the web application. The types of vulnerabilities that cannot be mitigated by HTTP security extension are the ones that involve the web application logic, and the storage and retrieval of data.

Not all mitigations will be covered in depth, instead the mitigations that offer the most protection against the most severe vulnerabilities will be prioritized.

II. Background

In this section the background of HTTP's development and features are presented.

A. HTTP

HTTP was developed by Tim Berners-Lee and his team at CERN during 1989-1991[5]. The protocol evolved from a file exchange protocol to now being the underlying protocol of the modern internet[5]. The first version (later called 0.9) was simple with only one request, GET which retrieved a resource based on the path sent. HTTP version 1.0 was used between 1991-1995, with version HTTP/1.1 being the version most used and the leading protocol between 1999-2015 when HTTP/2 was released. HTTP/1.1 contains the methods OPTIONS, GET, HEAD, POST, PUT, DELETE, TRACE and CONNECT.

As shown in Figure 1 each HTTP request contains a request header and a request body. The request header contains a method, path and version of the protocol.

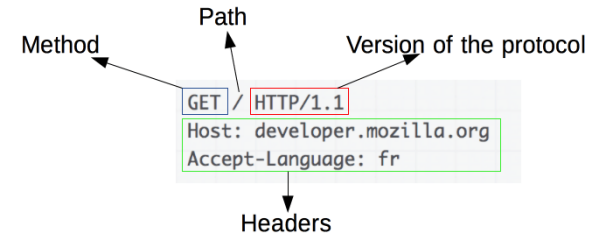

Fig. 1. HTTP Request parts[6]

B. HTTP Header Fields

HTTP headers is a way for the client and server to send additional information between each other. This happens using a HTTP request and HTTP response[7]. The header contains four different fields: General-header, Request-header, Response-header and Entity-header[8]. General headers are used to send requests and responses but with no connection to the data sent in the body[7]. Request headers contain information about the resources being fetched or information about the source requesting information[7]. Response headers contain information about the response such as information about the origin of the response[7]. Entity headers contain contents regarding the body of the response, this can be information such as content type or length[7]. Each field contains a name followed by a ":" and then the value of the field[8].

1) HTTP Security Extensions: HTTP security extensions comes in the form of HTTP response headers that is returned as a response from a HTTP request[8], instructing the browser how to behave when loading the website and its resources[9]. When it comes to security, the more important fields are the ones that handles scripts and what resources to allow[9].

2) Directives: Some HTTP headers have more than one operational mode. Directives instruct on which operational mode to use for that header. These directives are defined by the header and handle one specific task for the header and can therefore be used in conjunction with other directives for that header[8]. Directives can either enforce certain aspects of a header or relax its restrictions[8].

C. HTTPS

HTTPS is the protocol to use TLS to secure a HTTP connection. A connection is initiated via a TLS handshake[10]. Once the handshake is finished the client can then send the first HTTP request, all HTTP data is sent over TLS. TLS was previously called SSL and is based on asymmetric public key infrastructure[11]. By encrypting traffic HTTPs prevents anyone from snooping on network traffic even if the packets are intercepted[11]. HTTPs uses the URI 'https' instead of 'http' and the domain needs to match that on the certificate presented. If it does not the client needs to present an error page or close the connection[10].

III. Method

There are many vulnerabilities in web applications and many mitigations are done only in the web application's source code[12]. It is therefore necessary to do a prioritization of the currently most prevalent and severe vulnerabilities and limit this prioritization list to the most relevant for this survey. Each vulnerability and mitigation will be presented in order based on the ranking system described below.

A. Vulnerability and Mitigation Prioritization

In order to prioritize the vulnerabilities for this paper it is not sufficient enough to just pick the most prevalent and severe vulnerabilities. The vulnerability must have a mitigation in a security extension to HTTP, which is not always the case for the OWASP Top 10 vulnerabilities in Web Applications[12]. In order to find relevant HTTP security extensions and the vulnerabilities they mitigate, Mozilla's Web Security Guidelines[13] was used. These mitigations were prioritized based on security benefit and ease of implementation.

Vulnerabilities were selected based on the OWASP Top 10 Web Application Security Risks 2013[14] and 2017[12]. OWASPs rank these vulnerabilities on a scale of 1-3 on four different aspects; exploitability, weakness prevalence, weakness detectability and technical impacts[12]. These aspects are added together to produce the final ranking[12]. This paper will add the number given by OWASP for the four aspects together to create an overall ranking number, ranging from 0-12. This ranking system takes into account both the likelihood of an attack succeeding and the damage of the potential attack[12].

Mitigations were selected based on Mozilla Web Security Cheat Sheet[13], which ranks the mitigations based on their security benefit and implementation difficulty[13]. Each mitigation is given a score on the two aspect ranging from Maximum, High, Medium and Low[13]. To give the mitigations an overall ranking, they are assigned a value from 1-4 for each of the aspects which is then added to produce the ranking. This gives a final ranking value between 1-8. Which means that the mitigation that produces the highest security benefit and which is the easiest to implement ranks at the highest.

B. Paper Selection Method

After identifying which vulnerabilities and mitigations that were relevant for this report, searches on Google Scholar was done to identify relevant sources and scientific articles for each vulnerability and mitigation. Beyond scientific articles, a range of industry guidelines and documentation was used.

IV. Survey Overview

In this section an overview of the currently most prevalent and severe vulnerabilities as well as the most effective mitigation are presented in an ordered list. The list is then limited to vulnerabilities and mitigations that are within the scope of this paper.

Vulnerability	Exploitability	Prevalence	Detectability	Technical Impact	Total Score
Injection	3	2	3	3	11
Security Misconfiguration	3	3	3	2	11
Cross-Site Scripting	3	3	3	2	11
Sensitive Data Exposure	2	3	2	3	10
XML External Entities	2	2	3	3	10
Broken Authentication	3	2	2	3	10
Using Components with known vulnerabilities	2	3	2	2	9
Cross-Site Request Forgery	2	2	3	2	9
Broken Access Control	2	2	2	2	9
Insufficient Logging & Monitoring	2	3	1	2	8
Insecure Deserialization	1	2	2	3	8
Clickjacking	-	-	-	-	-

TABLE I
RANKING AND RANKING CRITERIA FOR EACH VULNERABILITY.

A. Web Application Vulnerabilities

In this section the currently most prevalent and severe web application vulnerabilities are listed based on prevalence and severity of the exploits they enable, see Table I. The vulnerabilities are presented in order of severity based on the ranking of *The Ten Most Critical Web Application Security Risks* from 2017[12] and 2013[14], combined with two additional vulnerabilities that are not among the top 10 but still relatively prevalent and severe. Each vulnerability is described briefly and what consequences the vulnerability might cause. The ranking and the ranking criteria for each vulnerabilitiy is presented above, see table I.

1) Injection: Injection covers vulnerabilities that exploit different injection methods such as SQL, OS, NoSQL and LDAP Injections. An injection is when data is sent to a server which interprets it as a query or a command which can trick the server into executing commands or accessing data[12].

2) Security Misconfiguration: The Security Misconfiguration vulnerability involves using more permissive configuration than necessary for the web application. This can be the use of more network ports than what the web application uses, old unused pages that lack updates and full print-outs of stack traces when a component on the web application crashes. These misconfigurations could be used to gain unauthorized access or knowledge of the system[12].

3) Cross-site Scripting (XSS): Cross-site Scripting (XSS) is a vulnerability which allows arbitrary JavaScript-code execution in the domain of the web application run on the browser of the user. There exists three variations of XSS; reflected, persistent and DOM-based. Through XSS it is possible to obtain user's session cookies and credentials to name a few[12].

4) Sensitive Data Exposure: Sensitive Data Exposure involves the exposure of sensitive data such as healthcare, financial or other personal data. This data can be used to commit identity theft and other crimes.

This vulnerability involves poorly secured sensitive data. This can be due to lack of or weak encryption, leaked passwords or failure to protect against compromises of data. The data can either be compromised in transit or at rest[12].

One aspect of Sensitive Data Exposure is that data should not be sent unencrypted and this can be achieved by enforcing HTTPS. Thereby disabling the possibility of accessing the data unencrypted.

5) XML External Entitites (XXE): XML Eternal Entities (XXE) vulnerabilities exists in web applications that accepts XML-documents from untrusted sources, either through uploads or directly linked. The XXE vulnerability enables data extractions, mapping of the internal network as well as exploitation of vulnerable code[12].

6) Broken Authentication: Broken Authentication is a vulnerability where general authentication policies for a web application is misconfigured. This can involve session tokens that last much longer than needed, as well as allowing dictionary attacks on the web application. One major flaw is having default admin login credentials and showing the session ID in the url[12].

7) Using Components With Known Vulnerabilities: When developing a web application many different frameworks and components are used. Over time, these components and frameworks could be discovered to contain severe vulnerabilities and if not continuously checked could act as a way to attack the web application. Not knowing which components and framework are in use in the web application is a great security risk[12].

8) Cross-site Request Forgery (CSRF): Cross-site Request Forgery (CSRF) is an vulnerability where the attacker tricks a user into sending a HTTP request without the users knowledge which bypasses the authentication methods. This can be done through social engineering or through executing scripts in a web application. This attack can be achieved by create a fake link, form or image which when interacted with sends a request on the users behalf. The attacker can force a user into performing a request to transfer funds, change their password etc[14].

9) Broken Access Control: Broken access control involves poorly enforced restrictions on what authenticated users are allowed to do. An attack can use this to access unauthorized data, retrieving data from other users accounts or changing access rights. This can for example involve accessing an URL that only admins should have access to[12].

10) Insufficient Logging & Monitoring: This security risk is due to insufficient logging and monitoring combined with no or poor incident response. This enables the attacker to

move further into systems and tamper or destroy data. Often companies do not know that they have had an intrusion until several months after the initial attack, in 2016 the average was 191 days between an attack was started and when it was found out [12].

11) Insecure Deserialization: Insecure deserialization can be used to perform several different attacks such as injection attacks, replay attacks and privilege escalation attacks as well as general remote code executions. These flaws are hard to discover and often need human assistance to identify and validate the issues[12].

12) Clickjacking: When a web application does not restrict UI Layers or frame objects from other domains or applications, the user can be lead to interact with something different than what is displayed on the web application[15, 16]. This vulnerability is called iFrame Overlay, or Clickjacking, and result in the user undesirably executing a task for which the user is authenticated for[17, 18].

B. Mitigations

In this section common mitigations for web applications are listed based on the selection from *Mozilla's Web Security Guidelines*[13]. These mitigations are briefly described and ranked based on their security benefit, and ease of implementation, see Table II,

Mitigation	Security Implication	Ease of implementation	Total Score
Resource Loading	4	4	8
HTTP Redirect	3	4	7
X-Frame Option	3	4	7
HSTS	3	4	7
CSP	3	2	5
Cookies	3	2	5
Subresource Integrity	2	3	5
Referrer Policy	1	4	5
X-Content-Type	1	4	5
X-XSS Protection	1	3	4
Public Key pinning	1	1	2
CSRF Tokenization	3	-	-

TABLE II
RANKING AND RANKING CRITERIA FOR EACH MITIGATION.

1) Resource Loading: This mitigation is about preventing resources from being loaded over insecure channels, for example JavaScript being loaded via http instead of https[13]. Attempts to load JavaScript insecurely on a HTTPS website will result in the browser blocking the resource[13]. This can lead to incomplete UI and a "mixed content" warning from the browser[13]. To prevent this from happening developers should make sure that all resources are loaded securely before deploying the website.

2) HTTP Redirections: Most websites listen on port 80 (HTTP) to connect clients than type url into the address bar, since all browser connect over HTTP for the first request. Web sites that listen on port 80 should immediately redirect the client to use HTTPS instead. Redirections should only be done within the domain to ensure that HSTS is set properly[13].

3) X-Frame Options: The HTTP header X-Frame Options allows sites to control how a site may be framed within a an iframe. This helps to prevent Clickjacking and is expected to be supported by all web sites to allow protection for browsers that do not support Content Security Policy (CSP). The recommended setting for X-Frame Options is to deny any external sites to framed within an iframe[13].

4) Strict Transport Security (HSTS): This mitigation tells the client to only connect to the web site through HTTPS even if the chosen protocol was HTTP. The HTTP Strict Transport Security (HSTS) header will transparently upgrade the request to HTTPS and tell the browser to handle TLS certificates more strictly[13]. This takes effect next time the client connect to web site again but HSTS has a directive to enable browsers to have a list of domains to enforce HSTS before the initial visit. This results in the browser never using HTTP for these domains and their subdomains[13].

5) Content Security Policy: This mitigation allows the developer of a website to have control over where resources on the site can be loaded from and is one of the best ways to prevent Cross-Site-Scripting[13]. Content Security Policy (CSP) is standard for all new websites and is highly recommended, especially for high risk websites[13]. CSP disables the use of unsafe inline JavaScript, preventing user input from being interpreted by a web browser as JavaScript[13]. One downside of CSP is that all JavaScript needs to be loaded from within a *script* tag and style attributes may fail to load.

6) Subresource Integrity: By ensuring the integrity of external JavaScript resource, hosted on content delivery networks, the mitigation prevents attacks against web site using this resource. The way this is done is by locking the known contents of JavaScript resource at a specific time, if this is changed in some way the browser will refuse to load it. This distinction is determined by a cryptographic hash[13].

7) Cookies: Cookies often contain sensitive information and session identifiers which could be collected in a Cross-site Scripting (XSS) attack. In order minimize the damage the cookies should be set to expire as soon as possible and to be forced to only be sent over HTTPS. By prepending *__secure-* you ensures that the cookies cannot be overwritten by untrusted sources. By making it forbidden to include cookies in cross-site requests it is a strong Cross-site Request Forgery (CSRF) prevention[13].

8) Referrer Policy: This mitigation improves the privacy of users by allowing for the control of when the referrer header is transmitted. This referrer header is used to inform the destination site of the origin of the request, for example when using a hyperlink or an external resource is loaded on a website. Referrer policy allows the developer to control how and when a HTTP referrer header is used[13].

9) X-Content-Type Options: This mitigation allows for the control of MIME sniffing, which protects against XSS when the browser incorrectly detect file types and then executes the files[19]. X-CONTENT-TYPE-Options allows the developer to set which MIME types of files that they use and explicitly only allow those, or even set the header to "nosniff" to disable MIME sniffing completely[13].

10) X-XSS-Protection: This mitigations prevents pages from loaded if a Cross-Site-Scripting attack is detected. This mitigation is enabled in Internet Explorer and Chrome though

it is mostly unnecessary, as a well implemented CSP provides similar protection. For users with older web browsers that do not support CSP, X-XSS-Proection can still be relevant.[13]

11) HTTP Public Key Pinning: This mitigation is mostly relevant for high risk sites. It works by binding a site to specific root certificate authority or end-entity public key, this prevents any certificate authority from issuing an unauthorized certificate for a given domain[13]. The certificate authority may issue a certificate maliciously or through an social engineering attack which this mitigation prevents. This mitigation further prevents Man-in-the-middle attacks or impersonations of a website by disabling any unauthorized certificate[13].

12) Cross-site Request Forgery Tokenization: A common mitigation against CSRF is the use of an unpredictable token for any sensitive actions on the web site. This token is then passed around in a cookie in a secure way and only allowing same-site requests[13].

C. Selected Vulnerabilities and Mitigations

In this section list the vulnerabilities that has mitigations in a HTTP security extension and the HTTP security extensions that mitigate one or more of these vulnerabilities.

The vulnerabilities that were in the scope of this survey are the following:

- Cross-site Scripting (XSS)
- Cross-site Request Forgery (CSRF)
- Clickjacking
- Sensitive Data Exposure

These vulnerabilities are mitigated by these security extensions:

- Content Security Policy
- Strict Transport Security (HSTS)
- X-Content-Type Options
- X-Frame Options

V. Vulnerabilities

This section will describe the different vulnerabilities that were selected in-depth in order of severity and prevalence.

A. Cross-site Scripting (XSS)

Cross-site Scripting (XSS) is a type of code injection attack where malicious JavaScript code is injected through an user-supplied input[20].

When performing an XSS attack the malicious JavaScript code is executed within the domain of the website which makes it possible to steal users cookies, and log users key strokes[20]. Since JavaScript is usually used to make website more dynamic it is possible to construct XSS-attacks that create fake login forms[20]. This is used to stage phishing attacks by exploiting the the fact that the website is treating the XSS-payload as part of the website[20]. There exists three different kinds of XSS-attacks, reflected, persistent and DOM-based XSS attacks[20].

A persistent XSS-attack exploits the websites ability to store user-generated data in a database which is visible to anyone on the website. Whenever the user-generated data is loaded the malicious JavaScript within that data is executed, resulting in a XSS-attack for any user who visits that part of the website[20]. A reflected XSS-attack exploits the ability to copy a link to the website where the XSS-attack has been performed, usually a link to a search result, and having the user click on that link[20]. When the user clicks the link the XSS-attack is carried out as the website execute the search request which include the XSS-payload, resulting in a XSS-attack on anyone who clicks the link[20].

DOM-based XSS-attacks are similar to reflected XSS-attacks in that it requires a user to click on a link that carries out the attack. What is different with DOM-based XSS-attacks is that the XSS-payload is placed in the url, not in the input field on the website[21, 22]. This is harder to do and harder for the website to detect but make it possible to execute XSS-attacks on websites that do not have any input fields, resulting in enabling XSS-attacks on arbitrary websites[22]. It is also possible to put a # in the url before the XSS-payload to stop the XSS-payload reaching the server[21, 22]. This makes the XSS-attack completely local making it even harder for the website to detect the attack[21, 22].

B. Sensitive Data Exposure

Data exposure is becoming more critical as there are more regulations and requirements such as EU's GDPR which could result in fines for companies who fail to protect their customers data. It is mitigated mostly through client-side changes and through salting and encrypting the database and through cryptographic key management[13]. It can be related to HTTP Security Extensions through HTTP Strict Transport Security (HSTS) as a way to mitigate man-in-the-middle attacks by enforcing encryption.

C. Cross-site Request Forgery (CSRF)

CSRF (Cross-Site-Request-Forgery) is an attack that enables the attacker to initiate arbitrary HTTP requests from the victim of the attack[23]

According to OWASP Top 10 - 2013 CSRF was ranked 8 based on that the exploitability was average, prevalence common, detection easy and impact moderate[14]. It was removed in the OWASP Top 10 - 2017 and added as an "honorable mention" as mitigations and preventions of the vulnerability has gotten better and it is no longer as large of a problem. CSRF can be mitigated using a SameSite configuration for cookies, which forbids the sending of cookies via cross-origin requests[13].

When performing a CSRF attack, if the victim of the attack is authenticated against a service, it could allow the attacker to bypass any underlying authentication mechanics[23]. This could enable the attacker to post messages, send emails or even change the login information of the user[23]. In comparison to many other attacks, this attack is not aimed at trying to steal the users session ID, instead this attack tries to exploit the fact that the web application can not differentiate between an intended user request and a request that an attacker tricked the user into doing[23]. This can for example be achieved by

an HTML form which assembles a POST request which is in turn sent by a short JavaScript code or the attacker can embed a request into an image which will automatically be executed when a user enters the page[23].

D. Clickjacking

Clickjacking is a type of web framing attack in which an iframe is used to hijack a user's web session[24]. By hijacking the users clicks the attacker can perform undesired actions which can benefit the attacker[25]. As the attack is performed using an invisible iframe which leaves the user unaware that their clicks are going to a malicious website that sits on top of the users page[25]. One example of a use of Clickjacking is to hijack a users click and forward it to a social media to like or follow pages, such as a like on a Facebook page, this is sometimes called "Likejacking"[26, 27].

This vulnerability is not in the OWASP Top Ten list for either 2017 or 2013, however it is mentioned in the section "additional risks to consider"[14]. Due to it's mention in the OWASP 2013 report and due to that the vulnerability is widely mitigated using HTTP Security Extensions, the vulnerability was included in the selection.

VI. Mitigations

This section will describe the different mitigations available, what vulnerabilities they mitigate and how they are implemented. The mitigations will be prioritized based on the effectiveness of the mitigation and the prevalence of the attack.

A. X-Frame Options

X-Frame options is one way of defending against Clickjacking attacks. This works by introducing a X-Frame-Options HTTP Response header to indicate that the browser should not allow the rendering of Iframe and similar page in frame solution[25]. X-Frame Options was first introduced in Internet explorer 8 and is an HTTP header sent on HTTP response, which has two different values either DENY or SAMEORIGIN[24]. When DENY is provided the browser does not render the requested site if it is within a frame, while in SAMEORIGIN only sites from the same origin as the rest of the site is rendered[24].

X-FRAME options is very effective at protecting the user from Clickjacking[24]. X-Frame option has several downsides such as that the policy need to be specified per page[24]. X-Frame options also struggles with multi-domain sites as current implementation does not allow for a "whitelist" of domains when using the SAMEORIGIN response[24]. X-FRAME option is also vulnerable to proxies, as proxies are known to strip headers which would mean that the user would lose their protection[24]. Another issue is that X-FRAME options was not standardized and were therefor implemented in several different ways leading to attacks of double framing against some browsers[28]. Certain browsers allow for a third value "ALLOW-FROM" which specifies the origin of which frames are allowed, this feature are not supported in Chromium browsers[28]. Despite these shortcomings X-Frame Options is one of the most used HTTP Headers, with 11% of the top 1 million websites using it in May 2017 [29].

B. Strict-Transport-Security (HSTS)

HSTS is a HTTP header which tells the browser to only connect to a site over HTTPS even if the intial connection was HTTP. Browsers that support HSTS will upgrade all requests from a site to HTTPS if that site sends a HSTS header. HSTS also tells the browser to handle TLS and certificate errors more strictly by disabling the user from bypassing the error page.[13]

HSTS can be used to prevent threats from bookmarks or manually entered addresses which uses HTTP instead of HTTPS or from webapplications accidentally containing HTTP links or content which is served through HTTP. HSTS will redirect the HTTP traffic to the HTTPS for the target domain of the request. HSTS can also prevent Man-in-the-middle attacks where the attack has an invalid certificate. If the user attempts to bypass the error page for the bad certificate, HSTS will not allow this and therefor negates the attack.[30]

Sensitive data can be protected using a HTTP header HTTP Strict Transport Security to ensure secure encryption in transit[12].

HSTS is implemented in 5,4 % of the top one million websites and is one of the most widely used HTTP headers[29].

C. Content Security Policy (CSP)

The Content Security Policy (CSP) is a HTTP response header that allows server admins to supply white-lists of trusted sources to be loaded on the web application. Any other resources that is not listed in the white-lists is blocked from loading, relieving the need to implementing a filtering of input which is prone to flaws[31].

The main goals of CSP is to mitigate XSS with as little modification of the source code of the web application. CSP even stops a loophole that exists in a typical web application security model, the same origin policy (SOP) that only permits scripts running on the same site[31]. Even if an attacker manages to inject a XSS payload onto a SOP-trusted site, the payload would be rejected if the site uses CSP and has not listed the resource the XSS attack used[31].

By default CSP blocks any use of inline scripts thus blocking XSS attacks that make use of inline scripts. CSP has many different directives to allow customization of what resources to allow on the web site. These include sources to load within the same origin, script sources to load, Cascading Style Sheets (CSS), fonts and images[31] etc.

One more directive is the *frame-src* directive which is a list of sources that are allowed to be rendered inside frames on the website. This mitigates Clickjacking since a list of trusted of sources to be rendered in frames are listed, such as Youtube for example, and all other sources are blocked[32].

CSP also has a report directive to save instances of web sites and resources that has violated the CSP directives of the web site has therefore been blocked from executing[31].

In May 2017 around 1,6 % of all Alexa top one million websites had implemented Content Security Policy headers making it one of the lesser used HTTP headers[29]. CSP is growing quickly in adoption as in August 2015 only 0,15 % of top one million websites had it implemented[29].

D. X-Content-Type Options

X-Content-Type-Options is a HTTP header which tells the browser if it should use MIME Sniffing or not[19]. MIME Sniffing vulnerability is a type of XSS where a user uploads content to a website but disguises the file as something else[19]. For example if a browser requests a media file and it is served with an incorrect media type. The browser will still detect and execute the file which may contain scripts that are also executed. This opens up for different XSS attacks[33]. The way that X-Content-Type-Options mitigates these types of script attacks is by passing on a header "nosniff" which tells the browser not to MIME Sniff and hence no scripts hidden in incorrect file types will be executed[33]. This is an easy mitigation to implement as all it requires is to activate the header, however it only protect against some XSS attacks[33]. In May 2017 around 10,5 % of all Alexa top one million websites had implemented X-Content-Type-Options headers making it one of the most used HTTP headers[29].

VII. Discussion

This section will discuss the different vulnerabilities and mitigations that exist in web applications.

A. Source Criticism

A discussion and evaluation of the sources use will also be presented here. This report primarily used scientific articles and proceedings from well known journals. Apart from these sources industry guidelines and documentation were used. For background information general statistics databases and books were used to place the topic in context.

Among our sources were *Mozilla's Web Security Guidelines*[13] which is web security documentation that is published under Creative Commons by the Mozilla Foundation a nonprofit organisation. Mozilla Foundation has deep industry knowledge about Web application security and development and can be considered a reliable source. However, they are the developers and owners of Mozilla Firefox which is a popular browser. This could incentivize Mozilla Foundation to publish mitigation rankings and information that favors Mozilla Firefox browser to give the appearance that their browser is more secure than their competitors. To ensure that the information presented in the report is unbiased, other sources, such as OWASP industry documentation, have been used to confirm the general classification of the mitigations presented by Mozilla.

OWASP Top Ten Security Risks[12] report from 2013 and 2017 were used to select and rank the vulnerabilities presented in this report. The Open Web Application Security Project (OWASP) is a nonprofit foundation founded in 2001 that works to improve software security[34]. OWASP does not have any incentive to misinform and can be considered a reliable source when it comes to web security.

To further improve the quality of the sources the information presented in the report could use source triangulation by having multiple sources to confirm the same information. This is especially important in the ranking and ordering of vulnerabilities and mitigations as this report is limited in its ranking methodology.

B. Method Criticism

The method currently used to rank and select vulnerabilities and mitigations are primarily based on the opinions of OWASP and Mozilla Foundation, which while reliable may be biased or misinformed. Ideally the method would involve independent data gathering from industry sources, to create a quantitative method for ranking the prevalence and impact of vulnerabilities as well as the security benefit and ease of implementation of mitigations. This is beyond the scope of this report but would produce a more certain ranking.

The selection of sources used for this report may have been biased as no large data gathering was made. Data was only gathered through Google Scholar and other relevant search engines, looking for relevant scientific articles, as well as industry guideline and documentation sources. It is possible that there are other sources that would provide more information on the topic that was not used or covered in this report. Similarly there could be vulnerabilities and mitigations that were not covered or prioritized in the report, that could be relevant, which have been missed.

The number of relevant mitigations through HTTP Security Extensions were limited as many HTTP headers are deprecated and many modern mitigations were implemented through source code. Some mitigations are so prevalent and effective that the vulnerability is no longer relevant, these mitigations are required for all websites and are implemented in almost all new websites. Due to the low number of relevant mitigations the prioritization was mostly based on the scope of the report instead of the ranking of the vulnerabilities and mitigations.

C. Future Studies

Future research could focus on creating a quantitative method which does not rely on the opinions of other sources but which analyses industry data to select and rank vulnerabilities and mitigations. Some aspects to include in such an analysis could be the prevalence of mitigations by looking at which headers are used in the top ranking websites in the world.

Gathering information about the prevalence and impact of vulnerabilities is more difficult as victims of attacks often do not publicly disclose the details of an attack, however it could be possible to conduct a survey to try to gather information from security experts. Using such a survey you could gather information regarding what vulnerabilities and mitigations industry experts would prioritize. Alternatively a survey of large corporations could be made to try to identify

which vulnerabilties are exploited to perform attacks in the real world.

Another avenue of further research would be to conduct an analysis of the most used browsers and compare the mitigations offered by these browsers to the list of mitigations presented in this report. HTTP Security Extensions are implemented differently in each browser which offers different protection and different vulnerabilities, it would be valuable to have an overview on which browsers offers what protection.

One topic of further research would be to look at different implementation mistakes or misconfigurations, while many mitigations offer sufficient protection against vulnerabilities if configured correctly. If they are misconfigured, they often offer no protection or even open up new vulnerabilities. It would be interesting to see a report which takes the listed mitigations in this report and surveys the top websites for misconfigurations of these mitigations and presents the consequences of these misconfigurations.

VIII. Conclusion

HTTP Security Extensions are still relevant and some of the most prevalent and impactful vulnerabilities can be mitigated using HTTP Headers. Even though many mitigations are implemented in source code, HTTP Security Extension can often be an easier way of mitigating vulnerabilities. For example Cross-site Scripting can be mitigated using the powerful tool of Content Security Policies. With the correct configuration, there may not be a need for any mitigations in the source code, instead you can rely on the CSP to protect the website against XSS attacks. However as old browsers often do not support CSP, it is still industry standard to use both CSP and mitigations in the source code, to provide a full coverage mitigation.

Unfortunately several of the mitigations prioritized in this report can be difficult to implement correctly and requires understanding of how the headers work. This initial difficulty may be part of why developers shy away from mitigating attacks through HTTP Security Extensions and instead opt for the familiar and safe way of implementing mitigations in source code. The best outcome would be to use both alternatives to offer the best protection possible and prevent vulnerabilities in the case of misconfiguration of either security extension or mitigations in the source code.

One example of a mitigations that is simple to implement yet powerful is X-Frame-Options which is another HTTP Header that can prevent Click-jacking attacks and is considered mandatory by many. X-Frame-Options is by some measures the most implemented mitigation using HTTP headers and this may be due to the ease of implementation.

In summary HTTP Security Extensions are often neglected, but they are an important tool in the mitigations against web attacks and should receive more attention from developers. With the correct HTTP Header configurations many of the modern vulnerabilities can be mitigated. The industry should work to make HTTP Security Extensions more available to developers and help spread the knowledge about this often forgotten way of mitigating vulnerabilities.

Sources:

[1] Martin Armstrong. Infographic: How many websites are there?, Oct 2019. URL https://www.statista.com/chart/19058/how-many-websites-are-there/.

[2] Cyber security market size & share report, 2020-2027, Jun 2020. URL https://www.grandviewresearch.com/industry-analysis/cyber-security-market.

[3] Colin Walls. *Introduction to SSL*, page 344–344. Elsevier, 2006.

[4] Usage statistics of default protocol https for websites, Apr 2021. URL https://w3techs.com/technologies/details/ce-httpsdefault.

[5] Evolution of http, 2021. URL https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/Evolution_of_HTTP.

[6] An overview of http, 2021. URL https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview.

[7] Web technology for developers - http headers, 2021. URL https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers.

[8] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – http/1.1. rfc 2616 (standards track), 1999. URL https://www.ietf.org/rfc/rfc2616.txt.

[9] Lilyana Petkova. Http security headers. *Knowledge International Journal*, 30(3):701–706, 2019.

[10] Eric Rescorla. Http over tls, 2000. URL https://tools.ietf.org/html/rfc2818t.

[11] Clourflare Inc. What is https?, 2021. URL https://www.cloudflare.com/learning/ssl/what-is-https/.

[12] The Open Web Application Security Project (OWASP) foundation. Owasp top 10, 2017. URL https://owasp.org/www-project-top-ten/2017.

[13] Mozilla. Mozilla web security guidelines, 2021. URL https://infosec.mozilla.org/guidelines/web_security.

[14] The Open Web Application Security Project (OWASP) foundation. Owasp top 10, 2013. URL https://owasp.org/www-pdf-archive/OWASP_Top_10_-_2013.pdf.

[15] PLOVER. User interface (ui) misrepresentation of critical information (cwe-451), 2006. URL https://cwe.mitre.org/data/definitions/451.html.

[16] CWE Content Team. Improper restriction of rendered ui layers or frames (cwe-1021), 2017. URL https://cwe.mitre.org/data/definitions/1021.html.

[17] CAPEC Content Team. Clickjacking (capec-103), 2014. URL http://capec.mitre.org/data/definitions/103.html.

[18] CAPEC Content Team. iframe overlay (capec-222), 2014. URL http://capec.mitre.org/data/definitions/222.html.

[19] X-content-type-options http header - keycdn support, 2018. URL https://www.keycdn.com/support/x-content-type-options.

[20] Shashank Gupta and Brij Bhooshan Gupta. Cross-site scripting (xss) attacks and defense mechanisms: classification and state-of-the-art. *International Journal of System Assurance Engineering and Management*, 8(1): 512–530, 2017.

[21] Germán E Rodríguez, Jenny G Torres, Pamela Flores, and Diego E Benavides. Cross-site scripting (xss) attacks and mitigation: A survey. *Computer Networks*, 166:106960, 2020.

[22] Amit Klein. Dom based cross site scripting or xss of the third kind. *Web Application Security Consortium, Articles*, 4:365–372, 2005.

[23] Nenad Jovanovic, Engin Kirda, and Christopher Kruegel. Preventing cross site request forgery attacks. In *2006 Securecomm and Workshops*, pages 1–10. IEEE, 2006.

[24] Gustav Rydstedt, Elie Bursztein, Dan Boneh, and Collin Jackson. Busting frame busting: a study of clickjacking vulnerabilities at popular sites. *IEEE Oakland Web*, 2(6), 2010.

[25] D Kavitha and S Ravikumar. Click jacking vulnerability analysis and providing security against web attacks using white listing url analyzer. *International Journal of Computer Techniques*, 2015.

[26] Lin-Shung Huang, Alex Moshchuk, Helen J Wang, Stuart Schecter, and Collin Jackson. Clickjacking: Attacks and defenses. In *21st {USENIX} Security Symposium ({USENIX} Security 12)*, pages 413–428, 2012.

[27] Stefano Calzavara, Sebastian Roth, Alvise Rabitti, Michael Backes, and Ben Stock. A tale of two headers: A formal analysis of inconsistent click-jacking protection on the web. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 683–697. USENIX Association, August 2020. ISBN 978-1-939133-17-5. URL https://www.usenix.org/conference/usenixsecurity20/presentation/calzavara.

[28] Stefano Calzavara, Sebastian Roth, Alvise Rabitti, Michael Backes, and Ben Stock. A tale of two headers: a formal analysis of inconsistent click-jacking protection on the web. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, pages 683–697, 2020.

[29] William Buchanan, Scott Helme, and Alan Woodward. Analysis of the adoption of security headers in http. *IET Information Security*, 12, 10 2017. doi: 10.1049/iet-ifs.2016.0621.

[30] Http strict transport security cheat sheet¶, 2021. URL https://cheatsheetseries.owasp.org/cheatsheets/HTTP_Strict_Transport_Security_Cheat_Sheet.html.

[31] Imran Yusof and Al-Sakib Khan Pathan. Mitigating cross-site scripting attacks with a content security policy. *Computer*, 49(3):56–63, 2016.

[32] Sid Stamm, Brandon Sterne, and Gervase Markham. Reining in the web with content security policy. In *Proceedings of the 19th international conference on World wide web*, pages 921–930, 2010.

[33] Use ‘x-content-type-options‘ header, 2021. URL https://webhint.io/docs/user-guide/hints/hint-x-content-type-options/.

[34] About the owasp foundation, 2021. URL https://owasp.org/about/.