Session Armor:

Protection Against Session Hijacking using Per-Request Authentication

A Thesis

Submitted to the Faculty

of

Drexel University

by

Andrew J. Sauber

in partial fulfillment of the

requirements for the degree

of

Master of Science in Computer Engineering

September 2017

© Copyright 2017
Andrew J. Sauber. All Rights Reserved.

Dedications

This work is dedicated to

Amy
Michael
Andrew
James
Sijia
Daniel
Kyla

Acknowledgements

This work would not be possible without the support, insight, and patience of my advisor, Dr. Harish Sethu. I'm repeatedly inspired by his depth of knowledge and the work that he does for his students and for the world.

I would like to also thank Dr. Baris Taskin, Dr. Mark Hempstead, and all of my educators throughout the years.

Table of Contents

List of Tables vii

List of Figures viii

Abstract x

1. Introduction 1

    1.1 Overview 2

    1.2 Cookie-backed Sessions 4

    1.3 Motivation – The Session Hijacking Attack 5

        1.3.1 Session Sidejacking 6

        1.3.2 Cross-site scripting (XSS) 7

        1.3.3 Session Fixation 8

        1.3.4 Physical Access 8

        1.3.5 Cross-Site Request Forgery (CSRF) 9

    1.4 The Session Extension Attack 10

    1.5 Rogue Browser Extensions 11

2. Observing Session Hijacking of Existing Web Applications 13

    2.1 SessionJack 13

        2.1.1 Analysis of Bearer Token Vulnerabilities 14

        2.1.2 Analysis of Bearer Token Lifetime 20

    2.2 JackHammer 21

3. Existing Session Hijacking Mitigation Techniques 26

    3.1 Standards for Session Token Protection 26

        3.1.1 HTTPOnly 26

        3.1.2 Secure 26

        3.1.3 Expiration Time 27

3.1.4 HSTS 27

3.1.5 HTTP Digest Authentication 28

3.2 Proposed Methods for Session Protection 30

3.2.1 Secure Cookie Protocol 30

3.2.2 SessionLock 31

3.2.3 Web Key 32

3.2.4 HTTPI 33

3.2.5 One Time Cookies 34

3.2.6 SecSess 36

3.3 SessionArmor Compared to Existing Methods 38

4. The Session Armor Protocol 40

4.1 Goals of the Protocol 40

4.2 Features and Mitigation Techniques 41

4.2.1 Fully Specified and Configurable HMAC 41

4.2.2 Secure Storage of HMAC key 42

4.2.3 Careful Choice of Cryptographic Primitives 42

4.2.4 Time-based Replay Prevention 43

4.2.5 Nonce-based Replay Prevention 44

5. Formal Specification of the Session Armor Protocol 45

5.1 Syntax of Parameter Specifications 45

5.2 Formatting Conventions for HTTP Headers 46

5.3 Setup Phase 47

5.3.1 Client Setup Phase 47

5.3.2 Server Setup Phase 49

5.4 Session Phase 60

5.4.1 Client Session Phase 60

5.4.2 Server Session Phase 63

5.5 Session Invalidation Phase 69

5.6 On the Use of HMAC-SHA3 71

6. Formal Verification of the Session Armor Protocol 72

6.1 Proof of Secrecy Using ProVerif 75

6.2 Proof of Authentication Using ProVerif 75

7. Reference Implementation of Session Armor 78

7.1 Server Implementation as a Django Middleware 78

7.1.1 Exceptions 79

7.2 Client Implementation as Google Chrome Extension 82

7.3 Performance Evaluation 83

8. Conclusions 93

Appendix A. Session Armor 97

A.1 Server Reference Implementation, Django Middleware Source Code 97

A.2 Client Reference Implementation, Chrome Extension Source Code 115

Appendix B. SessionJack Data 125

Appendix C. Formal Verification 128

C.1 Proverif Model 128

C.2 Proverif Verification Output 131

Bibliography 135

List of Tables

2.1 Domain Susceptibility to Session Hijacking 15
2.2 Mitigation of Session Sidejacking by Alexa Rank 15
2.3 Mitigation of Cross-Site Scripting by Alexa Rank 18
5.1 Hashing Algorithm choices in Client Support Bitmask 48
7.1 Session Creation Time Performance (ms) 85
7.2 HTTP Server – Application Request Performance (ms) 87
7.3 SessionArmor Server – Application Request Performance (ms) 87
7.4 SessionArmor Client – Application Request Performance (ms) 87
B.1 Domain Susceptibility to Session Hijacking 125

List of Figures

1.1 Bearer Tokens in the form of HTTP Cookies. The current practice for authentication of web requests 5

1.2 Warning dialog for installing a browser extension with unlimited cookie permissions 12

1.3 Warning dialog for visiting a domain with an expired TLS certificate 12

2.1 Number of sites that are vulnerable to a Session Hijacking attack via Cross-Site Scripting, Sidejacking, and Bearer Token Extraction 14

2.2 Alexa rank for websites which mitigate Session Sidejacking and those that don't 16

2.3 Alexa rank for websites which mitigate Cross-Site Scripting and those that don't 17

2.4 Token lifetimes vs site popularity for each type of hijacking protection . . . 21

2.5 Chrome (left) with logged-in session. Firefox (right), with no logged-in session 23

2.6 The JackHammer user interface, a test for Cross-Site Scripting vulnerabilities has been initiated 24

2.7 The JackHammer test yields a positive result, Firefox now has a logged-in session using an exfiltrated token 25

3.1 Username and password dialog box for HTTP Digest Authentication . . . 29

5.1 Client ready header in un-encoded and base64 notation 49

5.2 Proxies may modify or append to the headers of a client request 56

5.3 Server header for the establishment of a new session in un-encoded and base64 notation 57

5.4 Example HMAC input and output, with key and output encoded in hexadecimal notation. SHA-512 is in use 62

5.5 Client header for one request in un-encoded and base64 notation 64

5.6 HMAC of last-request-time enables a stateless inactivity timeout 66

5.7 All possible cases of evaluating a request nonce for replay prevention purposes 68

5.8 Invalidation header first showing parameters and then HMAC result. . . . 69
5.9 Example run of the SessionArmor protocol 70
7.1 To-Do application using SessionArmor live on the Internet 84
7.2 Session creation times for SessionArmor server 85
7.3 Session creation times for SessionArmor client 85
7.4 Session creation times for HTTP-server 86
7.5 Application performance for HTTP Server 88
7.6 Application performance for SessionArmor server, full protocol 88
7.7 Application performance for SessionArmor server, no NBRP 89
7.8 Application performance for SessionArmor server, no Header Auth. 89
7.9 Application performance for SessionArmor server, no TBRP 90
7.10 Application performance for SessionArmor client, full protocol 90
7.11 Application performance for SessionArmor client, no NBRP 91
7.12 Application performance for SessionArmor client, no Header Auth. 91
7.13 Application performance for SessionArmor client, no TBRP 92

Abstract

Session Armor:
Protection Against Session Hijacking using Per-Request Authentication
Andrew J. Sauber
Dr. Harish Sethu

Modern life increasingly relies upon web applications to provide critical services and infrastructure. Activities of banking, shopping, socializing, entertainment, and even medical record keeping are now primarily conducted using the Internet as a medium and HTTP as a protocol. A critical requirement of these tools is the mechanism by which they authenticate users and prevent transaction replay. Despite more than 20 years of widespread deployment, the de-facto technique for accomplishing these goals is the use of a static session *bearer token* to authenticate all requests for the lifetime of a user session. In addition, the use of any method to prevent request replay is not in common practice. This thesis presents Session Armor, a protocol which builds upon existing techniques to provide cryptographically-strong *per-request* authentication with both *time-based* and optional *absolute* replay prevention. Session Armor is designed to perform well and to be easily deployed by web application developers. It acts as a layer on top of existing session tokens, so as not to require modification of application logic. In addition to Session Armor, two additional tools are presented, JackHammer, a cross-browser extension that allows developers to quickly discover session hijacking vulnerabilities in their web applications, and SessionJack, a tool for analyzing the security properties of session tokens found on the web. A formal specification of the Session Armor protocol is provided. An implementation of the protocol is included as a Python Django middleware and a Chrome browser extension. Performance data is provided with a comparison to previous methods. A formal validation of secrecy and correspondence properties is presented in the Dolev-Yao model.

1. Introduction

Fundamental activities in society are now inextricably linked to the world wide web. In fact, for many people they are synonymous with its use: shopping, banking, catching up with friends and family, dating, grocery shopping, and even medical service now often begin online. However, HTTP, conceived as a networked document retrieval system, was not designed to be used in some of these sensitive contexts. The protocol developed at CERN for simplifying dissemination of scientific data is now used as the means for interacting with applications responsible for authenticating and authorizing these critical activities. In the original HTTP standard as implemented, there was no explicit capability or expectation for arbitrary data to be sent to the server by the user. [8] In fact, HTTP requests are defined as *idempotent* in this historic specification.

One feature that applications using HTTP wish to provide is the notion of a user session, whereby a user first provides some credentials as a means of authentication (usually a username and password), and is subsequently authorized to access and/or modify the state of the application for some period of time (the session length), or until the user logs off, to explicitly terminate the session. Unfortunately, there are a number of means by which an attacker can take control of user sessions, even when current state-of-the-art practices of session protection are in-use.

This thesis presents Session Armor, a protocol used to prevent session hijacking of web applications. It provides a mechanism for per-request authentication of a protocol-required set of data, in addition to application-specific data. Most importantly, it also provides for robust replay prevention using both a time-based and nonce-based method. The nonce-based method provides *absolute* replay prevention. Built into Session Armor is the enforcement of a session expiry deadline and inactivity-based expiry, to prevent a session

extension attack. A block cipher on the server-side and HMAC on both ends are used as cryptographic primitives to achieve these goals.

1.1 Overview

In chapter 1, we present the current state-of-the-art in HTTP session protection, Cookie-backed sessions and the related concept of bearer tokens. A description of techniques for Session Hijacking, malicious control of user sessions, are also presented. These techniques include Session Sidejacking, Cross-Site Scripting, Session Fixation, physical access, Cross-Site Request Forgery, and Rouge Browser Extensions. The Session Extension attack is also presented a means for a malicious party to increase the likelihood of a bearer token being obtained.

In chapter 2, two tools are presented that were built to observe Session Hijacking in the wild, SessionJack, and JackHammer. The first can be used to analyze browser-extracted cookie data for known vulnerabilities, and record session lifetimes. The second uses a web-socket server and two browser extensions to shuffle specified sets of unprotected cookie data between two web browsers. This tool can be used to instantly test for session token extraction vulnerabilities of live web applications. Over 100 sites were tested, with a wide range of popularity according to web rankings. It was found that over 31% left themselves open to Cross-Site Scripting, over 56% to Session Sidejacking, and 100% to bearer token extraction. The vulnerable sites included major financial institutions, e-commerce websites, business services, and social media web applications. For both vulnerability and session lifetime, there was no correlation with website popularity, indicating widespread under-utilization of the existing protections.

In chapter 3, we present existing techniques for mitigating the risk of Session Hijacking. First, five techniques which have been standardized in specifications ratified by the

IETF are presented. Next, six proposed HTTP authentication protocols which have appeared throughout the literature are presented. A number of weaknesses in these protocols are discussed, including vulnerability to Cross-Site Scripting (because of use of the use of the URL location to store secrets), weakened implementation due to incomplete specification of HMAC input data, the use of block cipher modes with known attack vectors, and undeployability due to the use of replay prevention techniques that rely on out-of-band parameters.

In chapter 4, an overview of the SessionArmor protocol is presented including the goals which motivate its design. These include goals of security, performance, deployability, and ease-of-use. The features and techniques which SessionArmor uses to distinguish itself from prior methods are also presented in this chapter. These are: fully specified configurable input data to the HMAC, secure storage of the the HMAC key, careful choice of cryptographic primitives, time-based replay prevention, and absolute replay prevention.

In chapter 5, a formal specification of the SessionArmor protocol is presented. An exact and detailed description of all data transferred between the client and server in each phase of the protocol is given. A syntax is used which clearly specifies required data, optional data, lists of data, and functional operations of the protocol. Examples of each message type are given in both encoded and unencoded form, and timeline diagrams are used to clarify the operation of the protocol over the course of a user session.

In chapter 6, we describe a formal verification of the protocol that was performed using a proof assistant called ProVerif. This allowed for the description of SessionArmor in an algebraic model of computation known as process calculus. By compiling the concurrent processes of the protocol into Boolean expressions, ProVerif was able to use established satisfiability techniques to make claims about the security properties of the protocol. Two properties were considered, the secrecy of the HMAC key in the presence of an adversary, and one-to-one correspondence of client requests and server responses, the second

of which is equivalent to strong authentication. Both of these properties were proven by the ProVerif model.

In chapter 7, a reference implementation of SessionArmor is presented, with documented source code. The server implementation is a web application middleware, implementing the entire SessionArmor specification with additional developer-oriented features such as descriptive exceptions for all error conditions, full configuration options, and isolated debug logging. The client implementation is a Google Chrome extension with zero-configuration needed from the user. Both implementations were performance-tested for application requests and session creation time, and both performed well within the specified goals. With analysis of the histograms presented in this chapter it can be seen that most of client and server processing time for session creation happens in less than 370μs and most of client and server processing time for application requests happens in less than 1.7ms. Finally, in chapter 8 we include some concluding remarks.

1.2 Cookie-backed Sessions

As a stateless protocol, HTTP treats each request and response as an independent transaction. Thus, there is a fundamental challenge in associating some specific client, and some permissions, with each request and response. One solution to stateless request authentication is for the client to include some data with each request that the server may be able to use to identify and associate that request with an active application session. The most common method of implementing this technique is to use a *bearer token* embedded in an HTTP Cookie, which is sent with each request. This is depicted in fig. 1.1.

Bearer tokens have been given that name because they "bear the burden" of authenticating *all* requests for the entire duration of the session. Critical security issues associated with this practice are the vulnerabilities that facilitate session hijacking; the ability for an

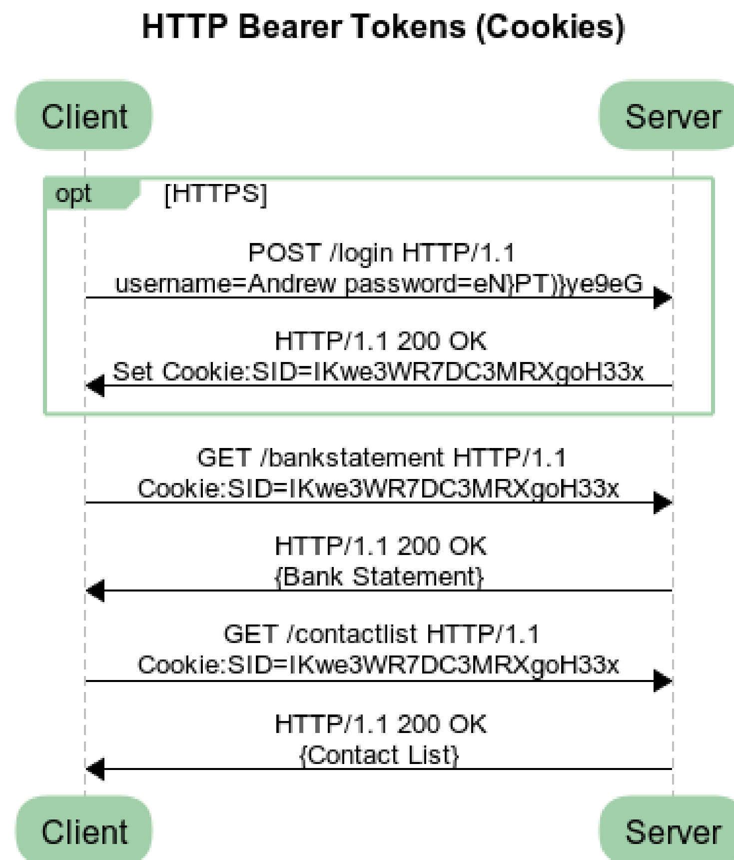

Figure 1.1: Bearer Tokens in the form of HTTP Cookies. The current practice for authentication of web requests

attacker to perform unlimited actions and/or repeated actions on a user's behalf if they can obtain this token.

1.3 Motivation – The Session Hijacking Attack

Session hijacking is a widespread vulnerability that fundamentally undermines the security of web applications. We will show that this is the case in section section 2.1 using measurements from SessionJack. With a variety of attack vectors, it allows a malicious party to gain control of an authenticated session, make requests on behalf of the authenticated party, and perform privileged actions, often without limitation in both scope and

time. The authenticated session may be that of an individual user, or an automated connection used for industrial, government, or other purposes.

This token is almost always stored as an HTTP Cookie, and the data it contains is called the Session ID. This token is pre-authenticated by user credentials and is used by the application to authenticate all requests until it expires. The period of time between token creation and expiration is the "session". The weakness of this approach is that this token is *static* for the lifetime of the session, which is often days or even months, and that it is used to authenticate all requests regardless of the resource requested by the client. Session hijacking is performed by exfiltrating a bearer token from the victim's browser, user agent, or communication medium with the web application. If exposed by one of the following vulnerabilities, it is always in a form that will be accepted as the session identifier by the server.

The following exfiltration vectors are common:

1.3.1 Session Sidejacking

The attacker uses packet sniffing, offline traffic decryption, or obtains requests logs (such as those stored behind an HTTPS gateway) in order to acquire a victim's authenticated Session token. As long as the bearer token is valid, the attacker can use it to make unlimited privileged requests. With man-in-the-middle (MITM) capabilities, the attacker can use the bearer token to modify requests in real-time and make unrestricted additional requests. Techniques for obtaining MITM include: local packet sniffing via open Wi-Fi, decrypted Wi-Fi, ARP poisoning of switched networks, and source routing at the WAN level. Offline traffic decryption is even possible on the latest WPA2-secured WiFi networks if the attacker observers derivation of a Group Temporal Key (GTK), and has knowledge of the Pre-Shared Key (PSK). [41] [31]

One mitigation for Session Sidejacking is to use HTTPS (TLS) for every client request.

If used for all resources during a session, the secure tunnel of TLS cannot be breached even when packet-sniffing or offline link-layer traffic decryption is used, and thus a session token cannot be recovered by an attacker. However, it is interesting to note that Sidejacking is *not* always prevented by the use of TLS. If the session cookie does not have the secure flag set, all that is needed is for the attacker to coerce a victim to visit any page under their control, for example, the landing page for a cafe WiFi access point, or the contents of an HTML email. It is trivial, e.g. for an attacker to coerce sensitive cookies to be sent over the network by causing the user to request an `<img src=>` for the target domain using the insecure scheme for an HTTP connection. This will cause the victim's browser to make an insecure request destined for the target domain, including the desired session cookie: a protocol-level downgrade attack. The recently introduced HTTP Strict Transport Security (HSTS) standard [26], is a mitigation to this attack. However, it is not widely deployed, as we will show using the results from Session Jack.

1.3.2 Cross-site scripting (XSS)

If an attacker can execute JavaScript code in the context of the target web application, and the HTTPOnly flag has not been set on the Session Cookie, then the Session ID can be stolen easily by making an cross-domain request containing the (programmatically accessible) Session ID to a server under the attacker's control. Cross-Site Scripting (XSS) vulnerabilities of this type are quite common [46], and are often introduced when user-generated content is not properly escaped as text. For example, an improperly sanitized comment on a blog post might contain JavaScript, executed unwittingly by any user who views the comment.

If the malicious payload size is not large enough to allow for an `XMLHttpRequest` or `fetch` request, an image node can be added to the DOM, with a `src` attribute crafted to the contain the Session ID, inciting the browser to make a `GET` request with this target payload

to the attacker's domain. Most browsers will aggressively prefetch even those image nodes which will not be visible to the user. Conveniently, the response from the attacker's server need not be valid image data as the request alone satisfies the desired outcome of divulging the session token. Also, image requests are not subject to the same origin policy, which makes then even easier to implement than the traditional asynchronous request, which requires the attacker to configure a CORS policy for their domain.

1.3.3 Session Fixation

With Session Fixation the attacker chooses a Session ID for the victim, which the victim subsequently authenticates while under attacker control.

The Session ID can be set, for example, using a modified HTTP response from the server if a MITM is in effect, or in an email message containing a Session ID in the URL. The latter requires that the web application implements a (hopefully legacy) mode of operation in which the Session ID can be initialized with a GET parameter. The former can be achieved with a Set-Cookie header or with injection of the lesser used `<META http-equiv>` tag, which allows for arbitrary headers to be set. The META tag allows for the attack to be performed even by a proxy that can only modify the HTTP document body. [30] Session fixation can be universally prevented by issuing a new Session token on the server-side each time there is a new authentication event on the part of the client, for example, the time at which a username and password is provided.

1.3.4 Physical Access

If the attacker has physical access to a machine, it is trivial to obtain session credentials from files stored by the victim's user agent. Once an attacker has obtained the bearer token, he or she can perform unlimited actions on behalf of the user for the duration of session, which can often be extended by the attacker indefinitely.

1.3.5 Cross-Site Request Forgery (CSRF)

Although not a direct Session Hijacking vulnerability, Cross-Site Request Forgery must be mentioned here, as it is one of the more severe web vulnerabilities that allow actions to be taken on a user's behalf. For the most part, unwanted cross-domain requests are prevented by the browser's Same Origin Policy (SOP). This is the policy that prevents one website from making a JavaScript-based request for data that resides on a different domain. However, there is a case where the SOP does not aid in protecting from unwanted external requests: `GET` requests that mutate state on the server-side. A sign of poor understanding of HTTP, having such `GET` endpoints opens an application to denial-of-service and allows actions to be performed on behalf of users without their knowledge.

In the ideal attack scenario for CSRF, there is a `GET` endpoint served by the target application that performs some desirable action, e.g. creating a financial transaction (which should be done with a `POST`). The attacker places a reference to this URI in a `<script>`, `<img>` or `<link>` tag on a page that they control. They then coerce the victim to visit the page (or perhaps view their payload on an advertisement on a popular website). Upon attempting to load the specially crafted resource URI, the victim's browser will make the sensitive request to the target domain, and make a request to perform the privileged action, *including the victim's session token*. This means CSRF is an invisible attack for the victim that can be performed off-domain and leverage an active session.

There are two primary mitigations for CSRF. One is to not mutate any state behind `GET` endpoints, keeping those endpoints read-only and cacheable. However, it should be noted that HTTP POST endpoints that expect `multipart/form-data` can receive cross-site requests from a hidden form placed on a malicious domain and submitted via JavaScript. Therefore, the only true mitigation is to place a pseudorandom value on each page of the origin domain that must be included with any subsequent HTTP request. This is called an

anti-CSRF token.

1.4 The Session Extension Attack

In the abstract, the purpose of issuing an expiry time for an authenticated session is to enforce a known maximum elapsed time between acquisitions of credentials from a user. This is a contract that the user should expect from a secure web application; if the user provides their secrets to begin a session, the session should only be valid for a specific amount of time defined by the moment at which those secrets were divulged. Therefore, it would be counterproductive to upholding this contract if any user request was able to extend the length of a session without requiring re-authentication. However, in some web application frameworks, the session expiry time is extended every time that session data is modified. [20]

An attacker could take advantage of this vulnerability by periodically sending requests to the server known to modify session data in some innocuous way, for example toggling a flag used to show or hide help information on the page. In this way the attacker could keep a stolen session "alive" for much longer than the intended expiry time, even indefinitely. For those frameworks which are vulnerable, this nullifies any potential benefit of regular session tokens having a specified expiry time. Of course, a pre-requisite for this exploit is the acquisition of an active session token, which Session Armor prevents entirely.

In some cases, the expiration time of a session cannot be verified by the server, so any mitigation would seem fruitless. An example is a session ID implemented with a regular cookie that does not embed a cryptographically signed session expiration time.

In all cases, it is important to protect against session extension, because it limits the duration that the client's user agent will keep the bearer token in memory or on disk, making it available to an attacker. Most importantly, it limits the time period for which a given

bearer token is valid. Prevention of this exploit is fairly simple, the expiration time of a session token must never be modified without re-authentication from the user.

1.5 Rogue Browser Extensions

Another method for cookie exfiltration is the use of a malicious browser extension. These add-ons, which add functionality to browsers such as screenshot-taking, bookmark-management, and custom styling, can surprisingly be given permission to access all cookies stored by the browser without limitation.

The permissive environment available to browser extensions periodically receives critical attention [25], but little has been done to address the issue. Major browsers require extensions to request a specific permission for unfettered cookie access, and have an automated approval process that scans extensions for malicious behavior. Despite these efforts, the dialog presented to the user indicating the extent of the add-on's capabilities is much too tame. Installing one of these add-ons for any length of time gives it permission to send authenticated bearer tokens for all logged-in web applications to a remote party. Although browser vendors may claim that they check for this activity, any function which operates on cookies and is loaded via a third-party script can be modified at runtime to exhibit this behavior. The loading of third-party code into browser extensions is not only possible but encouraged by the developer documentation [19]. Compare the warning for this capability, 1.2, to the one presented for a potential compromise of an encrypted tunnel to a *single* domain, fig. 1.3.

Not only is the extension warning less imposing, but there is also much less user friction in allowing it. The TLS warning requires the user to first click "Advanced" and then click a small grey link if they wish to ignore the vulnerability. Ideally, the browser would present the extension warning with the same level of severity. This is a classic trade-off between

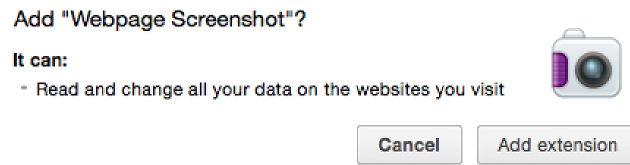

Figure 1.2: Warning dialog for installing a browser extension with unlimited cookie permissions

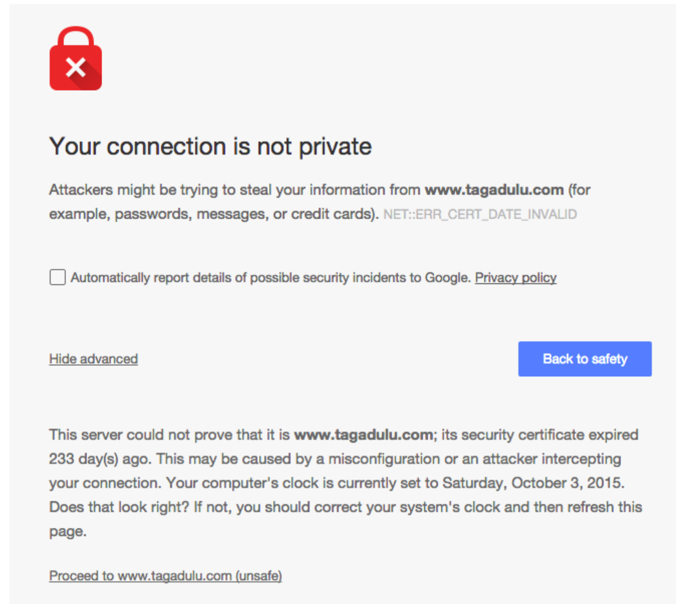

Figure 1.3: Warning dialog for visiting a domain with an expired TLS certificate

convenience and security. This research used a browser extension to analyze cookie usage across the web, and it would not have been possible without this capability.

2. Observing Session Hijacking of Existing Web Applications

2.1 SessionJack

A tool called `sessionjack` was written to gather data on cookie protection, and support the claims of: the widespread use of bearer tokens, common vulnerabilities, and excessive session lifetimes. This tool can be used for analysis of Session Hijacking vulnerabilities in any web application. SessionJack was used to analyze the authentication mechanism of 108 web applications, spanning a range of popularity from rank 1 to rank 3,489,038 as reported by the traffic analytics company Alexa Internet, Inc. The following data was recorded:

- Whether or not the site was vulnerable to Bearer Token Exfiltration
- Whether or not the site was vulnerable to Cross-Site Scripting (XSS)
- Whether or not the site was vulnerable to Packet Sniffing (Sidejacking)
- Average of all token lifetimes if vulnerable to both XSS and Sidejacking
- Average of protected token lifetimes if protected from XSS
- Average of protected token lifetimes if protected from Sidejacking

To verify that these websites were indeed vulnerable if SessionJack marked them as such, a web browser add-on was used to load the vulnerable cookies and observe that the session was still active. The most noteworthy finding from this data was that all 108 sites were vulnerable to bearer token extraction. The meaning of this in practical terms is that nearly all web applications today, from the highest to lowest profile, are storing a persistent token in a cookie or GET parameter, which can be re-used to make authenticated requests

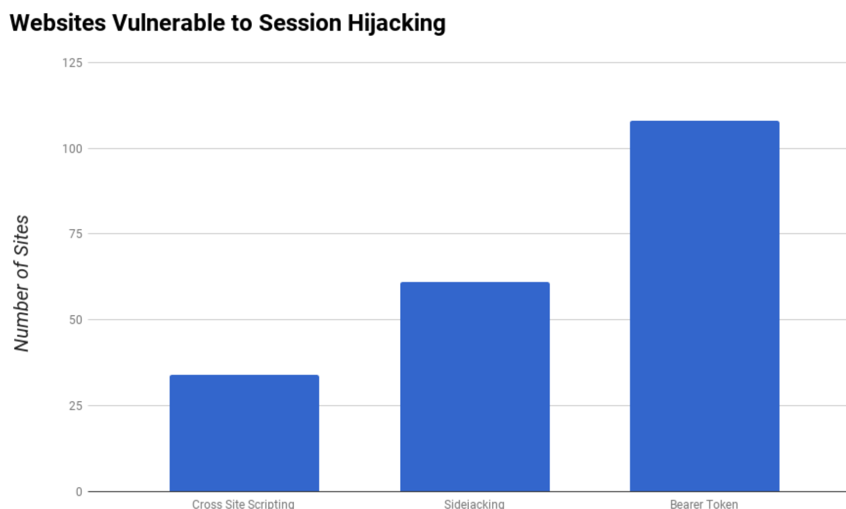

Figure 2.1: Number of sites that are vulnerable to a Session Hijacking attack via Cross-Site Scripting, Sidejacking, and Bearer Token Extraction

of the application. Cookies are protected by the browser via the same-origin policy, which enforces that script running on one domain cannot read cookies stored for another domain. However, there are multiple ways that cookies can be exfiltrated silently by a remote adversary as discussed in the prior section.

In fig. 2.1 the number of sites potentially vulnerable to XSS, Sidejacking and Bearer Token Extraction are plotted. The ratios are 31.48%, 56.48%, and 100% respectively. Table table 2.1 includes a selection of the vulnerability data. A complete listing can be found in Appendix B.

2.1.1 Analysis of Bearer Token Vulnerabilities

In fig. 2.2 the popularity rank is plotted for sites that are vulnerable and sites are not vulnerable to Session Sidejacking. More popular sites have a lower rank, for example, google.com has rank #1. A summary is provided in table 2.2. We can say that, on average, websites that protect against Session Sidejacking are higher-profile, more popular

Table 2.1: Domain Susceptibility to Session Hijacking

Domain	Bearer Tokens	XSS	Packet Sniffing
mail.google.com	Vulnerable	Protected	Protected
amazon.com	Vulnerable	Vulnerable	Vulnerable
instagram.com	Vulnerable	Vulnerable	Protected
alibaba.com	Vulnerable	Vulnerable	Vulnerable
venmo.com	Vulnerable	Protected	Protected
tdbank.com	Vulnerable	Vulnerable	Vulnerable
wordpress.com	Vulnerable	Vulnerable	Protected
paypal.com	Vulnerable	Protected	Protected

Table 2.2: Mitigation of Session Sidejacking by Alexa Rank

	Count	Mean Alexa Rank	Median Alexa Rank
Total	108	63109	4296
Does Mitigate	47	6040	1198
Does Not Mitigate	61	103248	11012

sites. However, the difference in the mean between those that are protected and those that are vulnerable is less than one standard deviation of the distribution of those sites tested. We can therefore draw the conclusion that a significant number of high-profile websites are vulnerable to Session Sidejacking. These sites span a variety of industries and classes of risk for the user. They include: (entertainment) twitch.tv, steampowered.com, netflix.com, (shopping) alibaba.com, amazon.com, ebay.com, and (finance) tdbank.com, chase.com.

The relationship between popularity rank and vulnerability to Cross-Site Scripting is plotted in a similar fashion in fig. 2.3 and summarized in table 2.3. Comparing the tabular data for both, we can see that a larger proportion of sites use the `HTTPOnly` flag than those that use the `Secure` flag. This makes sense from a project management standpoint. In the development of most web applications, it is easier to enforce that JavaScript is *not* used to access the session token, than it is to deploy always-on HTTPS. In the same manner as for Session Sidejacking, we can draw two major conclusions: on average, higher profile sites protect against Cross-Site Scripting, and yet a significant number of high-profile

Figure 2.2: Alexa rank for websites which mitigate Session Sidejacking and those that don't

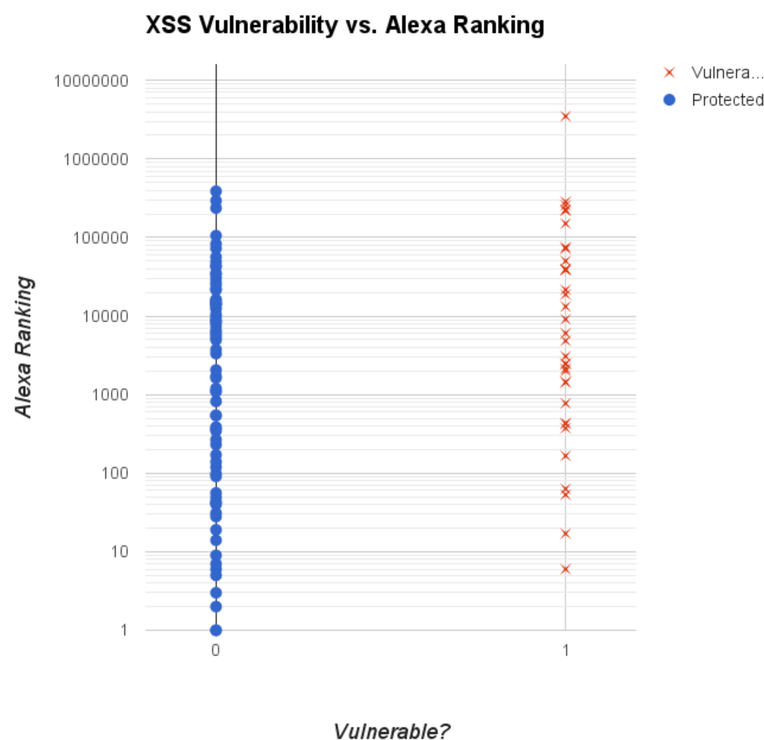

Figure 2.3: Alexa rank for websites which mitigate Cross-Site Scripting and those that don't

websites are vulnerable. These vulnerable sites span a variety of industries and classes of risk. They include: (business services) ibm.com, oracle.com, myminifactory.com, (entertainment) hitbox.tv, kongregate.com, soundcloud, and (finance) tdbank.com.

The vulnerabilities observed for major American banks and top-100 websites is disheartening, when enabling these slight protections represents little development cost. Some of the vulnerabilities are particularly noteworthy. Airdroid.com is a service which allows unrestricted access to an Android phone via a web application. Airdroid has made the following claim, "Your data security is among the top priorities of AirDroid and industry standards are strictly implemented." However, it is vulnerable to both Session Sidejacking

Table 2.3: Mitigation of Cross-Site Scripting by Alexa Rank

	Count	Mean Alexa Rank	Median Alexa Rank
Total	108	63109	4296
Does Mitigate	74	24102	3534
Does Not Mitigate	34	148004	5445

and Cross-Site Scripting. Hitbox.tv, a popular streaming video service, persists sessions in a way distinct from all of the other 107 sites analyzed. It stores the session bearer token using the `localStorage` API, and restores a session cookie using this information even if cookies are cleared. This means that even if the session cookie were protected with `httpOnly`, any JavaScript, domain-wide, can access the token via the `localStorage` API, which has no protection from access. This is an interesting way to "resist" cookie-clearing behavior of users, but makes the site more vulnerable to attack. Regardless, hitbox.tv does not protect its session cookies from either of the two attacks under consideration.

The developer access website of Oculus.com also has an uncommon way of obtaining its bearer token. It uses JSONP to make a cross-domain request to graph.oculus.com to retrieve an authenticated OAuth2 session token. This token is protected from both attacks, but it reveals a weakness of OAuth: although the authentication protocol is slightly more complicated than the usual POST request and `Set-Cookie:` response, when implemented in a web browser, it results in the same weakness of the traditional scheme: a static bearer token is persisted with the client for a possibly indefinite period of time.

One potential argument against enabling Sidejacking protection, is that it also requires that all requests are transmitted via HTTPS, which can be expensive to deploy. However, some of the sites which are vulnerable are clearly making an attempt to be "HTTPS only", but have not fully protected their users. These sites redirect any insecure request to an HTTPS URL using by using responses HTTP 302, "Found", or better HTTP 301, "Moved Permanently", which instructs the browser to cache the secure origin indefinitely. There

is an ever better protection defined in RFC 6797 [26], "HTTP Strict Transport Security" (HSTS), which instructs the browser to enforce that all future requests to this domain use HTTPS.

Sites that implement this redirection technique should be using HSTS, but many are not. This results in the exposure of an HTTP downgrade attack. This is a simple attack to carry out on an unprotected network, or when DNS poisoning is possible. The attacker entices the victim to make a non-HTTPS request to the target domain via any number of means. Most simply, the attacker can present a link to the domain in an email or other communication medium. Another options is to replace an image `src` attribute on an *unrelated* domain with an insecure location on the target domain. [44] In this way, the victim will not recall interacting with the target domain at the time of the attack. In all cases, as soon as the victim's browser visits an HTTP location for the domain, their session cookie will be sent in the clear on the network. Domains that attempt to mitigate but are vulnerable to this downgrade include: ibm.com, tdbank.com, bitcoin.cz, chase.com, and codechef.com.

Two out of the 108 sites analyzed were using a particularly insecure technique for session persistence which was common before cookies were introduced to the web by Montulli in 1996 [35]. This being the inclusion of a session token as a GET parameter on every URL rendered on the page. Doing this has an important security implication. The path of a GET request is much more likely to be logged, cached, or stored in other ways than the body of a POST request. For example, it will be saved in a bookmark, pasted into an email, logged by a proxy, and appear on a network monitoring tool. The two websites that do this are usaco.org and fourmilab.ch. This is somewhat understandable, as both of these websites have been online since before 1998.

There are also some impressive standouts in terms of Session Hijacking mitigation. Intuit.com, the tax preparation service, protects against both sidejacking and XSS using the standard flags. In addition, it uses a heuristic to determine if the application is being

accessed from an unknown device. If it is, a one-time code is emailed to the user before login can proceed. This is close to implementing second factor of authentication, "something you have", however it is really an additional "something you know" factor. So, this could be considered a method for implementing 1.5 factor authentication. Venmo.com, the person-to-person payments system, mitigates against both common vulnerabilities, and also the HTTP downgrade attack by using HSTS with an expiry of 1 year, meaning that they do not expect to serve an insecure HTTP request for a period of one year following any recent request. Intuit one-ups this by using a max-age parameter of 0. They expect to never serve an insecure HTTP request again. The combination of these techniques is an excellent example of current best practices.

2.1.2 Analysis of Bearer Token Lifetime

There are few parameters of web session security currently under the control of the application developer, one of them is session lifetime. This is the duration for which a generated bearer token is valid. The longer this period of time, the more opportunity there is for an attacker to obtain and use the token without the victim's knowledge. For high risk applications, such as government, finance, or health, a value of 30 minutes would be reasonable. For low-risk applications, such as entertainment or an enthusiast community, 30 days is the recommended upper limit.

The plot in fig. 2.4 compares cookie lifetime in days against the popularity of websites as measured by their Alexa rank. Points in blue are all cookies from web applications that are vulnerable to both Sidejacking and XSS. Points in red are the all `Secure` cookies from those sites which mitigate Sidejacking. Points in yellow are all `httpOnly` cookies for those sites which mitigate XSS.

There are no apparent trends in this data. Even some of the most popular sites have extremely long-lived tokens, and some of least popular sites have short-lived ones. Only

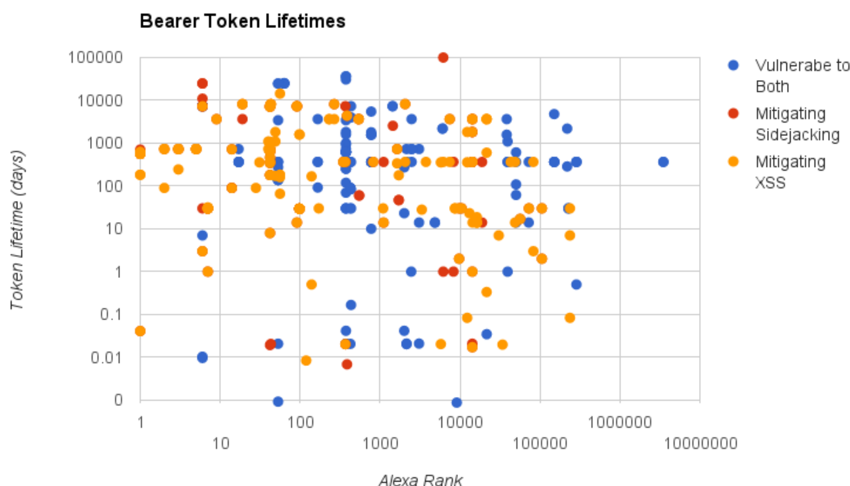

Figure 2.4: Token lifetimes vs site popularity for each type of hijacking protection

a few sites have cookies which are limited to 30 days of validity, and even less limit this to less than 1 day. There is no clustering among cookies that mitigate a certain vulnerability. Therefore, the primary conclusion that we can draw is that there is no relationship in practice between the choice of session lifetime and the use of these protections. This informed the enforcement of a maximum session lifetime of 30 days for the SessionArmor protocol.

2.2 JackHammer

Another tool was written called `JackHammer` which demonstrates Session Hijacking vulnerabilities live, using two web browsers to observe the attack. `JackHammer` includes a Google Chrome extension, a Mozilla Firefox extension, and a websocket server written in Python. Websocket is a JavaScript API that allows on-page web scripts to maintain persistent connections with a server.

`JackHammer` operates as follows. When each browser is initialized, its respective extension connects to the local websocket server if it is available. Then, the user visits a webpage on Chrome for which they would like to simulate a hijacking attack. They log in

to the application in order to obtain an active session token. They can then right click on the page and choose "JackHammer" to open a new tab with the user interface. A list of peer Firefox browsers which have connected to the local websocket server is displayed. A button can be clicked which clears all cookies for the selected domain in the Firefox browser. At this point the user should observe that they are logged in to the application on Chrome but not on Firefox, this can be seen in fig. 2.5. The Chrome browser is on the left, which simulates the victim. The Firefox browser is on right, which simulates the attacker. The user can then click one of three buttons to simulate a Session Hijacking attack.

- **Bearer Tokens** tests if the site's session is vulnerable if all of its cookies can be exfiltrated
- **Cross-Site Scripting** tests if the site's session is vulnerable if its cookies without the `HTTPOnly` flag can be exfiltrated
- **Packet Sniffing** tests if the site's session is vulnerable if its cookies without the `Secure` flag can be exfiltrated.

Upon pressing one of these buttons, the websocket server is used to transfer the vulnerable cookies from Chrome to Firefox. A Firefox notification appears indicating how many cookies were transferred. This can be seen in fig. 2.6, in which the user has chosen to test for Cross-Site Scripting vulnerability. The user can then refresh the page in the Firefox browser and note if they remain logged in to the application. This can be seen in fig. 2.7. Note that this website is potentially vulnerable to an XSS attack, as personal information is immediately visible upon refreshing the browser on the right.

Figure 2.5: Chrome (left) with logged-in session. Firefox (right), with no logged-in session

Figure 2.6: The JackHammer user interface, a test for Cross-Site Scripting vulnerabilities has been initiated

Figure 2.7: The JackHammer test yields a positive result, Firefox now has a logged-in session using an exfiltrated token

3. Existing Session Hijacking Mitigation Techniques

3.1 Standards for Session Token Protection

Session tokens are most commonly cookies, stored in the browser and automatically included in each request from the client. The cookie may be set client-side, via JavaScript, or server-side, via any HTTP response from the web application domain. At the time that the cookie is set, there are options that can be used to enhance security. Sometimes these options are misunderstood or ignored by application developers.

3.1.1 HTTPOnly

`HTTPOnly` is a flag that can be set when storing a Cookie. This flag indicates to the browser that the Cookie should not be readable by JavaScript, and thus mitigates the effects of a Cross-Site Scripting attack. However, there have been browser bugs in the past that allowed JavaScript to read HTTPOnly cookies. [23] Session IDs are a highly valuable target for a malicious actor because they allow unlimited forged requests. Clearly, it would be beneficial if there were a mechanism for secure storage of session information that was inaccessible to any client-side scripts, including browser extensions. The Session Armor protocol proposes an encrypted storage location in browsers, inaccessible from all types of client-side scripts, to be used specifically for the storage of session secrets.

3.1.2 Secure

`Secure` is a similar flag for HTTP cookies. It indicates that a cookie should never be sent over a non-SSL or non-TLS secured connection. This does effectively mitigate most network-based attack vectors for Session Hijacking. However, if this flag is set, and

`HTTPOnly` is not, the cookie remains vulnerable to XSS. Also the cookie may be logged unencrypted or exfiltrated on either side of the the TLS termination (for example by a reverse proxy or rouge browser extension). SessionArmor enables authentication for which the data being transmitted over the network is not a valuable payload by using a mechanism for absolute replay prevention. This obviates the need for a secure tunnel in maintaining the integrity of user authentication.

3.1.3 Expiration Time

When a cookie is set, it is possible to specify an expiration time. If no expiration time is set, then the cookie is deleted when the user's browser is closed. If session tokens are used, it is advisable to set the expiry time to as early as is reasonable given the sensitivity of the application. However, in practice, web applications often set session tokens that last for months or even years as discussed in section 2.1.2. This gives an attacker ample time to exploit a session hijack without being noticed.

3.1.4 HSTS

Of course, integrity, secrecy, and authentication are accomplished by "always-on" TLS. However, as noted earlier, TLS can be expensive to deploy for multiple subdomains and when using geographically distributed reverse-proxies. HTTP Strict Transport Security is a an IETF standards track specification now in use by some websites. It specifies an HTTP header that instructs the browser to never request a resource from a particular domain using an insecure connection. This a means for Sidejacking vectors of cookie leakage to "fail closed", a desirable security property in many situations.

3.1.5 HTTP Digest Authentication

Interestingly, HTTP has had since 1997 [24] a specification for request authentication that does not use cookies at all. It defines a session as the period of time during which the solution to a cryptographic challenge is valid for a given user, and does not store a session secret in a JavaScript accessible location. This is HTTP Digest Authentication, which operates in the following way.

When a user makes a request for a protected resource, the server responds with status code 401, Unauthorized, and also includes a `WWW-Authenticate:` header field. This header field includes data that is used by the browser to instruct users that they must provide a specific set of authentication credentials in order to continue, and also informs the browser of parameters that the authentication mechanism will use. These parameters include a nonce from the server, an opaque token, and an indication if request body integrity should be computed for each request.

The server then only responds with data to subsequent requests if the following conditions are met. The browser must respond with a "digest", which is a hash-based message authentication code (HMAC). **HMAC is mechanism by which an entity can prove knowledge of a secret without revealing the secret itself.** Also, in the process, **HMAC authenticates the origin of some data as being an entity with knowledge of the secret**. The general construction of an HMAC is a function with two parameters: the key, and the data to be authenticated.

$$\text{HMAC}(\text{key}, \text{data}) \coloneqq H(\text{key} \oplus \text{ipad}, H(\text{key} \oplus \text{opad}, \text{data})) \tag{3.1}$$

With HMAC, H is any cryptographically-strong hash function, i.e. a lossy compression function that is collision resistant. The value of an HMAC result, the "MAC", is something which can be recomputed by an entity which challenges the authenticity of the data, and

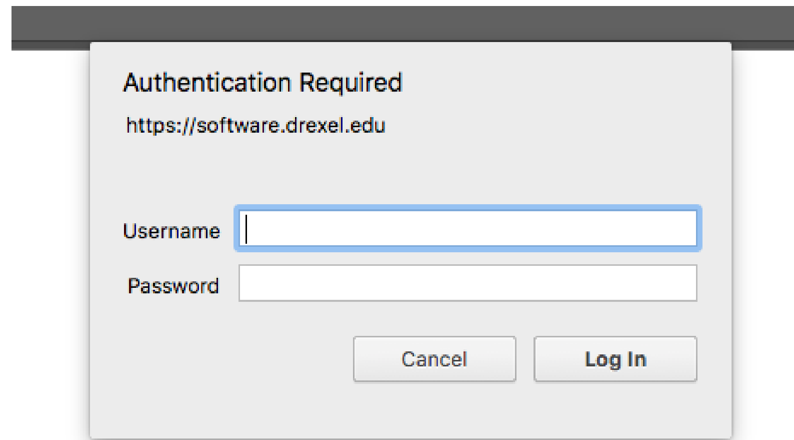

Figure 3.1: Username and password dialog box for HTTP Digest Authentication

must have access to the key. The *ipad* and *opad* are used to prevent length-extension attacks on the data. [7] Digest authentication actually uses only a single iteration of the hash-function and so it is potentially susceptible to these attacks.

In the case of Digest Authentication, the key is itself is a hash of the user's username and password. The authenticated data is a nonce from the server, a nonce from the client, a counter from the client, the requested path, and optionally the requested body data. The first two allow the client and server to mutually authenticate, although this is optional. The counter allows the server to provide absolute replay prevention, although this is optional. These optional components weaken its security properties. There is also no mechanism for the client to authenticate header data, or enforce that time-based replay prevention be used by the server.

Despite these shortcomings digest authentication is much more secure than cookies because each request is authenticated individually. Unfortunately, one of the primary reasons that it is not used by most web applications is that there is no control over the appearance of the username and password dialog, which looks like the one seen in fig. 3.1.

3.2 Proposed Methods for Session Protection

3.2.1 Secure Cookie Protocol

In 2005, Liu et al. [32] proposed a cookie protocol with the goals of authentication, confidentiality, integrity, and anti-replay. They first present a related cookie protocol from Fu et al. [21] that relied on the following two fields to be to be set in a cookie by the server:

- Username | Expiration Time | Data
- HMAC(sk, Username | Expiration Time | Data)

Where sk is a global server secret, and Data is any data that the server wishes to remember about the client. They note that this protocol has three weaknesses: lack of confidentiality, no resistance to replay, and lack of resistance to volume attacks. An alternative protocol is then presented that aims to correct these flaws. It has the server set a cookie with the following two fields.

- Username | Expiration Time | E(k, Data)
- HMAC(k, Username | Expiration Time | Data | Session Key)
  - Where k = HMAC(sk, Username | Expiration Time)

The server key, sk, is known only by the server and used for all sessions. The derived key, k, can be computed only by the server. The claim of confidentiality comes from the fact that only the server can decrypt "Data", using a key that is derived per-session. This claim appears to be sound, although web applications often retrieve session data from some persistent medium once they can validate an authenticated request, obviating the need for this token.

The proposed method for anti-replay comes in the form of *Session Key*, which is presented as the SSL session key for the connection between the server and client. The concept being that even if recorded SSL traffic is cracked via some means, the attacker would not know the SSL session key. There are two flaws with this scheme. First, the SSL session key in all hardened implementations is not accessible via any application programming interface, and so this HTTP session mechanism would have to be deployed in conjunction with a modified SSL stack. Also, by definition, if SSL traffic has been decrypted to the point that an entire cookie is recovered then the attacker is likely in possession of the session key.

They present "volume attacks" in the first protocol as being attacks on the underlying hash algorithm if the same key is used for all cookies. This is a legitimate concern and their solution is reasonable because the derived key includes a time component. SessionArmor also does not re-use HMAC keys. A major drawback of both of there protocols is the lack of per-request authentication, they do not authenticate *user intent*.

3.2.2 SessionLock

In 2008 Aida et al. conceived of SessionLock. [1] The primary concept is that it revives the notion of a session key stored within each URI on the client webpage. However, rather than use GET parameters, which are preceded by a "?" in the path, it uses the fragment identifier, which is preceded by a "#". The difference between a fragment identifier and a GET parameter from a security standpoint is that a fragment identifier never leaves the browser when requests are made. Thus, their protocol is not subject to eavesdropping. The session key is transferred to the client via a `Secure` cookie, at which point the session may be downgraded to standard HTTP.

Message integrity is provided by an HMAC with each request computed using a JavaScript implementation of SHA-1. JavaScript automatically adds event handlers for all link-clicks, form-submits, and XHR requests, to intercept and add the HMAC. The session key is

“transferred” from page to page by executing a JavaScript function which appends the key as a fragment to every URI on the page upon page load. Essentially, the fragment identifier is used as a “safer” storage location for the HMAC key than a cookie, because it is never sent over the network.

One nice feature is that a mechanism is provided to restore the session key if a user clicks a link “too quickly” for it to have been placed in the fragment by JavaScript. This same mechanism highlights that SessionLock is clearly vulnerable to an XSS attack. If malicious JavaScript were to execute in the context of the application, the HMAC secret would be strewn throughout the page and easily accessible. Interestingly enough, it is highly resistant to CSRF, because the HMAC key can *only* be obtained by JavaScript running on the first-party domain. Session Lock does not specify the exact fields which should be included in the HMAC.

3.2.3 Web Key

In 2008 Close conceived of Web-Key [13], which is similar to SessionLock in that it uses URL fragments sent by the server for authentication, which are assumed to be sent over TLS. Rather than reject the notion that “fragment-added” URLs might be bookmarked or inadvertently shared by the user as in [2], Web-Key embraces this, each URL fragment is unique and represents a different “capability based” permission. For example, a user might share a URL that gives a friend permanent access to view a file but does not allow access to any other resource. Close rejects the notion of “ambient credentials”, i.e. cookies, and goes so far as to propose that if Web-Key was in place then the same origin policy would not be necessary.

Web-Key at first appears to be a solid concept, but there is a strange omission in the presentation: an explanation of how the user obtains the URI-based credentials in the first place. Each resource is supposedly protected by a unique URL fragment key. Web-Key

presents the scenario in which the link for a sensitive resource might be indexed by a search engine, meaning that web-key protected URIs are sent as part of the body of HTTP responses. If this is the case, then providing access to a single *root type* resource via Web-Key is equivalent to providing access to all resources reachable from it in the web graph. The authors write, "A web application using web-keys in effect generates passwords on behalf of the user." This might seem like a welcome paradigm shift, but in practice, as soon as a user mistakenly links to a URI with a sensitive Web-Key, all of that user's private information would be available for public access.

3.2.4 HTTPI

HTTPI is a transport protocol proposed by Choi and Gouda in 2011 [12]. Its primary goal is to provide a middle of the road option between HTTP and HTTPS. The authors note that HTTP is inexpensive to use and is easily accommodated by middle boxes on the internet, such as caching proxies and distributed content delivery networks, whereas HTTPS is the opposite in both respects. HTTPI promises to create message integrity with limited cryptographic overhead but does not seek to provide secrecy. The protocol operates as follows:

1. The web client and server negotiate a session key using a "TLS-like" protocol.
2. The server assigns the web client a Session ID, and allocates resources for the session
3. Each subsequent message contains the session id and an HMAC keyed using the session key. This uses SHA-1 for all immutable fields in the header, including cookies
4. The Content-MD5 header from HTTP is required rather than optional. They note that SHA-1 could be used here, but feel that MD5 is sufficient and is already implemented on most middle box software.

Interestingly, to avoid the attack that will be demonstrated in section 3.2.6, the entire certificate signing infrastructure of TLS would be necessary for the "TLS-like" protocol to be secure. This is an unrealistic burden that is largely ignored in the presentation. They propose that the HMAC header fields be treated as end-to-end immutable headers as some have been specified. This ensures that the server can verify the integrity of each request by performing a hash using the predetermined session key. Also, because the Content-MD5 header is not a MAC and is separate from the SHA-1, it can be used to cache the same content for multiple clients, and also avoid caching of duplicate content. Another positive aspect of HTTPI is its performance, which is within 2% of HTTP.

HTTPI has some weaknesses when it comes to session protection. An attacker can eavesdrop on the connection and use the plaintext Session ID to replay requests. There is neither time-based nor nonce-based replay prevention. However, an attacker cannot request additional resources or inject new GET or POST parameters due to the HMAC. There is some protection against XSS because the session key is not accessible via JavaScript. However, there is no protection against CSRF, because the browser will perform the HMAC on the user's behalf. Another insecure aspect of HTTPI is that it promotes long-lived sessions as a feature, which increases the chance of replay attacks. There are known collision attacks against MD5, so using it for request body integrity is not secure. Most notably, they demonstrate that a web-cache would need access to a session's HMAC key in order to validate a request before sending a response. This undermines their claim of cacheability as a primary contribution of the protocol.

3.2.5 One Time Cookies

In 2012, Dacosta et al. [14] created One Time Cookies, an unencrypted session protocol with the goal of message integrity and the prevention of replay attacks. A second goal was limiting the state required by the application, to ease deployment in a highly distributed

environment. The protocol operates as follows.

1. The user makes a request via TLS to the server including an HTTP header, `X-OTC:`, username, and password

2. The server responds via TLS with a *non-cookie* HTTP Header that includes in plaintext a nonce, expiration time for the session, session key, and domain/path for which the session key is valid. It also includes an opaque value computed via symmetric encryption for which the key is an XOR of a server-only key and the nonce. The values encrypted are the user id, session key, and expiration time for the session.

3. When a user makes any request, in the clear they include: the nonce, a single-request expiration time, and an HMAC. The data authenticated is the url, the single-request expiration time, and the data of the request, keyed by the session key. They also include the opaque value sent by the server.

4. The server receives the request, and proceeds to decrypt the opaque via symmetric encryption using its secret XORed with the nonce. The server now has access to the session key, user id, and session expiration time with no database access. These values are used to verify the HMAC. The request is dropped if the HMAC is invalid or if its individual expiration time has expired.

This method, dubbed OTC, has some advantages. It prevents XSS because the session key is not accessible via JavaScript, rather the HMAC was implemented as a browser extension. It provides message integrity via the HMAC, and prevents replay via the per-request expiration time, recommended to be 30 seconds. Perhaps the greatest advantage is that it requires no database access, even for the user id. The only attack of those described above to which it is vulnerable is CSRF, again because the browser performs the HMAC on behalf of any request to a given domain.

Some disadvantages of OTC were observed in the reference implementation source code provided by the author. It did not authenticate HTTP headers, presumably because some headers are modified in-transit by web servers and reverse proxies. It did not authenticate the request body of POST or PUT requests. It also does not offer absolute replay prevention.

3.2.6 SecSess

Published in 2015 by De Ryck et al., SecSess [15] is another HMAC-based protocol for maintaining authenticated user sessions. Its primary contribution is the way in which the HMAC secret key is determined by the client and server. Their claim is that by using the Hughes variant of the Diffie-Hellman key exchange protocol [27], the client is able to choose the HMAC key in advance of sending the first request, and thus authenticate to the server their intent to begin a session. Upon receiving the first request, the server allocates some persistent storage for both a Session ID (with associated state) and an HMAC key, which it will know eventually. It responds to the first request with its public component of the DHE, Y. The client then continues to HMAC subsequent requests and provide its public component of the DHE, X.

Their claim is that this prevents the scenario which "allows an eavesdropper to respond to the first response, injecting his key material into the session". They are referring to the following scenario with a non-Hughes Diffie-Hellman exchange.

1. A legitimate end-user makes a request to begin a new session with no HMAC
2. The server, using traditional D–H, sends the parameters:
   (a) p, the prime modulus
   (b) g, the primitive root modulo p

(c) $A = g^a \bmod p$

(d) The Session ID

3. A attacking man-in-the-middle responds with $B = g^b \bmod p$, to the server, and stores the Session ID
4. The attacker responds to the legitimate end-user with its own choice for the D–H parameters and Session ID.
5. The attacker ignores the HMAC of the requests from the client, modifies the client's requests, and signs them using the secret upon which the attacker and server agreed.

By using the Hughes variant of Diffie-Hellman, it would appear that the attacker cannot interpose with its own secret in this way, because the client chooses the secret in advance. However, this is not the case. The attacker can still perform the following sequence of operations.

1. A legitimate end-user requests to begin a new session with an HMAC, and provides the D–H parameters p and g
2. The attacker, with man-in-the-middle capability, then makes a similar request to the legitimate server.
3. The server responds to the attacker with the public component of its newly and randomly generated Hughes parameters, Y, and a Session ID.
4. The attacker makes another request to the server with its public component of the Hughes parameters, X, corresponding to the attacker's own request in step 2.
5. The attacker responds to the legitimate user with the server's response, including the Session ID, but with the public component of newly and randomly generated Hughes parameters, Y.

6. The legitimate user responds to the attacker with its public component, X, and believes it has shared its secret with the legitimate server.

7. The attacker now ignores the HMAC of requests from the client, modifies the client's requests, and produces its own HMAC for requests using the obtained Session ID and the attacker's chosen secret.

The reason that this is possible is that neither version of Diffie-Hellman support authentication. The "public key" used by the server is randomly generated for each exchange, and is not signed by some well-known authority. This was a critical oversight in the development of SecSess. The notion that there is some advantage to being able to send an HMAC with the first request also holds no weight. The purpose of the HMAC is to validate that a request is genuine, and drop any response if it is not. The fact that the server will respond to the initial request without being able to validate the HMAC means that the first HMAC has no purpose. SecSess goes even further in permitting responses without first validating the HMAC, "SecSess addresses [dropped messages] by continuing to send the [client's] public component as long as the server has not confirmed the session establishment."

An additional disadvantage of SecSess is that it does not specify the input to the HMAC. It uses this property to demonstrate that it operates in the presence of caching proxies, claimed to be a benefit. However, if an attacker can replay a request and receive a response from a cache, then there is no replay prevention mechanism enabled by the HMAC. Despite this, SecSess has at least one good design property. It explicitly specifies that session secrets must be persisted in a secure location, inaccessible from on-page JavaScript.

3.3 SessionArmor Compared to Existing Methods

Like many of the methods just discussed, SessionArmor uses an HMAC to authenticate user requests in the context of an HTTP application. However, the way in which it

uses the HMAC differentiates it from all previous methods. SessionArmor is completely specified in terms of its HMAC inputs, which gives it ideal request integrity. Robust replay prevention is implemented using two techniques. Time-based replay prevention is enabled by including a per-request expiration timestamp in the HMAC. Absolute replay prevention is optionally enabled by including a nonce in the HMAC, which is stored server-side using a bit-vector compression scheme for minimal deployment overhead.

SessionArmor has configuration options which allow it to be deployed in the presence of caching proxies, without any loss of request integrity. Web servers and reverse proxies have the opportunity to modify some request headers, which makes it impossible to authenticate them as originating from the client. Therefore, SessionArmor has the option to specify request headers that should be authenticated by the client, and any additional non-standard headers. For maximum efficiency, this configuration is transmitted with each request as a bit vector. By using lightweight configuration transmitted with each request, SessionArmor is completely stateless; meaning it can operate behind load balancers and reverse proxies without any auxiliary database.

SessionArmor does not use cookies or URI fragments to store HMAC keys, instead choosing a secure location inaccessible from JavaScript, preventing Cross-Site Scripting vulnerabilities. Unlike all other protocols presented in this section, the use and negotiation of cryptographic algorithms in SessionArmor is fully specified, making it future-proof and hardened against cryptanalysis. SessionArmor also defines a limit on the length of time for which a session is valid, eliminating the Session Extension attack and reducing the attack surface *in time* for Session Hijacking opportunities. This is further strengthened by specifying the notion of an inactivity timeout, and implementing it with a stateless technique. In addition SessionArmor makes preventing Session Fixation a top priority, and will only begin to secure a session if it can prove that a new session credential has been generated for a recent authentication event.

4. The Session Armor Protocol

Having discussed existing mitigation techniques for the Session Hijacking attack, we now present the overall design and features of SessionArmor. As motivating design criteria, seven goals were laid out before beginning formal specification or implementation of the protocol. They included goals of security, performance, and ease of deployment. These are listed in section 4.1.

In the following section are the major features and attack mitigation techniques of the protocol. These are a high-level description of what makes SessionArmor novel and valuable in the landscape of HTTP authentication mechanisms. For undesirable features that are specifically avoided one can refer to section 3.3.

4.1 Goals of the Protocol

Session Armor aims to meet the following design criteria:

1. Individual Request Authentication

   A man-in-the-middle cannot tamper with request data to perform actions on a user's behalf.

2. Replay Prevention

   An attacker cannot replay requests to perform actions on a user's behalf.

3. Minimal Overhead

   Less than 100ms overhead for session setup and 10ms overhead per request round trip.

4. Easy Deployment

   No server-side storage should be required, except for that required by nonce-based replay prevention, if enabled.

5. No Application Modification

   Web application code often relies on the existing Session ID as a bearer token for a variety of purposes. Session Armor *leverages* this key rather than *replaces* it, so that no changes are needed in the application code.

6. TLS Required During Setup Phase Only

   Even on an unencrypted channel, request integrity and replay prevention are guaranteed once a session has been established.

7. TLS Protected Applications Benefit

   Secrecy and whole-payload integrity are not provided, or the protocol would be a wholesale re-implementation of TLS. Session Armor can, and should, be used in conjunction with TLS to mitigate Cross-Site Scripting attacks, and exfiltration of session tokens behind the TLS endpoints.

4.2 Features and Mitigation Techniques

4.2.1 Fully Specified and Configurable HMAC

Session Armor supplants bearer tokens by generating and including a hash-based message authentication code with each client request. With this property alone, an attacker cannot perform unlimited actions by capturing requests, as the data of each request is authenticated individually by the server. The HMAC algorithm used for a given client is chosen by the server from a set presented by the client. This gives the server an opportunity

to choose an algorithm that it trusts for security and performance reasons.

In addition to allowing for a choice of HMAC algorithm, Session Armor also provides a robust mechanism for the server to choose portions of the request that will be protected by the HMAC. These inputs are specified in their entirety, unlike prior protocols as discussed in the previous section. The server can indicate **arbitrary header fields** and **non-standard header fields** that it would like the client to authenticate. This configuration data is transmitted **per-client with no server-side state needed** in a compact bit-vector format.

4.2.2 Secure Storage of HMAC key

As discussed in chapter 3 some existing proposals to protect against Session Hijacking remain vulnerable to XSS. They store the session secret in the DOM, in Cookies, or in the document location. Given how often XSS vulnerabilities are discovered in even high-profile web application, Session Armor takes the stance that the risk of this attack surface is too high to be allowed. Therefore, HMAC keys must be stored in a location inaccessible to client-side scripts, including browser extensions. Some browsers do not offer this type of protected storage location. Thus, a browser standard for secure storage of session data would be a pre-requisite to a true implementation of SessionArmor as intended. Luckily, Google Chrome, which was used as a platform for the reference implementation, does offer isolation of browser extension data.

4.2.3 Careful Choice of Cryptographic Primitives

SessionArmor makes use of two cryptographic primitives which are essential to its operation, a hash function and a block cipher. Any time that these powerful tools are used, care must be taken to use them correctly. In the case of HMAC, there is only one way to use the underlying hash function, and so the strength of the HMAC relies entirely upon the cryptographic strength of the hash function; its tendency to avoid collisions, and produce

apparently random bit flips for any modification to an input bit. To allow for an upgrade path to stronger cryptographic hash functions as they are designed, SessionArmor includes a bit-vector with every message indicating the hash algorithm in use, which is itself protected by the HMAC.

The use of a block cipher is somewhat more complex. Block ciphers can operate in various modes of encryption, which use as input previous output data, counters, and external functions to varying degrees. SessionArmor uses a block cipher only on the server-side, so the application implementor is in complete control of its operation. There are some modes of operation which have known vulnerabilities, e.g. Electronic Code Book mode is quite susceptible to frequency analysis, and Cipher Block Chaining mode has a chosen-ciphertext vulnerability known as the Padding Oracle attack which is described in section 5.3.2. To prevent this class of attack, SessionArmor specifies that a strong symmetric cipher is used in conjunction with a known-robust Authenticated Encryption with Associated Data (AEAD) mode of encryption.

4.2.4 Time-based Replay Prevention

Even with request integrity, individual request replay cannot be prevented without additional measures in place. Replay can be devastating: the most straightforward example being a malicious request for a repeated financial transaction. To combat this, Session Armor uses either time-based or nonce-based replay prevention. With time-based replay prevention, a timestamp is included in-the-clear and authenticated using the HMAC. The server is able to verify that the timestamp has not been tampered with and that some expiration time, e.g. 4 seconds, has not elapsed.

In addition to enforcing individual request expiry. SessionArmor treats session expiry as a top priority. Sessions cannot last longer than 30 days, and sessions are expired if inactive for a configurable inactivity period. Inactivity timeout leverages the HMAC and is

implemented using a stateless technique discussed in section section 5.4.2.

4.2.5 Nonce-based Replay Prevention

If the application developer wishes to configure Session Armor with a persistent back-end, such as a relational database or key-value store, then robust, nonce-based replay prevention can be used. In this scheme, a monotonically increasing counter is included with each request and authenticated using the HMAC. Session Armor stores a bit-vector indicating nonce values that have already been seen, and denies any repeated requests. A bit-vector lookback scheme is required because web requests are often made concurrently and arrive at a server out-of-order.

5. Formal Specification of the Session Armor Protocol

For any cryptographic protocol to be useful, a formal specification is in order. This specification should include an exact and detailed description of all data transferred to and from any entity in each stage of the protocol, and the purpose of each data field. A syntax should be employed which allows the specification of required data, optional data, lists of data, field names, and other operations of the protocol. The specification should include example messages for each message type, in an encoded and unencoded form. There should also be a description of the protocol in prose that clarifies any potential confusion or misunderstanding that may arise during implementation. Diagrams should be used to introduce concepts that have effective visual metaphors. The reader of such a specification should take care to adhere to the use of the terms, "must", "should", and "may", in addition to their negations. This chapter presents such a formal specification for SessionArmor.

5.1 Syntax of Parameter Specifications

This document uses the standard parameter specification format employed by Unix manual pages.

- Required parameters are enclosed in angle brackets:
  <required>

- Optional parameters are enclosed in brackets:
  [optional]

- Choices are a comma separated list enclosed in braces:
  {choice1, choice2, choice3}

- Variable length parameters end with a comma then three dots:
  [param1, param2, ...]

  Extensions to these conventions are the following:

- A value may be the result of a function invocation, indicated by a name immediately followed by parameters enclosed in parenthesis:
  base64(param1 | param2)

- Concatenation is denoted with the pipe operator. Whitespace surrounding the pipe operator should not be included in the value:
  (part1 | part2)

5.2 Formatting Conventions for HTTP Headers

The standard for formatting HTTP headers can be found in section 3.2 of RFC 7230. [18] Standard HTTP headers are often formatted as as hyphen-separated title case names, followed by a colon, followed by a single space character, followed by a bare value. HTTP headers themselves are separated by the sequence `x0Dx0A` or `\r\n`. If a web application developer wishes to introduce a non-standard header, the convention is to prefix that header with the sequence "`X-`", for example the "`X-Forwarded-For:`" header, which is used to indicate the originating IP address when an HTTP request is forwarded through a proxy or load balancer. Based upon the standard and these conventions, the headers that Session Armor will use will be prefixed with "`X-S-Armor:`".

5.3 Setup Phase

5.3.1 Client Setup Phase

Both the web client and web server must run an implementation of the Session Armor protocol in order for it to function. Ideally, Session Armor is built into web browsers on the client-side and web servers on the server side. For a client to indicate that it supports Session Armor, it sends a header with every request to all web servers indicating that it is ready to begin a Session Armor session. This header also includes the set of HMAC algorithms that the client supports. Because this header is sent with every client request, it must be small in size to present minimal overhead in bandwidth consumption.

The client provides to the server with the header `X-S-Armor:` with every request. The initial value used by the client is:

```
X-S-Armor:  r:base64(<byte0><byte1>[byte2,...])
```

The value of the header, which is used for all phases of the protocol, is always semi-colon separated key-value pairs. Each pair is of the format `<key>:<value>` with the `:` being surrounded by no whitespace. This is similar to the format used by the Set-Cookie header [6]; however, `:` is used instead of `=` because the values in this protocol are often base64 encoded, which uses = as a padding character. The only key in this initial header is `r`, which means 'ready'.

The value of 'r' is a variable-length bitmask. The first `<byte0>` is the number of bytes to follow and `<byte1>[byte2,...]` are those bytes. The payload of this header value, `<byte1>[byte2,...]`, is a bitmask indicating the HMAC algorithms supported by the client. A given bit in the bitmask with a value of 1 indicates that the client supports one of the following algorithms. A client must support one or more of these algorithms. This list may be expanded over time as part of the protocol standard. Note the absence of SHA-1, as recent attacks have been developed that undermine its cryptographic strength. [45] For

Table 5.1: Hashing Algorithm choices in Client Support Bitmask

bit	algorithm
0	SHA-256
1	SHA-384
2	SHA-512
3	RIPEMD-160

an explanation of why HMAC-SHA3 is not currently an option for the HMAC algorithm see section section 5.6.

Note that this algorithm negotiation is "one-sided". That is, the server must support all of the specified algorithms, as there may exist clients which each only support one of the algorithms. Contrast this with the TLS handshake, in which the server may refuse service to a client not supporting any of the cipher suites that it does. Future versions of SessionArmor might support such refusal as attacks are found on these algorithms; but, there is an even more severe issue with weakened negotiation in general. This characteristic can lead to a so-called *downgrade attack*. That is, if a man-in-the-middle were able to control even this one byte of this client ready message that indicates HMAC support, the MITM could force the use of the weakest option. Such a downgrade attack is what enabled the POODLE attack on SSL 3.0 to be useful in practice. Browsers choose to diverge from the TLS specification and retry failed TLS connections with SSL 3.0. [36] The reason that we do not consider this a weakness in SessionArmor is the requirement that the first request-response pair be conducted over TLS, which thwarts all MITM attacks if used correctly.

Shown in fig. 5.1 is an example of the client ready header for a client which supports SHA-256 and SHA-512 only. First, the un-encoded header is shown, with binary notation, then, the actual encoded header is shown, with base64 notation.

RIPEMD-160 is a 160-bit cryptographic hash function, designed by Hans Dobbertin, Antoon Bosselaers, and Bart Preneel in 1996 with support from the German Information Security Agency and the Katholieke Universiteit Leuven. It was intended as a stronger

```
X-S-Armor:  r:0b0000000100000101
X-S-Armor:  r:AQU=
```

Figure 5.1: Client ready header in un-encoded and base64 notation

replacement for the popular 128-bit hash functions at the time: MD4, MD5 and RIPEMD. It has a 160-bit output, similar to SHA-1, although unlike SHA-1, no weaknesses in RIPEMD-160 have been publicly disclosed. Therefore, RIPEMD is currently considered a strong alternative to SHA-1 and in some cases SHA-256, when the use of those algorithms is not desirable.

5.3.2 Server Setup Phase

A server that supports Session Armor must take no further action in the Session Armor protocol until the following conditions have been met. *At the time of response*:

1. An HTTPS connection is currently established with the client and will be used to send the current response.

2. The server has prepared a `Set-Cookie:` header, containing a Session ID, to send with the current response.

Condition 1 must be met because the server will send an HMAC key, which must be kept secret, to the client with this response. It is assumed that if these two conditions are met, then the client has just authenticated with the server by sending credentials over a secure tunnel. Therefore, rather than using a cryptographic key exchange mechanism, Session Armor takes advantage of the existing secure channel to send the HMAC key. In the case of a web application framework, which may operate on requests that have TLS terminated upstream by a reverse-proxy or load balancer, the `HTTP-X-Forwarded-Proto:`,

or `HTTP-X-Forwarded-Protocol:` headers may be used to determine the protocol in-use. Ensure in this case that the proxy is properly configured to overwrite the value of this header with 'http' in the case that a malicious party has sent this header to the server themselves.

These requirements do pose a weakness: the server chooses an HMAC key with no input from the client. This means that if the secure tunnel has been compromised, then the Session Armor session has been compromised as well. However, if this is the case, then the user's long-term authentication credentials have also been compromised, and protecting any session that begins with these credentials is no longer possible. This choice was made in support of goal 3 and 7: minimal overhead, and supplementing TLS rather than re-implementing it.

Condition 2 must also be met for the current response. Initially, a cookie-based session may or may not be active with the sever, i.e. the client may have a session cookie that the server is using as a key to persist state related to that client's current activities on the site; a so-called "anonymous" session. In this case, historically, many web frameworks would authenticate a user by bringing up a short-lived HTTPS connection, performing username and password authentication, and *continuing to use the existing Session ID*. This was a logical flaw that enabled a vulnerability called *Session Fixation*, in which an attacker obtains a Session ID from the server and uses XSS or network techniques to plant this Session ID in a target client. The next time that the client authenticated, the attacker would have access to an authenticated session token for that user.

The proper mitigation for this attack is to generate a new Session ID *on the same request in which username and password authentication* is taking place, transfer existing session data for the user to the new token, and issue this new "transferred" Session ID to the authenticated user. An initial Session ID `"Set-Cookie:"` from the server often takes place in the clear, because this token will be sent in the clear in all future requests for the duration

of the anonymous session. Therefore, by using the two conditions above, Session Armor can assume that the user has just authenticated, if and only if the web framework has properly mitigated against Session Fixation. If it has not, then the SessionArmor protocol will not continue, and the user will be authenticated via HTTP Cookie with a stale Session ID. If possible, SessionArmor detects this case and logs an error for the developer indicating that there is no mitigation for Session Fixation in use.

Protecting the existing Session ID distinguishes Session Armor from other bearer-token-avoiding techniques for HTTP authentication. This is because it leverages rather than replaces the existing authentication mode of web frameworks. SessionArmor can authenticate anonymous session data that has been "upgraded" to authenticated session data, a common pattern that is not explicitly specified by other protocols in this domain.

In general, the name of the Session ID cookie must be accessible to the server-side Session Armor middleware. Luckily, in many frameworks, e.g. Django, Ruby on Rails, PHP, and J2EE, the name of this Cookie is a globally accessible and thus does not need to be configured specifically for SessionArmor by the developer.

Once the two conditions above are met, then the following actions are performed by the server to begin a session. They should be performed in this order for the sake of performance, i.e. "fail fast".

1. Server verifies that the *only* key-value pair in the X-S-Armor header is `r:<value>`

2. Server *removes* the Session ID cookie from the `Set-Cookie:` header, but leaves other Cookies intact within the header. The Session ID is stored in-memory for use in the next step.

3. Server generates a nonce `n` that will be used as the initial value for counter-based replay prevention. Counter-based replay prevention is optional. It is the only feature

of Session Armor that requires persistent state for the protocol itself on the server-side, besides Session data itself. `n` must be 32 bits.

4. Server encrypts the Session ID using a secret shared among all servers, `k`
5. Server chooses a key `Kh`, that will be sent to the client for HMAC use
6. Server sends an `X-S-Armor:` header with the protocol initialization data to the client.

The protocol initialization data are the following keys and values.

`s`: AESGCM(Server Secret, IV, Session ID | Kh | Session Expiration Time)
`iv`: Initialization Vector
`tag`: AEAD tag
`kh`: HMAC key for the client to keep secret
`h`: Chosen HMAC algorithm bitmask <byte0><byte1>[byte2,...]
`ah`: Bitmask of headers to authenticate <byte0><byte1>[byte2,...]
`eahs` (optional) : Extra headers to authenticate <string0>,[string1,string2,...]
`n` (optional): Nonce value for replay prevention

Each of the values are base64 encoded to allow transfer of binary data over HTTP, which is a plaintext protocol. The value of `s` is the "opaque" token containing the sensitive Session ID which is never known to the client. `s` is decrypted by the server on subsequent requests.

The function `E` is a keyed symmetric cipher. This cipher operates entirely server-side, so it can be chosen by the implementer. This cipher must be AES-256 or another of equal or greater strength. AES-256 is recommended due to the fact that many CPUs feature Intel's

instruction set extension AES-NI, which improves performance for this algorithm.

The mode of encryption for E must be Galois Counter Mode (GCM) For a detailed description of why this mode was chosen, see section section 5.3.2. The inputs for an Authenticated Encryption with Associated Data (AEAD) block cipher mode, such as GCM are the following, with their names as used in SessionArmor:

`k` = Server Secret (the key)

`iv` = Cryptographically random nonce generated for each session

`m` = Session ID | Kh | Session Expiration Time (the plaintext)

`a` = h | ah | eah (authenticated data)

The `iv` should be 96-bits as recommended by the GCM spec [34]. A long-standing criticism of GCM is that the authentication mechanism is **broken entirely** if an attacker can observe any ciphertext for which the same nonce and IV have been used twice. [28] Therefore it is critical that the nonce be unique for every SessionArmor session and include epoch time as a component. The "authenticated data" for AESGCM are the protocol parameters, concatenated in their raw binary form, which are available to the server with each request. This ensures that an attacker has not tampered with the set of headers that will be authenticated, or the HMAC bitmask. This is an anti denial-of-service mechanism of the protocol, as modifying these parameters would simply cause the server's HMAC validation to fail.

The value of `tag` is an auxiliary block of data produced by GCM, it authenticates that the plaintext, `m`, and authenticated data, `a`, were encrypted with knowledge of the secret, `k`, which prevents a chosen ciphertext attack. It is 128 bits, and must be sent to the client so that the server can decrypt `s` on subsequent requests with authentication.

The key used for `E` is "Server Secret" a shared secret among all servers, which must

be rotated so that it is always less than 30 days old. Transitions are handled by trying the second-most-recent key if initial decryption fails.

The value of `Kh` is the HMAC key to be used by the client. It is chosen in a cryptographically random fashion by the server for each client session.

The value of 'Session Expiration Time' is the final second for which the session is valid in Unix epoch time. Sessions must not last longer than 30 days.

The value of `h` is the HMAC algorithm chosen by the server, it uses the same bit vector format `and length` sent by the client, but with only one of the bits set to 1, to indicate the chosen HMAC algorithm.

The key `ah` stands for "Authenticated Headers". This is a bit vector using the same dynamic length format that's used for the HMAC algorithm selection. A bit being high in this bit vector indicates that a certain header should be authenticated. The list of authenticatable headers are part of this standard and can be read in order, with descriptions, as part of the reference implementation in Appendix Appendix A.1. The masks are generated dynamically from this order via bit-shifting with the declaration of the variable `AUTH_HEADER_MASKS` seen there. The first bit of this mask indicates whether or not the server is choosing to use nonce-based replay prevention, note that this field must be included in this bitmask to prevent denial of service: an attacker could attach a nonce to a request not using nonce-based-replay prevention. If so, there would be no other means to determine whether or not the nonce should be included in the HMAC computation.

The key `eah` stands for "Extra Authenticated Headers"; it is a list of custom headers that should be authenticated, as required by the web application. For example, the header `X-Content-Type-Options` is popular for protecting against MIME type modification. [43] The strings are the full, hyphenated, names of the headers to be authenticated, delimited by commas with no surrounding whitespace.

The `ah` and `eah` values can be thought of as the "expectation" feature of Session Armor.

This feature allows the server to indicate which fields in the request should be authenticated. Ideally, the entire request, headers and body, would be authenticated by the client. However, this is not always practical, or possible. For example, in previous session protection protocols, it was recognized that certain header fields were not accessible to client implementation frameworks such as browser Add-On APIs. This led to a reduction in integrity checking. For example, in the reference implementation of OTC [14], the only field authenticated was the path string. Because SessionArmor authenticates the set of chosen headers using AEAD encryption, it strengthens the authentication of client requests.

The primary reason that both standard and non-standard HTTP headers must be configurable for an ideal HTTP HMAC are middle-boxes. We wish the client to authenticate *all* request data to which it has access at the time of request. However, some request headers will be modified or appended by proxies and load-balancers, thus rendering them unable to be authenticated. These headers include `Accept Encoding:`, `X-Forwarded-For`, and `Via:`. [40] Once the headers are altered, we need a mechanism by which we can determine what headers the client was actually able to authenticate at the time of the request. We therefore assume that the web application is aware of a minimal set of headers that will not be touched by middle-boxes. A demonstration of this issue can be seen in figure fig. 5.2.

The value of `n` is the nonce for counter-based replay prevention. It is optional and should be chosen in a cryptographically random fashion by the server. Note that either `eah` or `n` or both can be omitted without affecting client parsing of this initialization response.

Note that the global protocol options can be changed on the fly for new sessions, because the protocol is stateless. Sessions established using old settings will continue to present those options to the server until the session expires. This implies that if nonce-based replay prevention is used, and it is subsequently disabled, the nonce-cache must remain accessible for one period of the maximum session lifetime.

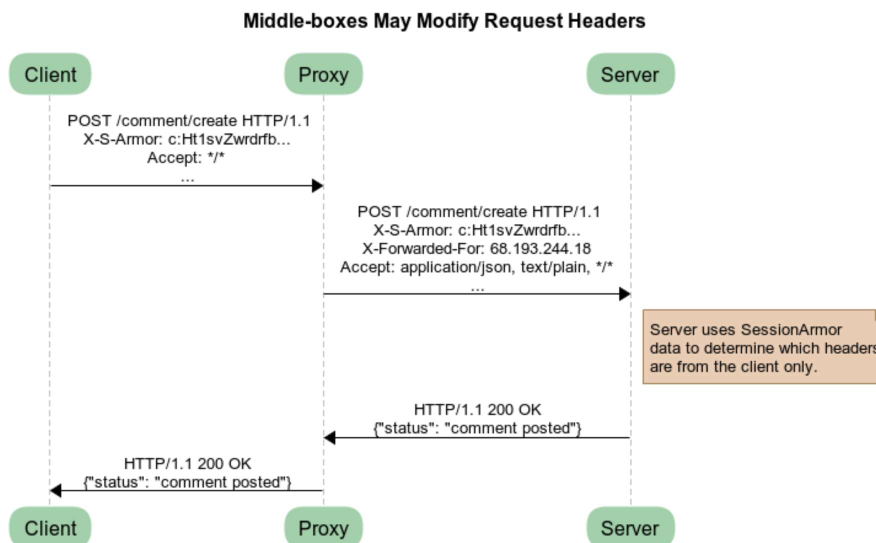

Figure 5.2: Proxies may modify or append to the headers of a client request

Shown in fig. 5.3 is an example of the server header for the establishment of a new session. First, the un-encoded header is shown with hexadecimal and binary notation as appropriate, then, the actual encoded header is shown, with base64 notation.

Mode of Encryption for the Block Cipher

The mode of operation for the cipher E must not be Electronic Code Book, ECB mode, as was suggested by Session Lock [2]. The reason for this is the nature of of the data being encrypted, namely the Session ID concatenated with the HMAC key. Session IDs are sometimes insecurely chosen by web frameworks to be sequential and monotonically increasing. SessionArmor prevents against this weakness by encrypting the SessionID with the use of an IV. If such an incrementing Session ID were to exceed the block size of AES, 16 bytes, which is likely, then the first block of every opaque token would be identical for all clients for a given period (during which the more-significant bits of the Session ID remain static). ECB would create a situation in which many encrypted blocks are identical, and reveal to an attacker where entropy lies in the generation of Session IDs.

```
X-S-Armor:  s:0x387A9CB5E8D8817B3261EC9CB8185DE9DF0435C859726
9C1D708D4DB07984239AE6C0D922A4E150F230D485948A70C2CED7C05E4B9
62898B33BB8;iv:0x59BFFC5E538F5DDF43107792;tag:0xF0852356E9B2F
6463961639B349824F2;kh:0x66E4DDBF507582A61DFA9A077A3AE9FA;h:0
b0000000100000100;ah:0b00000011000011010111001111101111;eah:X
-Client-App-Version,X-Legacy-App;n:2901798076

X-S-Armor:  s:jxu9utnbbD8d34INnPSWnqB6jZCDkluZydlWoe6Ran6hFZS
gvAU0hyzRpLPPdSJtbdpG1Vgg78fM9/Jf;iv:Wb3TrH92Eec5BV6O;tag:q7V
04PfTaDkx+Ro41yhz8Q==;kh:uvSBPp/Zh+E7B4dzp8g4hg==;h:AQQ=;ah:A
w1z7w==;eah:WC1DbGllbnQtQXBwLVZlcnNpb24sWC1MZWdhY3ktQXBw;n:3a
kgLw==
```

Figure 5.3: Server header for the establishment of a new session in un-encoded and base64 notation

A popular alternative mode of operation is Cipher Block Chaining, CBC mode. This mode uses an initialization vector, and results in all identical plaintext blocks being encrypted to different ciphertext blocks, as long as a truly random IV is chosen for each use of the cipher. CBC mode has a number of drawbacks, one of which is performance. Ciphertext blocks are fed-back into each use of the block cipher, so encryption cannot be parallelized.

The most critical drawback of CBC mode is that it requires the message plaintext to be padded to the block size. Padding can't be avoided because the plaintext is used as direct input to the block cipher. This padding means that CBC mode is susceptible to a type of chosen ciphertext attack called a padding oracle attack [39], the most famous of which is POODLE against SSLv3 [37]. It is carried out by iterating over byte-by-byte modification of the ciphertext, in reverse, to determine the plaintext. It uses the server as an "oracle" which signals to the attacker when the padding of the mutated plaintext is correct. Once the mutated plaintext has correct padding, the original *intermediate* ciphertext can be determined by XORing the mutated plaintext with the mutated ciphertext. With the

original intermediate ciphertext, the original plaintext can be determined by XORing the original ciphertext with the original intermediate ciphertext.

There are two factors at play which make this possible. One, is that single-byte modifications of the ciphertext result in single-byte modifications of the plaintext in a specific block. The second is that the oracle can detect this and may send back different error messages in each case. This attack is mitigated by the server sending back a generic error message, rather than distinct "Error" and "Padding Incorrect" messages. However, the fast failure of an "incorrect padding" case may still reveal this information, such that a timing attack is still possible. Another mitigation is the inclusion of a ciphertext MAC, to prevent any chosen ciphertext attack. However, even the verification of the MAC for the *un-padded* ciphertext can reveal the padding via timing analysis, this is the Lucky Thirteen attack on TLS. [3]

This rules out CBC mode for Session Armor because it is especially susceptible to a timing-based oracle attack. The difference between a successful decryption and an unsuccessful decryption in Session Armor is the difference between a valid Session ID that results in a database or cache query, and an invalid Session ID that results in an unauthenticated HTTP response which can be immediately sent to the client. With a successful but invalid decryption, which is the "probe" of the padding oracle attack, the timing difference would be amplified by this database or cache query, which cannot be made in a constant time consistent with the HTTP error response.

A mode that is not susceptible to padding oracle attacks and does not suffer from performance drawbacks is counter, CTR, mode. Padding of the input message is not needed because in CTR mode the only input to the block cipher is a monotonically increasing counter. This essentially turns the block cipher into a stream cipher, which is XORed with the plaintext to produce the ciphertext.

CTR mode performs well, and is secure against differential and statistical cryptanalysis

as long as a unique counter is chosen for each use. However, CTR mode is still not ideal, as discussed in [22]. A major drawback is that it does not provide cryptographic integrity, i.e. verification to the server that the encrypted message was one originally encrypted using the secret key. This means that in order to prevent an attacker from performing chosen ciphertext attacks, CTR mode must be used with a MAC of the ciphertext itself. In Session Armor's case, where the data being encrypted is itself an HMAC key, the overhead of supplying this "ciphertext MAC" in both directions approaches 50%, so it is not the best choice.

Modes of encryption that additionally provide integrity verification are called "Authenticated Encryption" modes. Six such modes have been standardized by ISO. For a number of reasons, one of them reigns supreme in current practice. Galois Counter Mode, GCM is the standard AE mode in IEEE 802.1AE, next generation Ethernet security, IEEE 802.11ad, next generation WiFi security, FC-SP, the Fibre Channel Security Protocol, IPSec, SSH, and TLS 1.2 [42]. GCM performs well, as both encryption and decryption can be performed in parallel, it does not require padding of the plaintext message, it provides message integrity, and Intel has dedicated an instruction to one of its most costly operations, PCLMULQDQ, which has been implemented since 2010 in x86 designs from all major vendors. Also, according to the authors of [34], GCM is not patent-encumbered.

In addition to these excellent properties, GCM, allows for the authentication of "extra" data sent in the clear, which makes it an AEAD mode, Authenticated Encryption with Associated Data. This is especially useful to Session Armor, because there is parametric data included with each request from the client that can and should be authenticated by the server, namely, the HMAC algorithm that has been chosen for the session, `h`, the authenticated HMAC header field bitmask, `ah`, and the extra authenticated headers, `eah`. Authenticating these values as associated data using the block cipher mode of encryption is a boon to the integrity of Session Armor against protocol downgrade attacks. These

would otherwise be protected only by the HMAC computed by the client. Without server-side integrity, an attacker would be able to forge an HMAC for a downgraded set of header fields if the HMAC key was obtained.

5.4 Session Phase

5.4.1 Client Session Phase

When a client receives the TLS protected `X-S-Armor` header from the server, it decodes and stores the parameters for the session.

When any data is stored by the client for the purposes of a Session Armor session, it is keyed by the *fully qualified domain name* of the server that sent the response. Thus, unlike cookies, a subdomain may not store a Session Armor token for a parent domain. This prevents the "Cookie Stuffing" vulnerability, in which a rouge subdomain overwrites cookies for a parent domain in a malicious fashion. In doing so this enables a variant of the Session Fixation attack, and other more benign issues related to cookie overwriting, all of which are mitigated by Session Armor.

It is recommended that built-in language support is used for storing and decoding bit-vectors, as many CPU architectures feature an instruction that can be used for quickly finding the most-significant set bit in a word.

The values of `s`, `IV`, and `tag` are each components of the opaque token containing the encrypted Session ID. They are stored by the client without modification. `Kh` is also stored without modification.

The value of `h`, the bitmask indicating which HMAC algorithm will be used for the session, is stored without modification, and may also be stored in another form such as a string or function reference. The requirement is that this value can be used by the client to quickly determine which HMAC algorithm to use for authenticating requests to a given

domain.

The values of `ah`, and `eah` are the bitmasks indicating which request fields are to be authenticated, and are stored without modification. They may also be stored in an unpacked fashion. The strings contained in `eah` are stored unmodified as the unpacked form of `eah`. The requirement here is that the client can quickly determine which fields must be authenticated by the HMAC.

If the first bit of `ah` is 1 and `n_c` is not present, or if `n_c` is present and the first bit of ah is 0, then an error is thrown and the session data for this domain is discarded. This should be silent to the user, who is prompted by the server to authenticate again upon subsequent requests. Recall that these setup actions should only be performed on a secure response with a new Session ID from the server, to prevent denial of service attacks against the client.

If `n_c` is present, then the first bit of `ah` is checked. If it is 1, indicating that nonce-based replay preventions is to be used, then `n_c` is stored in a way that allows it to be easily retrieved and incremented on subsequent requests.

Client authentication of requests

For each request, the client computes the following HMAC. Example input to the HMAC, with output, can be seen in fig. 5.4.

```
c : HMAC(Kh, [n] | "+" |
         t | lt |
         ah_val0 | [ah_val1 | ah_val2 ...] |
         eah_val0[, eah_val1, eah_val2 ...] |
         path |
         request body OR "")
```

The value of `n` is the current value of the nonce counter, included if the first bit of `ah` for this session is 1.

```
HMAC(0x3283416f2060c83f154ea762b20559ef,2736056|+|15057731
23|1505773123|127.0.0.1|Mozilla/5.0 (Macintosh; Intel Mac
OS X 10_12_5) AppleWebKit/537.36 (KHTML, like Gecko) Chr
ome/52.0.2743.116 Safari/537.36|text/html,application/xhtm
l+xml,application/xml;q=0.9,image/webp,*/*;q=0.8|gzip, def
late, sdch, br|en-US,en;q=0.8|https://127.0.0.1:8000/login
?next=/|/|)

0x1edd6cbd9c2b76b7db7ca2f16d722d079aa3ed5bea46f3de68bef92c
f5e7618ed235e728ffb972d7b0c625f7302778b98e447341a85a2caf65
4ce9918ef09b29
```

Figure 5.4: Example HMAC input and output, with key and output encoded in hexadecimal notation. SHA-512 is in use

The "+" between `n` and the rest of the parameters is the single character '+'. It is always included. This is used to distinguish the nonce from the request expiration timestamp in the server-side implementation. That is, when absolute replay prevention is not being used, the string being authenticated begins with '+'.

The value of `t` is the request expiration timestamp in Unix epoch time. It is the final second for which the request is valid. The recommended value is 4 minutes after the request time given the expected time-to-live of TCP traffic. The value of `lt` is the Unix epoch timestamp of the time at which the last request was sent. It is used by the server to determine if the session should be invalidated due to inactivity. It is included in the HMAC so that a man-in-the-middle cannot invalidate the session by setting `lt` to a value far in the past.

The value of `h` is the HMAC algorithm bitmask, with the chosen bit set. The value of `ah_val0 | [ah_val1 | ah_val2 ...]` are the values of the headers to be authenticated in the order of the `ah` bitmask.

The value of `eah_val0[, eah_val1, eah_val2 ...]` are the extra headers to be authenticated, as comma separated strings, in the order of the strings included in `eah`.

The value of `path` is the value of the `HTTP` path being requested, e.g. `/` or `/users/list?sort=lastname`. It is always included. The value of `request body` is the full body of the request. For example, if the request was a POST request and included post data, this would be the full post data beginning at the first byte after the `<CR><LF><CR><LF>` sequence following the last HTTP header.

After computing the HMAC, n_c is incremented by 1. The client then sends the following data to the server in an X-S-Armor header. An example of this header can be seen in encoded and un-encoded form in fig. 5.5, with fields encoded in hexadecimal and binary as appropriate.

```
c: Client HMAC result
t: The time of the request
lt: Last request time
iv: Initialization vector
tag: AEAD tag
h: Chosen HMAC algo bitmask
ah: Authenticated headers bitmask
eah: List of extra headers to authenticate
(optional) [n]: Nonce value for replay prevention
```

5.4.2 Server Session Phase

When the server receives a client request that includes the `c` key in the Session Armor header, the server knows that the request is for an already established session rather than a new session. At this point, the server must validate the request.

First `s` is decrypted using the server secret, `k`, along with the `IV` and `tag` provided in the client header. During GCM mode decryption, the `tag` is used to authenticate `s` automatically. This provides the server with three things: an HMAC key, a Session ID, and a Session Expiration Time.

```
X-S-Armor:  c:0x1edd6cbd9c2b76b7db7ca2f16d722d079aa3ed5bea46
f3de68bef92cf5e7618ed235e728ffb972d7b0c625f7302778b98e447341
a85a2caf654ce9918ef09b29;t:1505773113;lt:1505773123,iv:0x59B
FFC5E538F5DDF43107792;tag:0xF0852356E9B2F6463961639B349824F2
;kh:0x3283416f2060c83f154ea762b20559ef;h:0b0000000100000100;
ah:0b00000011000011010111001111101111;eah:X-Client-App-Versi
on,X-Legacy-App;n:2736056

X-S-Armor:  c:Ht1svZwrdrfbfKLxbXItB5qj7VvqRvPeaL75LPXnYY7SNe
co/7ly17DGJfcwJ3i5jkRzQahaLK9lTOmRjvCbKQ==;t:WcBGOQ==;lt:WcB
GQw==,iv:Wb/8XlOPXd9DEHeS;tag:8IUjVumy9kY5YWObNJgk8g==;kh:Mo
NBbyBgyD8VTqdisgVZ7w==;h:AQQ=;ah:Aw1z7w==;eah:WC1DbGllbnQtQX
BwLVZlcnNpb24sWC1MZWdhY3ktQXBw;n:Kb+4
```

Figure 5.5: Client header for one request in un-encoded and base64 notation

The value of `t`, the Session Expiration Time, is immediately compared against the current time to determine if the session has expired. If it has, the request should continue to be processed by the application with no Session ID attached. This treats the request as unauthenticated for the given endpoint. The response must include the Session Armor invalidation token as described in section section 5.5.

When performing session invalidation in this way, it may seem necessary to recompute session expiry (and necessarily recompute the HMAC in order to validate the values of `t` and `th`) as part of a hook at *response time* in addition to when it was computed at *request time*. This would be the case if requests are not mutable or some other request-specific auxiliary storage mechanism is not available, and a hit to performance would result. However, a good solution is to mark the request as expired by adding the header `X-S-Armor-Invalidate`. The response hook can check for this header and invalidate the session. Note well that this `X-S-Armor-Invalidate` header is meaningless to the client! Section section 5.5 describes an HMAC procedure for invalidation of the session that prevents denial of service.

Next, it is important that the server authenticate the remaining header values before proceeding to use them. By using `h`, `ah`, `eah`, and the request headers and body the server reconstructs the input to the HMAC. The server has decrypted `Kh`, the HMAC key, and uses the algorithm indicated by `h`. The request is only accepted as valid if the client's value `c` matches exactly the result of the HMAC computation. This simultaneously validates the request data, the request's expiration time, and the value of the nonce counter. If the value of `c` does not match, the request is rejected.

Rejecting the request means that the server returns a 403 response or redirects the request to the login page, whichever behavior the web application would have employed with an invalid Cookie-based session. The server must not invalidate the session. A request may be rejected for a number of reasons, not all of which require that the server discard the investment in the setup phase. For example, in especially congested network conditions, a request might be received more than four minutes after it was sent, resulting in an expired request being received by the server for innocuous reasons.

Once `t` and `lt` have known good values, they are used by the server to determine if the session should be expired due to inactivity. This is a way to perform an inactivity timeout with no server-side state, made possible by the fact that the client can authenticate any data for use by the server, including its last request time. This feature is depicted in fig. 5.6

Next, the session expiration time, `t`, is compared against the current time to determine if the *request* has expired. This is "time-based replay prevention". If `n` is present, then the server performs the following operations for "nonce-based replay prevention". This is absolute replay prevention. First, we describe some prerequisites. The server uses a persistent storage medium to retrieve the most recently seen nonce. This medium should be keyed by the Session ID and shared by all servers. A good place to store these values might be a shared cache with a cache timeout configurable to infinity. A bitmask indicating all recently seen nonces should also be stored per Session ID. These are n_r and n_b, the recent

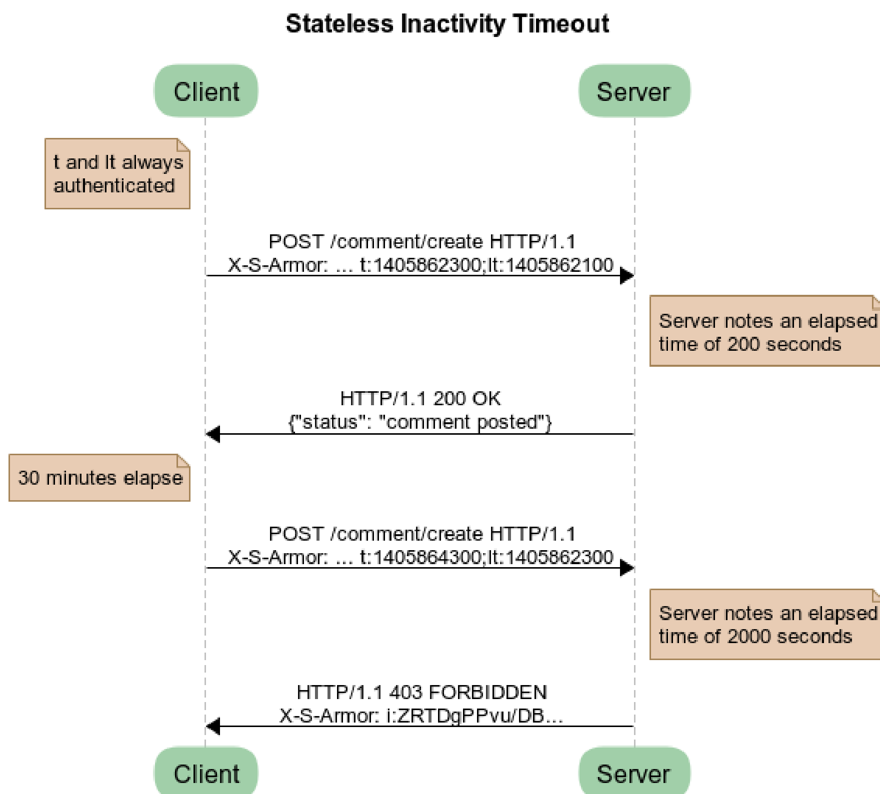

Figure 5.6: HMAC of last-request-time enables a stateless inactivity timeout

nonce and the nonce bitmask. It is recommended that the bit-length of n_b be the word size of the server's CPU for optimal performance, and it must be at least 32 bits. This allows for up to 32 in-flight requests from the same session, and 64 or more if a larger bit-vector is used. The following procedure is used to determine if a nonce should be accepted. Each of these cases is illustrated in fig. 5.7.

1. n_r and n_b are retrieved from the storage medium.

2. The client's nonce is now evaluated, which we now call n_c. First, $n_r - n_c$ is computed. If it is greater than `length(`n_b`)` then the nonce is too old. It has "fallen off the end" of the bitmask, and the request is immediately rejected.

3. If it is greater than or equal to 0, then the bitmask is checked to determine if the nonce has been seen before, by shifting an LSB to the left by $n_r - n_c$ and performing a bitwise-and. If the target bit is set to 0, then the request proceeds with a response and that bit is set to 1 in n_b. If the bit is set to 1, then the request is rejected as a repeated request.

4. If it is less than 0, this means that n_c is the most recently seen nonce. n_r is set to n_c and n_b is shifted by this offset so that the first bit would correspond to n_r. The request proceeds with a response.

The The bitmask n_b is being used here as a compact storage mechanism the most recently seen nonces. If a request is ever rejected by nonce replay, it is important to delete the stored values of n_r and n_b so that they are not incorrectly used to reject requests when a new session is established.

A valid request means that the server attaches the Session ID in the request's `Cookie:` header, using the Session ID cookie name expected by the web application. This cookie is set only on the *server backend* for the purposes of identifying the client using the existing

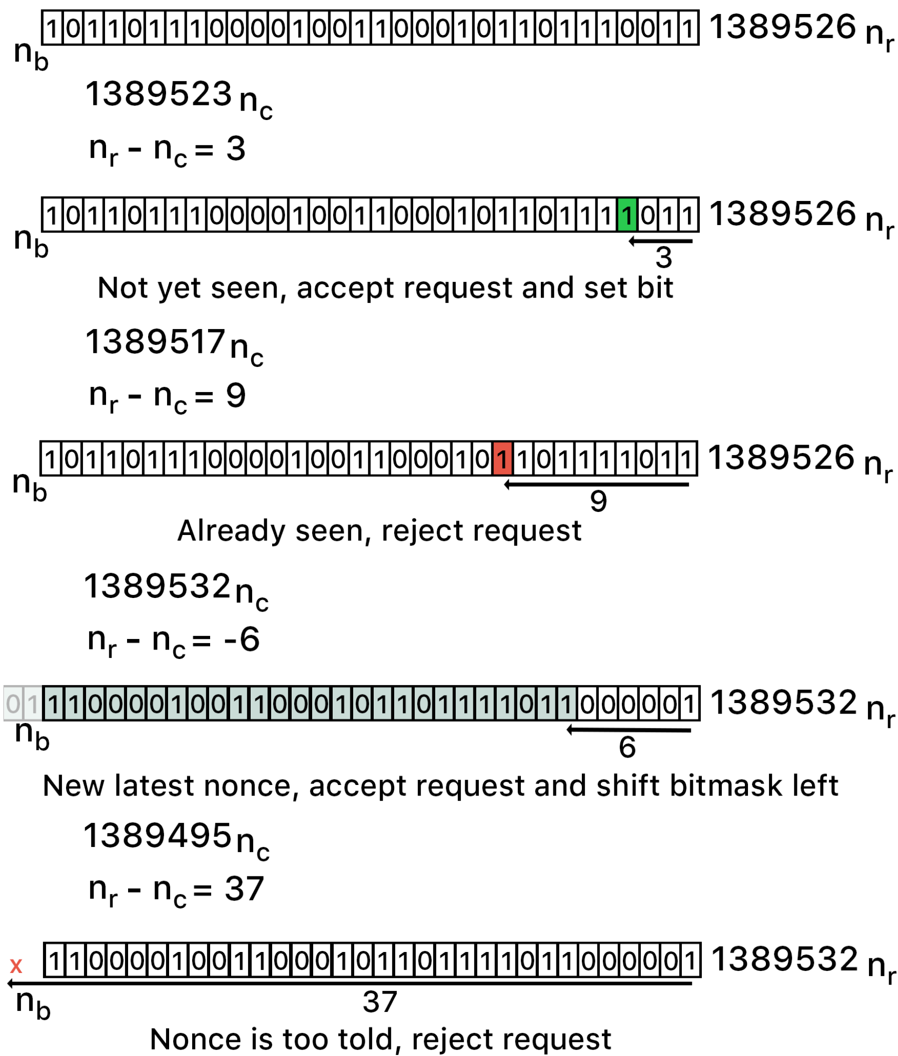

Figure 5.7: All possible cases of evaluating a request nonce for replay prevention purposes

```
X-S-Armor:  i:HMAC(0x3283416f2060c83f154ea762b20559ef, Ses-
sion Expired)

X-S-Armor:  i:ZRTDgPPvu/DBtdE/47IA0r4yhJfzix8vsIHXzXcP2896Fq
1KwwbhBHuuQWCqrnLiJ/OezAeGWs+BcJz/B+GKIQ==
```

Figure 5.8: Invalidation header first showing parameters and then HMAC result.

session management and permissions system. A corresponding `Set Cookie:` header must not be sent to the client.

5.5 Session Invalidation Phase

Only the server knows the expiration time of the session. It's contained in the opaque token. This is to prevent the "Session Extension" attack mentioned in section section 1.4. When a session has expired, the server sends the following X-S-Armor header data to the client to invalidate the session. It uses the HMAC algorithm chosen for that session. An example header, first shown with the parameters and then with the resulting HMAC can be seen in figure fig. 5.8. SHA-512 is in use.

```
i:base64(HMAC(Kh, "Session Expired"))
```

When the client receives these values it performs the HMAC using the secret HMAC key which it has stored for this domain. If it validates, then the client deletes all Session Armor data that it has stored for this domain. If it has no Kh for this domain, then the message is ignored. This "invalidation validation" is to prevent denial of service on the client by a man-in-the-middle sending this invalidation header field.

With this invalidation phase, all possible message types in SessionArmor have been presented. A diagram which summarizes the protocol can be seen in figure fig. 5.9.

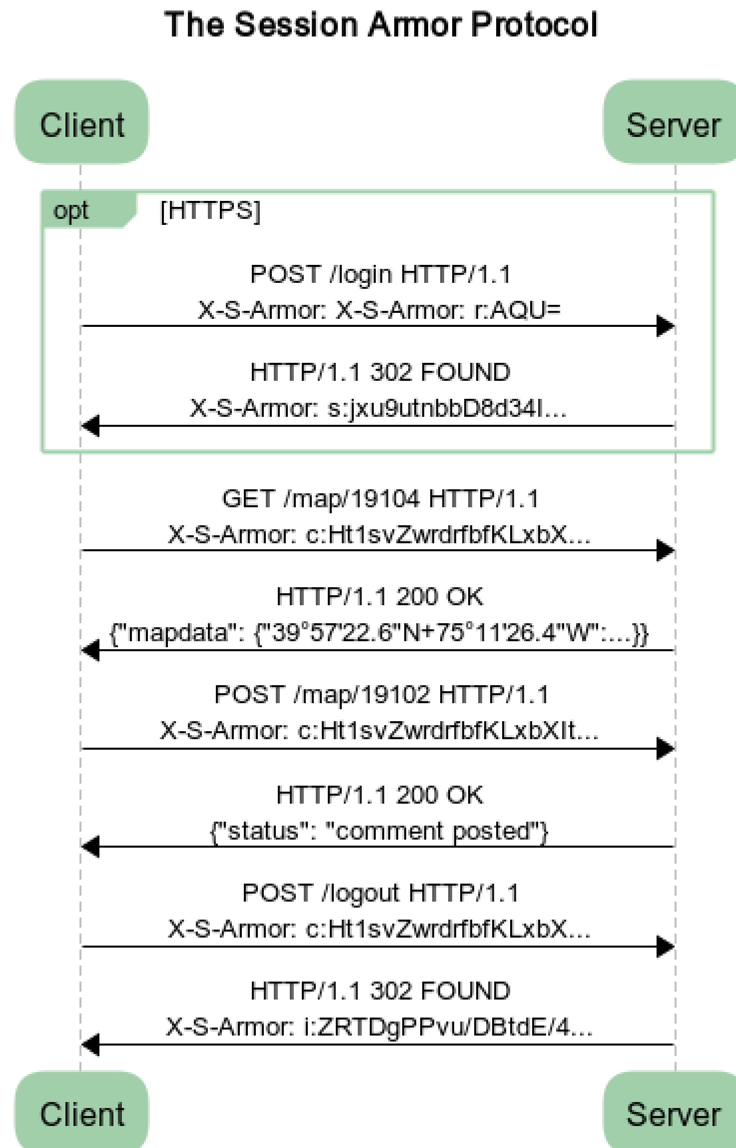

Figure 5.9: Example run of the SessionArmor protocol

5.6 On the Use of HMAC-SHA3

The authors of the SHA-3 algorithm, Keccak, believe that HMAC-SHA3 would not require the "ipad-and-opad" scheme of the generalized HMAC, because unlike SHA-1 and SHA-2, Keccak does not have a length-extension weakness. Instead, MAC computation can be performed by simply prepending the message with a key. [9] NIST has recently presented a standard for using Keccak in an authentication context, a "KMAC", [29] which has not been widely adopted. The predominant Python implementation of SHA3 warns against using it for the purposes of constructing an HMAC.

In addition, whenever an HMAC algorithm is widely deployed and in use, test vectors for the algorithm will be readily available from trusted sources. These test vectors are well-known input and output pairs than can be used to test an implementation of the algorithm. For example, test vectors for HMAC SHA-256 can be found in RFC 4231 [38]. There are no such widely distributed test vectors for HMAC-SHA3. HMAC-SHA256 is also considered secure at this time, and there is no pressing need for another cryptographic hash-based authentication system with similar characteristics.

6. Formal Verification of the Session Armor Protocol

Formal verification creates a link from the mathematics that have been developed for notions of secrecy and authentication — to the daily use of cryptography and cryptographic protocols. Perhaps the most widely used cryptographic protocol, SSL (now TLS), which underlies `https://` connections has undergone much scrutiny over the years, as flaws are found and new versions are released. Formal verification of TLS [10] and its underlying cryptographic primitives [5] is an area of research that has received much attention in recent years. Newly developed cryptographic protocols, such as Session Armor, can be verified using formal methods to provide the credential of being provably secure under the abstractions of a given model.

In the literature, there are two abstract models of cryptography that have been used to develop proof-checking systems. One, called the *computational model* treats messages as bitstrings. These messages represent what might be plaintext, ciphertext, or control messages. In this model, functions are defined with precise operations on bitstrings that produce bitstrings as output. One notable example of a verification system using the computation model is the Verified Software Toolchain [4] which combines a static analyzer for a subset of the C language called Verifiable C with a verified C compiler from INRIA called CompCert. This was the toolchain used to create a verified implementation of SHA-256 in [5].

The alternative basis for cryptographic verification is known as the *symbolic model*, which has a more abstract perspective on functions and data. Cryptographic functions are modeled as perfect black-boxes, and messages are terms that exist in an algebra with these functions, including equations. An adversary can compute only with these primitives. This model was first presented by Dolev and Yao [16], and so is commonly referred

to as the Dolev-Yao model. This model was used by Lowe [33] to present a man-in-the-middle attack on the Needham and Schroeder public key protocol, 17 years after it was introduced.

ProVerif is a protocol verification system that operates in the Dolev-Yao model, and is the system that we use to verify Session Armor. It defines a functional modeling language whose equations can be characterized as operating in the *applied pi-calculus*. This is an extension to pi-calculus, or "process" calculus, specifically designed for modeling the domain of cryptographic protocols. Process calculus defines a grammar that specifies notions of infinite repetition and simultaneous execution. For example, here is an excerpt from the proof of Session Armor:

```
process
new server_secret:  key;
new user_id:        bitstring;
new password:       bitstring;
( !(browser(user_id, password)) | !(webapp(server_secret)) )
```

The ! specifies infinite repetition and the | specifies simultaneous execution. So, any properties proven by this verification will be for infinite simultaneous executions of any possible user credentials and server secret. The extended grammar provided by Proverif include *free variables*, *channels*, *functions*, and *events*. Both free variables and channels can be defined as being public, available to an attacker, or private, unavailable to an attacker. Thus, the way we model HTTPS is simply as a private channel, a sequence point in a process at which messages might be observed. Two examples of functions are symmetric encryption and HMAC, which are defined as symbolic equations as follows:

```
(* Symmetric Encryption *)
fun encrypt(bitstring, key): bitstring.
reduc forall plaintext: bitstring, k: key;
```

```
decrypt(encrypt(plaintext, k), k) = plaintext.

(* Message Authentication *)
fun hmac(bitstring, key): bitstring.
```

ProVerif translates processes into algebraic expressions on messages using queries specified by the user. These queries are requests by the user to evaluate one of four security properties. [11]

- Secrecy, expressed as "Reachability"
- Authentication, expressed as "Correspondence"
- Strong Secrecy, expressed as "Observational Equivalence"

The properties tested for SessionArmor were *secrecy* of both `session_secrets` and `server_secrets` and *injective correspondence* of client requests with server responses. These expressions are translated by ProVerif into Horn-clauses, Boolean expressions of the form:

$$p \land q \land \cdots \land t \rightarrow u \tag{6.1}$$

These Boolean expressions, first-order implications, are then checked using satisfiability techniques. [11] When the number of executions of the protocol is unbounded, as is the case with ProVerif, the problem lies beyond the computational domain of NP-complete, and is undecidable, as was shown by [17]. This means that ProVerif will sometimes terminate with a inconclusive result. As a user of the system, one such situation is when ProVerif can negate the Horn-clauses, but cannot terminate at the stage in which this negation is translated back to an actual process for performing the attack (called a "derivation").

Time was spent with this type of no-derivation output for SessionArmor, with attempts to model sequential nonces using ProVerif's persistence features. Because these results resulted in non-termination of the derivation phase, SessionArmor was eventually modeled as a single iteration with an arbitrary nonce. Under this model, both secrecy and authentication were verified. The full model and ProVerif output can be found in Appendix C.

6.1 Proof of Secrecy Using ProVerif

Secrecy queries in ProVerif operate on free variables. The usage pattern is to declare a private free variable, use the variable in a process macro, and include a reachability query to determine if the attacker can observe the private variable in a process replication. Notated, this looks like the following:

```
(* Secrecy queries *)
query attacker(server_secret_test).
query attacker(session_secret_test).
...
out(HTTP, encrypt(server_secret_test, server_secret));
out(HTTP, encrypt(session_secret_test, session_secret));
```

One might see that the free variables are not the target data being tested for reachability, but rather a proxy for testing the reachability of `server_secret` and `session_secret`. Because the target secrets are declared as newly generated within each replication of the process, they cannot be declared as free variables. So, we test instead if an attacker could discover them as keys for encrypting `server_secret_test` and `session_secret_test` with an ideal symmetric cipher. Both server secrets and session secrets were found to be unreachable by an attacker.

6.2 Proof of Authentication Using ProVerif

Authentication in ProVerif is tested using so-called *correspondence* queries. A correspondence query verifies that a given event, with specific associated data is executed only `after` another event with the same associated data. This property represents authentication if the second event occurs in a simultaneously executing process; in this case representing a server actor in a client-server relationship. This stems from the concept of the event occurring just before some privileged action. This type of verification ensures that the action could not take place without first having been authorized by a client holding secret data. Notated in ProVerif, this looks like the following:

```
(* Correspondence Query *)
query
inj-event(serverResponse(session_secret, request_time, request_nonce,
                         request_url, request_data)) ==>
inj-event(clientRequest(session_secret, request_time, request_nonce,
                        request_url, request_data)).
...
event clientRequest(session_secret, request_time, request_nonce,
                    request_url, request_data);
out(HTTP, (request_time, request_nonce,
           hmac((request_time, request_nonce, request_url, request_data),
                 session_secret),
           session_token, request_url, request_data)).
...
let (server_hmac: bitstring) =
hmac((request_time, request_nonce, request_url,request_data), session_secret) in
if request_hmac = server_hmac then
    event serverResponse(session_secret, request_time, request_nonce,
                         request_url, request_data).
```

The type of correspondence query used here provides an additional guarantee known as *injective correspondence*. This constructs Horn clauses that resolve to a test of the first event occurring exactly once before the second event. Meaning that in addition to authentication, this verification ensures that a man-in-the-middle would not be able to manufacture an authentication request and subsequently expect the server to authenticate it. Thus, this validates the property of replay prevention.

7. Reference Implementation of Session Armor

7.1 Server Implementation as a Django Middleware

SessionArmor was implemented as a middleware for the Django web framework. Django middleware are Python modules that hook into the request and response processing of HTTP requests in the context of the Django web application. By observing the state of the `X-S-Armor:` header, and being able to modify it upon response, SessionArmor was implemented with all specified requirements. The full source code for this implementation can be found in Appendix A.1.

In support of goal 4, Easy Deployment, a number of configuration options are provided with sane defaults. These are:

- The number of seconds for which a SessionArmor session should be valid, defaults to 14 days.
- The amount of time for which a request is valid, defaults to 5 minutes.
- The amount of time between requests before a session is expired to due inactivity, defaults to 30 minutes.
- The set of standard headers to authenticate. The defaults are headers which are known to be available to browser add-in mechanisms. These are specified in text rather than in bitmask form, and translated on-the-fly to a packed bitmask, only once upon application startup.
- A flag to indicate if nonce-based replay prevention should be used
- A list of “extra authenticated headers”, two are provided as examples

For additional convenience, the Python logging module is used to set up a logger specifically for Session Armor. When Django is in debugging mode, log messages indicating any request failure modes are logged with a Session Armor prefix. For example, if a client HMAC fails to validate, the following log message appears.

```
[DEBUG] sessionarmor_django.middleware: The client's HMAC did not validate
```

Nonce-based replay prevention is implemented using Django's own caching mechanism. As long as there is a cache defined in Django's `CACHES` setting with the name "sessionarmor", both the nonce values and nonce storage bitmasks for each client will be persisted automatically. For the purposes of performance testing, this cache was configured to use Redis as a backend, with Redis configured to communicate over a Unix socket and never time-out values in its store.

One feature of the source code which may be useful to an application developer is a commented description of all standard HTTP headers that can be used as authenticated headers, in the order that they would appear in the packed bitmask. Lastly, the Python `cryptography` module was used to implement AES-GCM.

7.1.1 Exceptions

A number of exceptional conditions can take place during execution of the protocol that can cause the server to recognize that a request is invalid and respond in a certain manner. The following is a list of those conditions and how the server should respond. These are handled using custom exception classes in the Django middleware for easy reference. The exceptions are allowed to rise to the entrypoint of the middleware to be handled in a centralized area of the code. Messages are provided with each exception type to indicate the reason for the exception, which are logged to the SessionArmor logger when the application is at debug level. In all of these cases except for Session Expiration the server

responds with 403, Permission Denied, and does not allow the request to be processed as usual.

Invalid HMAC algo mask

If a ready-state client request contains an HMAC algorithm bitmask that does not have any bits set that match the set of supported HMAC algorithms, then it would appear that the client supports SessionArmor, but the protocol cannot proceed.

Invalid selected HMAC algo mask

If a client signed request contains a selected HMAC algorithm bitmask that does not have a valid bit set for the purpose of selecting a hash module, then the bitmask has been tampered with or is otherwise invalid for the purposes of authenticating requests with the server.

HMAC does not validate

If the client's HMAC does not match the server's HMAC, then it means that the request has been tampered-with in transit, or was otherwise truncated or modified in some way not intended by the client.

Opaque token failed authenticated encryption

In addition to the per-request HMAC of Session Armor, AEAD mode encryption is used to authenticate the ciphertext of the opaque token. The GCM tag is provided to the server with each request from the client. If the tag does not match the authentication code produced at the end of decryption, then then the server assumes that some form of chosen plaintext attack is taking place, and no attempt will be made to validate the authenticity of the request.

Opaque token does not contain the required fields

If, after decrypting the opaque token using the server secret, the server finds that it does not contain all of the required fields for request authentication, then processing of the request cannot proceed. These required fields are: Session ID, HMAC Key, and Expiration Time.

Request expiration

If, after validating the authenticity of the request's HMAC, it is found that the request is too stale, given the setting for request valid duration (5 minutes by default), then the request must be rejected by the server. This is the mechanism for time-based request replay prevention.

Nonce invalid

If nonce-based replay prevention is being used, and a client provides a nonce that has been seen before, then the request is rejected. The request is also rejected if the nonce is too stale, given the storage limitation of the nonce cache. The nonce cache for the Django middleware implementation uses a bit vector per session of one machine-word size. On the test machine, this is 64 bits. So, if the client provides a nonce with a value that is more than 63 values behind the latest value seen, the request is rejected. If the nonce is recent enough, the bit vector is tested.

Session Expiration

There are two conditions for session expiration: absolute expiration and inactivity timeout. As soon as the server decrypts the opaque token, which is one of the first steps of request validation, it checks the absolute expiration time for the session hidden within. If the

expiration time has passed, validation of the request is immediately aborted. On the other hand, the inactivity timeout is checked after the request HMAC has been validated, this is because the request time and prior request time are values contained within the HMAC. The inactivity duration is computed, and if it exceeds the configured threshold, then the session expired.

In both cases, the request *is allowed* to proceed as usual, but a Session ID is not injected. This means that the client will be presented with the response from normal application logic (most likely a 200), but as if the request were made anonymously. In some applications, this may immediately present the login form, which is what the user would expect.

7.2 Client Implementation as Google Chrome Extension

The SessionArmor client was implemented as a Google Chrome browser extension. Extensions are software components that operate around API hooks into the life-cycle of of browser events, such as the HTTP request-response mechanism. The hooks used by SessionArmor in particular were: `onBeforeRequest` to store request body data for HMAC input, `onHeadersReceived` to watch for opaque tokens that begin a session, and `onBeforeSendHeaders` as the primary code path to gather data about each request, perform the HMAC, and build the components of the `X-S-Armor` header.

In order to obtain raw POST data, which is required by the SessionArmor protocol, Chrome had to be modified. By default, the extension API provides request body data in the form of a hash-table. This removes the order information needed for the HMAC to hash the request body exactly as the server will see it. Chrome has the ability to provide the body as a byte buffer, but only does so if the body cannot be otherwise parsed. A modified Chromium browser was compiled making the byte buffer the default behavior.

The extension is seamless in operation for the user. After installation, an icon appears in the user's browser window that indicates the current SessionArmor state for the active tab. If there is an active SessionArmor session, the icon is badged with a green circle; otherwise, it's badged with a red circle. That's it! There's no set-up or user interaction required for a SessionArmor user. Full source code for the Chrome extension can be found in Appendix A.2.

7.3 Performance Evaluation

For a complete picture of the reference implementation, both the server and client were tested for overall performance characteristics. A test application was written, a To-Do List, using Django as a backend with the SessionArmor middleware installed, and accessed via Google Chrome with the SessionArmor extension installed. A screenshot of the application can be seen in figure fig. 7.1. Two tests were conducted to address each of the targets of goal number 3, and baseline performance of the entire HTTP processing time on the server-side was also established.

The first test was of session setup time. Recall from goal number 3 that the target SessionArmor overhead for session setup time was set at 100ms. This fairly lenient goal was set as such because a credential-providing (log-in) event is rare. Users are accustomed to this type of request taking longer than other types and perhaps including OAuth-style redirects. (Note that SessionArmor is compatible with OAuth's "bearer token", which acts as a Session ID)

The test was carried out by executing an `XMLHttpRequest` against the login endpoint of the application in a loop in the console of the web browser. Concurrent requests were made 32 at a time. This number was chosen because at around 64 concurrent requests SessionArmor's nonce vector was observed to be overloaded; which as discussed in sec-

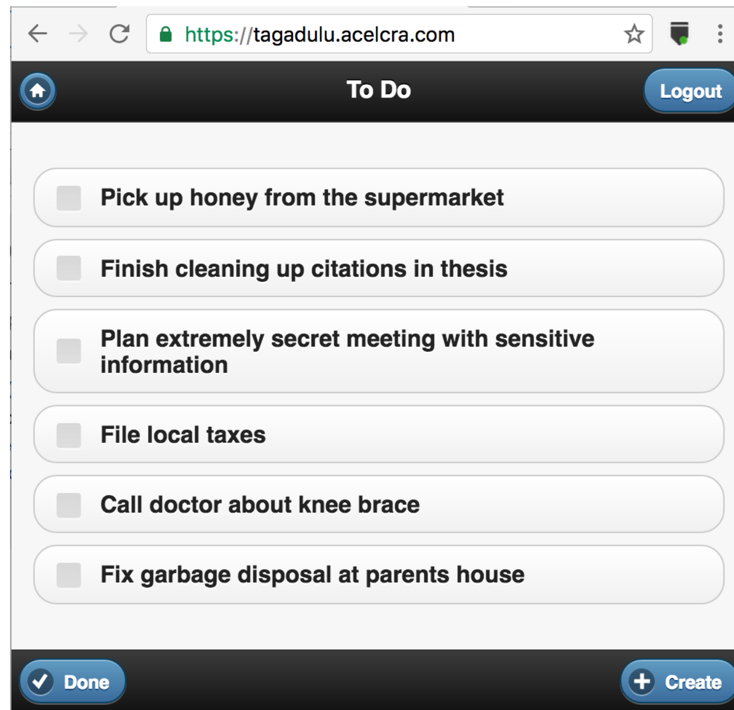

Figure 7.1: To-Do application using SessionArmor live on the Internet

tion 5.4.2 is a tunable parameter, and a trade-off between performance and storage cost.

On the server side, during this test, performance data was collected by instrumenting the return paths of `process_request` and `process_response` which are executed during session creation, to each log to a file their total execution time. These times were then summed to find the server-side execution time. The same was done on the client-side, but for session creation this only involved the return path of a single callback function, and instead of logging, results were appended to a value in `localStorage`. To evaluate performance of the overall HTTP processing time, the request handler of the WSGI server itself was instrumented. To eliminate jitter due to network interfaces, this was preferable to measuring response times at the client. The results can be found in table 7.1 and seen in fig. 7.2 through fig. 7.4.

A few interesting conclusions can be drawn from this data. First, even the sum of the slowest client and server processing times, 1.48 ms, is well below the target of 100 ms. On

Table 7.1: Session Creation Time Performance (ms)

Context	Samples	Min	Median	99%	Max	Mean
SessionArmor Server	128	0.223	0.253	0.521	0.710	0.275
SessionArmor Client	128	0.140	0.175	0.585	0.770	0.195
HTTP Server	128	9.121	39.615	70.705	89.462	33.346

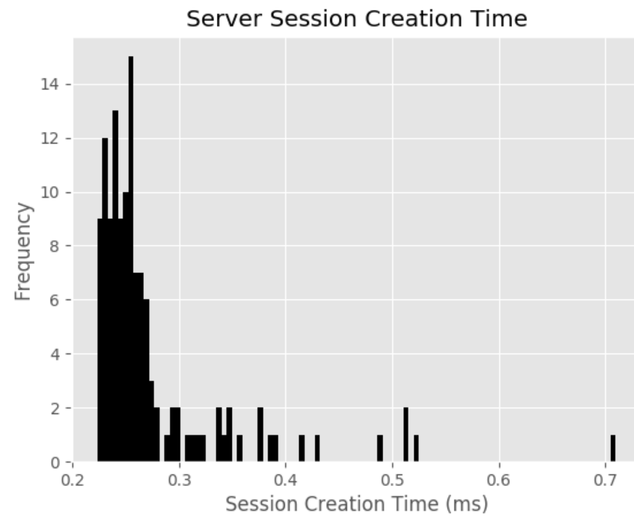

Figure 7.2: Session creation times for SessionArmor server

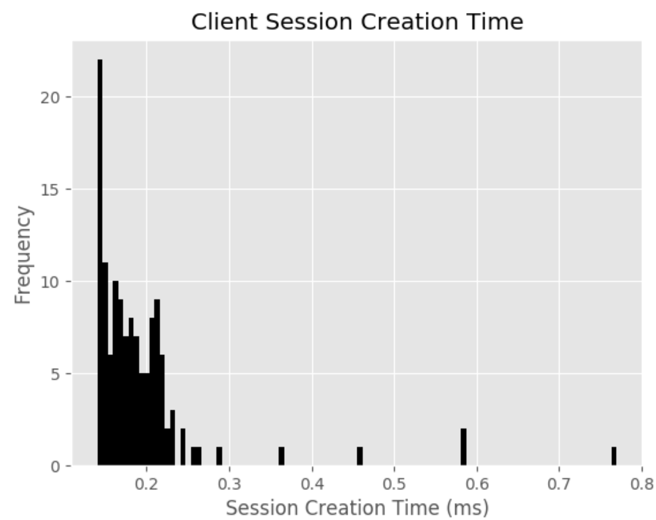

Figure 7.3: Session creation times for SessionArmor client

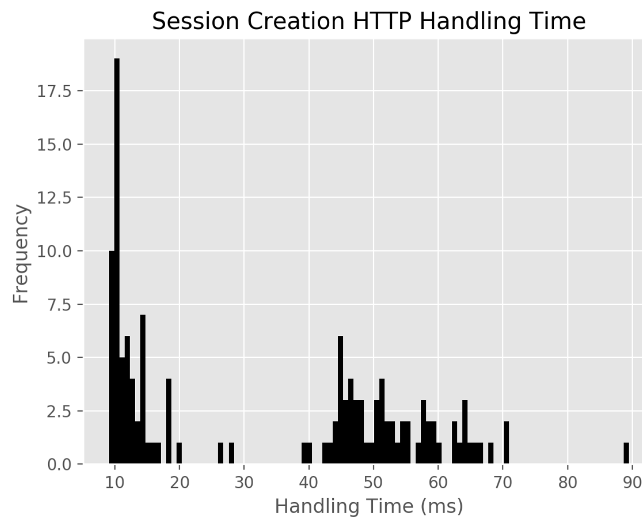

Figure 7.4: Session creation times for HTTP-server

average, the client processing time is 80 μs faster than the server processing time. From the histograms, we can see that most of the server and client processing happens in less than 300 μs, while the HTTP processing time is more bi-modal, with a mean of around 33 ms. This gives us an average overhead of 1.41% for SessionArmor during session creation.

The next test was that of SessionArmor overhead during application usage. The server, client, and HTTP server were instrumented as described earlier to measure their performance. This time, however, the processing time of two callbacks on the client-side, body storage and header processing, needed to be measured and summed in a similar fashion to the server implementation. The test was a POST request, inserting a new To-Do list item with a modest length of 48 characters. The requests were made using the web browser console in concurrent batches of 32.

To test the overhead of each security feature of SessionArmor, each was disabled successively and the test was re-run. First, nonce-based replay prevention, "NBRP" was turned off using the middleware setting. Then, header authentication was turned off by modifying the server and client code to not include it in the HMAC input. The same was then

Table 7.2: HTTP Server – Application Request Performance (ms)

Test	Samples	Min	Median	99%	Max	Mean
Full Protocol	1024	5.597	37.04	187.052	316.021	57.908

Table 7.3: SessionArmor Server – Application Request Performance (ms)

Test	Samples	Min	Median	99%	Max	Mean
Full Protocol	1024	0.605	0.93	2.992	3.97	1.055
No NBRP	1024	0.322	0.528	1.379	2.206	0.552
No Header Auth.	1024	0.253	0.398	1.211	3.153	0.436
No TBRP	1024	0.237	0.370	1.008	1.952	0.393

done for time-based replay prevention, "TBRP". Each feature was disabled in addition to all prior disabled features. The results can be seen in table 7.2 through table 7.4 and fig. 7.5 through fig. 7.13.

Some important conclusions can be drawn from this data. First, the full protocol, in the worst case, adds 7.04 ms overhead, below the target of 10 ms. This is a sum of the worst case client processing time and the worst case server processing time. On average, SessionArmor adds 1.843 ms to each request, an overhead of 3.18%. When nonce-based replay prevention is disabled, a suggested mode of operation for stateless operation, this overhead drops to 1.265 ms or 2.18%. These tests also allow us to see the average overhead of each feature: 578 μs for absolute replay prevention, 279 μs for header authentication, and 155 μs for time-based replay prevention.

Table 7.4: SessionArmor Client – Application Request Performance (ms)

Test	Samples	Min	Median	99%	Max	Mean
Full Protocol	1024	0.455	0.74	1.832	3.07	0.788
No NBRP	1024	0.400	0.685	1.51	2.205	0.713
No Header Auth.	1024	0.335	0.545	1.334	2.400	0.550
No TBRP	1024	0.260	0.420	0.947	1.300	0.438

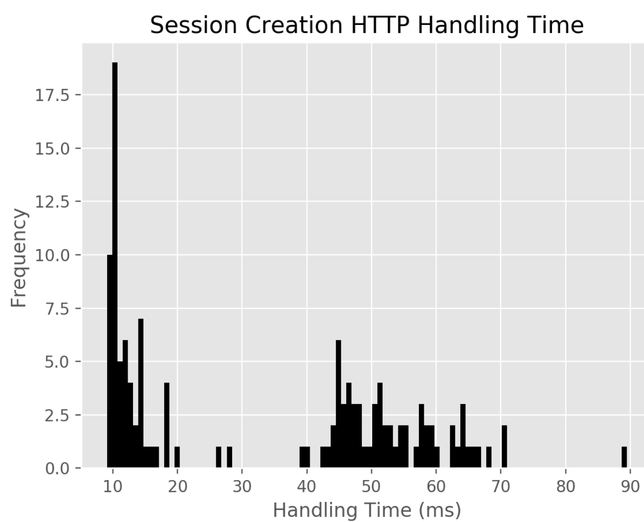

Figure 7.5: Application performance for HTTP Server

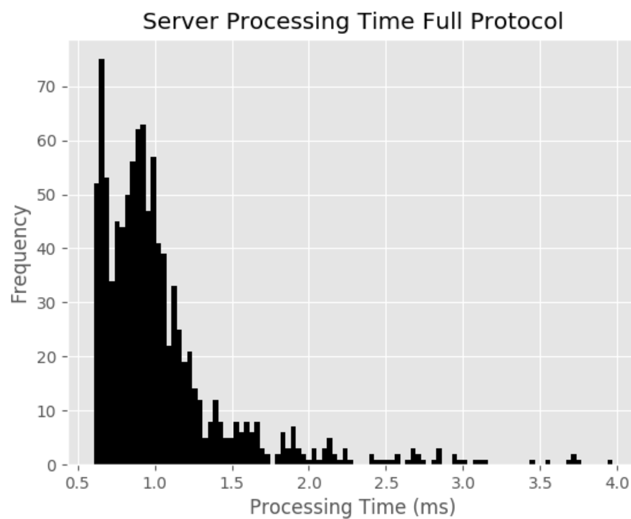

Figure 7.6: Application performance for SessionArmor server, full protocol

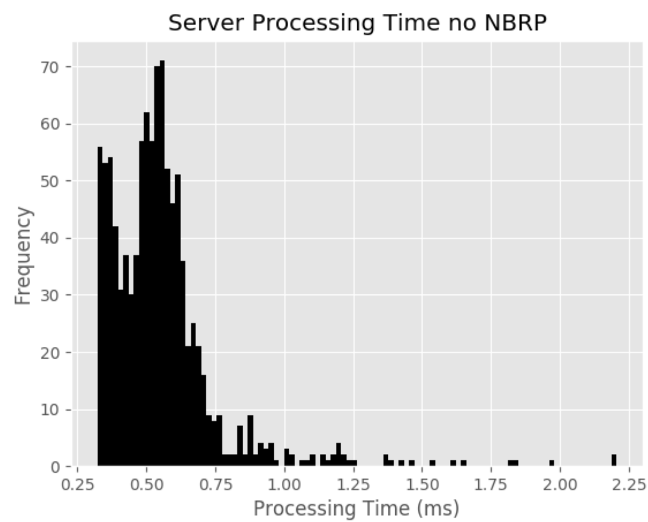

Figure 7.7: Application performance for SessionArmor server, no NBRP

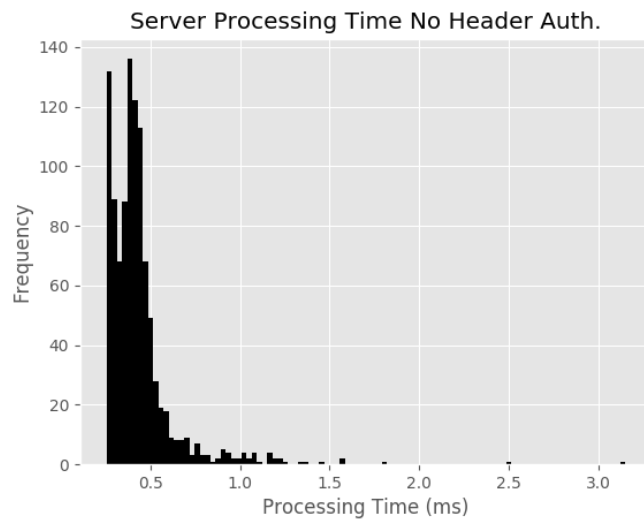

Figure 7.8: Application performance for SessionArmor server, no Header Auth.

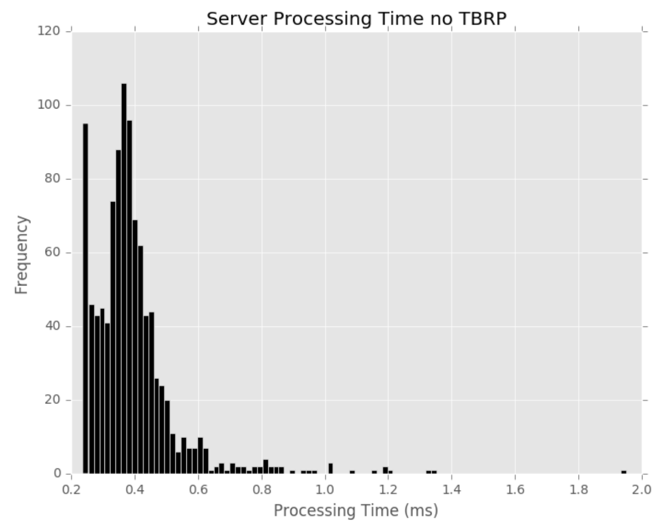

Figure 7.9: Application performance for SessionArmor server, no TBRP

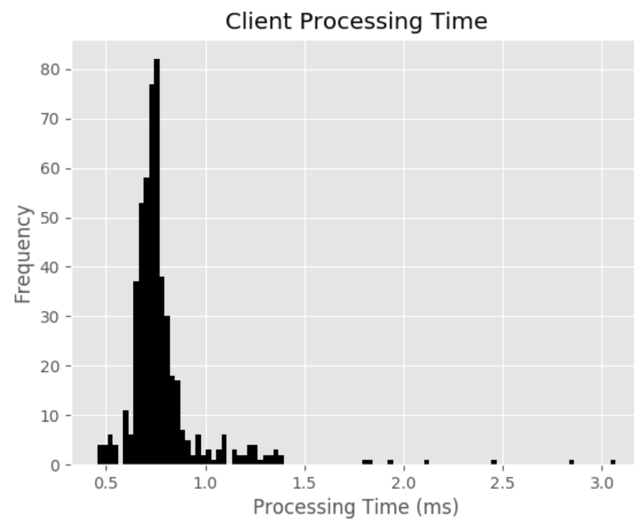

Figure 7.10: Application performance for SessionArmor client, full protocol

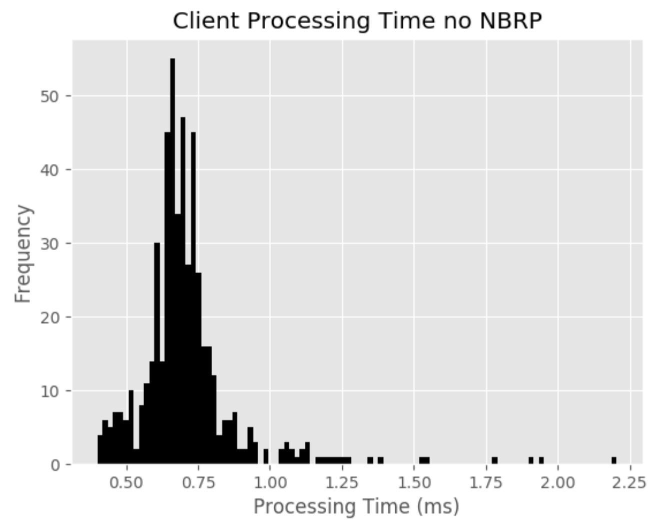

Figure 7.11: Application performance for SessionArmor client, no NBRP

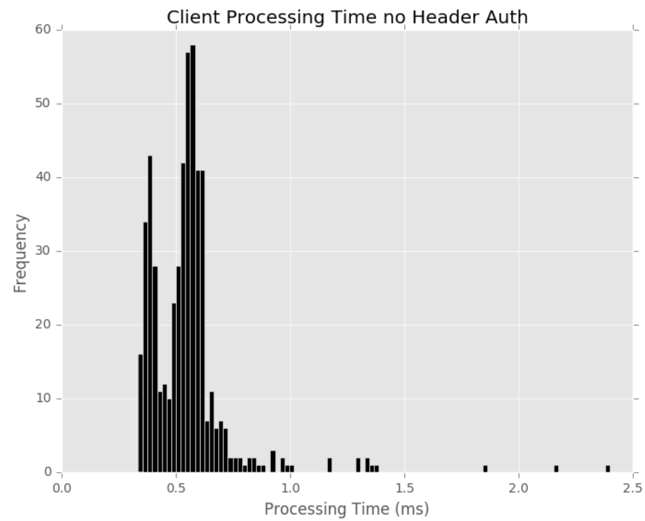

Figure 7.12: Application performance for SessionArmor client, no Header Auth.

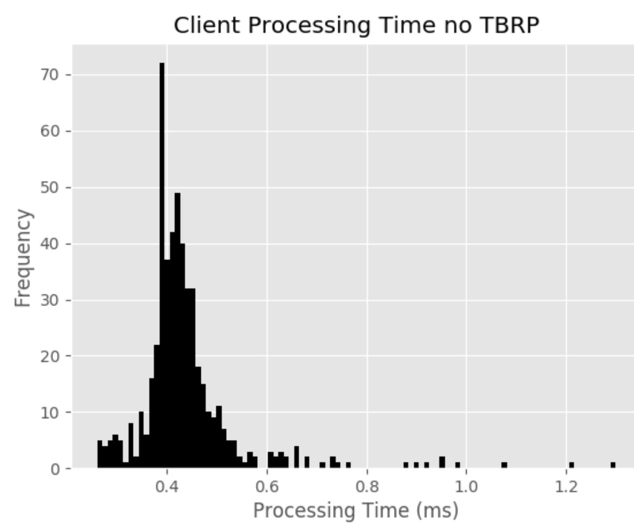

Figure 7.13: Application performance for SessionArmor client, no TBRP

8. Conclusions

There is a mounting trend for nearly all software to use the web as a platform. Even "heavy-lifting" tasks such as document creation, video editing, and 3D-modeling now find their home in a web browser. Government functions, such a tax return acceptance and health care distribution now famously use the web. It would seem paramount that we develop means to protect all users of these systems from leakage of sensitive information. Unfortunately, the predominant means of authenticating users requests is with bearer tokens stored in HTTP Cookies.

These tokens can be obtained through a number of means, and then used to perform unlimited privileged actions on behalf of the user. This is known as the Session Hijacking attack. We presented five means of performing Session Hijacking: Session Sidejacking, Cross-Site Scripting, Session Fixation, Rouge Browser Extensions, and physical access. The Session Extension attack was also presented a means to increase the likelihood of a bearer token being obtained by a malicious party.

There are two common means to protect session tokens from exfiltration, a flag which prevents access to the token via JavaScript, and a flag which prevents sending the token over unencrypted HTTP. Any web application which takes security matters seriously should have these two flags enabled. To observe the use of these protections in the wild, two tools were written. One called SessionJack, analyzed extracted cookie data for potential vulnerability and also session lifetime. Vulnerable cookies were manually loaded into a web browser to verify that they enabled session access. The second tool JackHammer, was created for more quickly performing this test by shuffling tokens between two web browsers using two newly created browser extensions and a websocket server.

Of 108 sites tested, over 30% left themselves open to Cross-Site Scripting, over 50% to

Session Sidejacking, and 100% to Bearer Token Extraction. The sites vulnerable to XSS included some very popular websites according to Alexa rankings, including major banks, business services, and e-commerce websites. The same trend was seen for Session Sidejacking. As far as session lifetime, there was no correlation with the protections enabled or popularity rank, which led to the conclusion that this important parameter does not receive enough consideration.

Existing standards for session token protection were also presented, these included the HTTPOnly flag, the Secure flag, Expiration Time, HSTS, and HTTP Digest Authentication, the last of which is surprisingly an HTTP standard HMAC protocol, alternative to Cookies, which is unfortunately often overlooked simply for user-interface reasons. Seven existing proposals for the protection of web sessions were presented: Fu et al.'s protocol, Liu et al.'s protocol, SessionLock, Web Key, HTTPI, One Time Cookies, and SecSess. Some of the concepts in these protocols would not find their way into the new protocol presented in this work, e.g. using the URI fragment identifier to "pass along" a sensitive token from page to page was simply too susceptible to Cross-Site Scripting attacks. Also, some of these protocols do not include per-request authentication, which makes them entirely vulnerable to request replay. However, a few ideas were borrowed, such as individual request HMAC and its ability to enable time-based replay prevention.

The primary contribution of this work is SessionArmor, a new protocol for protecting user sessions in web applications from being hijacked by malicious parties. SessionArmor accomplishes this using a per-request HMAC, with a secret key transmitted via TLS during a setup phase. SessionArmor provides both time-based and nonce-based replay prevention, making absolute replay prevention possible. It authenticates requests individually, so that no request data can be modified by an attacker, even on an unencrypted channel. To accomplish this goal, it offers configuration of standard and non-standard header data to be authenticated by the server in a stateless manner.

SessionArmor has a number of additional practical features that set it apart from previous work. It allows the the HMAC algorithm to be configured at the start of each session, to future proof the protocol against cryptographic weakness. All configuration is sent with each request in the form of a bit vector to save bandwidth. SessionArmor specifies a maximum session lifetime and a means for stateless inactivity timeout, to prevent long-lived secrets and eliminate the risk of Session Extension. To entirely prevent Session Fixation, a requirement of the protocol is that new session credentials are created upon secret generation. To limit the overhead of implementing nonce-based replay prevention, nonce storage is specified as being compressed in a bit-vector, with a shifting algorithm used to validate to uniqueness of a nonce that appears with a new request. SessionArmor has also been careful to prevent the use of less-modern modes of symmetric encryption, including ECB, CTR, and CBC. It requires the Authenticated Encryption with Associated Data mode GCM, which allows the origin of the server-originated secret and configuration data to be authenticated during decryption.

An implementation of the server was included as a Python Django middleware, and and implementation of the client was included as a Google Chrome extension, both with documented source code. Google Chrome had to be modified to fully support the protocol, in order to process unmodified request body data at at request time. Both are complete implementations of the protocol, including developer-oriented features such as: configuration options with sane defaults, logging with prefixing, and self-documenting exception handling. The client includes a minimal user interface and zero-configuration setup.

Both implementations were performance-tested for session creation time and application request performance. In both cases, the performance vastly exceeded expectations. For session creation time, most of the server and client processing happens in less than 300 μs, with an average overhead of 1.41%. For application requests, most of the server and client processing happens in less than 1 ms, with an average overhead of 3.18%. When

nonce-based replay prevention is disabled this overhead drops to 1.265 ms or 2.18%. The tests also allowed us to see the average overhead of each feature: 578 μs for absolute replay prevention, 279 μs for header authentication, and 155 μs for time-based replay prevention.

Session Armor was formally verified using the ProVerif protocol verification system. ProVerif implements the symbolic model of cryptographic verification using the Dolev-Yao abstractions of a hostile communication environment. Cryptographic primitives such as HMAC and symmetric encryption were implemented in a functional style. Additional primitives provided by the model checker include free and private channels, and the notion of simultaneous execution and infinite repetition. Queries for formal notions of *secrecy* and *injective correspondence* were used to prove that SessionArmor is resistant to man-in-the-middle attacks and request replay.

Appendix A. Session Armor

A.1 Server Reference Implementation, Django Middleware Source Code

---

```
'''
Session Armor Protocol, Django Middleware Implementation

Copyright (C) 2015 - 2016 Andrew Sauber

This software is licensed under the AGPLv3 open source license. See
LICENSE.txt. The license can also be found at the following URL, please note
the above copyright notice. https://www.gnu.org/licenses/agpl-3.0.en.html

Example configuration (place in your app's settings.py}:
S_ARMOR_STRICT = True
# 14 days
S_ARMOR_SESSION_VALID_SECONDS = 1209600
# 5 minutes
S_ARMOR_REQUEST_VALID_SECONDS = 300
# 30 minutes
S_ARMOR_INACTIVITY_TIMEOUT_SECONDS = 1800
S_ARMOR_AUTH_HEADERS = [
    'Host',
    'User-Agent',
    'Accept',
    'Accept-Encoding',
    'Accept-Language',
    'Referer',
    'Cookie',
    'Accept-Charset',
    'Range',
    'Date',
    'Authorization',
    'Origin',
    'DNT',
    'X-Csrf-Token',
]
# Must have a persistent Django cache named "sessionarmor" configured for the
# nonce-based replay feature to work.
# "Persistent" means that the cache is configured as follows:
#    * TIMEOUT is set to None
#    * MAX_ENTRIES is set larger than your max active sessions (maybe millions)
#    * CULL_FREQUENCY is set to float('inf') or culling is disabled
#    * The cache supports no-expiry, by passing None as the timeout
```

```
S_ARMOR_NONCE_REPLAY_PREVENTION = True
S_ARMOR_EXTRA_AUTHENTICATED_HEADERS = [
    'X-Client-App-Version',
    'X-Legacy-App',
]
'''


import base64
from cryptography.hazmat.primitives.ciphers.aead import AESGCM
from cryptography.exceptions import InvalidTag
from datetime import datetime, timedelta
import hashlib
import hmac
import json
import logging
import os
import struct
import time

from django.conf import settings
from django.contrib.sessions.exceptions import InvalidSessionKey
from django.core.cache import caches
from django.core.exceptions import PermissionDenied

COUNTER_BITS = 128
RECIEPT_VECTOR_BITS = 64
# All 1s followed by one 0
# Meaning: The first nonce hasn't been seen yet, but don't allow any
# lower-numbered invalid nonces.
INITIAL_RECIEPT_VECTOR = ((2 ** RECIEPT_VECTOR_BITS) - 2)
RECIEPT_VECTOR_MASK = (2 ** RECIEPT_VECTOR_BITS) - 1

CLIENT_READY = 'ready'
CLIENT_SIGNED_REQUEST = 'request'

HASH_ALGO_MASKS = (
    # these hashing algorithms are in order of preference
    (1 << 2, hashlib.sha512),
    (1 << 1, hashlib.sha384),
    (1 << 0, hashlib.sha256),
    (1 << 3, lambda: hashlib.new('ripemd160')),
)

SECONDS_14_DAYS = 1209600
SECONDS_5_MINUTES = 300
SECONDS_30_MINUTES = 1800

LOGGER = logging.getLogger(__name__)

assert len(settings.SECRET_KEY) >= 32, \
```

```
    "Django settings.SECRET_KEY must be at least {} bytes".format(32)
# use the first 256 bits of the Django SECRET_KEY as the AES key
SECRET_KEY = bytes(settings.SECRET_KEY[:32])

DEFAULT_AUTH_HEADERS = [
    'Host',
    'User-Agent',
    'Accept',
    'Accept-Encoding',
    'Accept-Language',
    'Referer',
    'Cookie'
]

ALL_AUTH_HEADERS = [
    # Hostname to which the client is sending the request
    'Host',

    # String indicating the software and/or hardware platform used to generate
    # the request
    'User-Agent',

    # Types of media that the client would accept in a response
    'Accept',

    # Desired behavior of the connection with the first remote machine
    'Connection',

    # Character encodings that the client would accept in a response
    'Accept-Encoding',

    # Human languages that the client would accept in a response
    'Accept-Language',

    # URI that caused or enabled the client to make the request
    'Referer',

    # Persistent general-purpose tokens that the client provides to the server
    'Cookie',

    # Character sets that the client would accept in a response
    'Accept-Charset',

    # The last modified time known by the client, response requested if
    # modified
    'If-Modified-Since',

    # An entity tag. A response is requested if the entity does not match.
    'If-None-Match',

    # Specifies a portion of the resource being requested
```

```
'Range',

# Time at which a request was sent that includes body data
'Date',

# Authentication credentials provided by the client for Basic or Digest
# HTTP Authentication
'Authorization',

# An indication of how the request should be treated by caching proxies
'Cache-Control',

# A list of origins that caused the request, e.g. used by a client script
# that has established allowable cross-origin methods via CORS
'Origin',

# General-purpose header field, most often used with "no-cache" to request
# a non-cached version of a resource
'Pragma',

# Boolean indicating that the user wishes not to be tracked by the server
'DNT',

# Nonce sent by the server to be used for Cross Site Request Forgery
# protection
'X-Csrf-Token',

# Version of the WebSocket protocol being used
'Sec-WebSocket-Version',

# Used with websocket handshake to indicate what application level
# protocols the client wishes to use
'Sec-WebSocket-Protocol',

# Randomly generated nonce used during the Websocket handshake
'Sec-WebSocket-Key',

# A list of registered websocket extended features that the client wishes
# to use with a websocket connection
'Sec-WebSocket-Extensions',

# Transfer Encodings that the user agent will accept, e.g. "deflate". Can
# also specify that "trailers" should be used for chunked transfers
'TE',

# Mechanism used to make the request, e.g. XMLHttpRequest
'X-Requested-With',

# IP addres or hostname that originated the request (after travelling
# through a proxy)
'X-Forwarded-For',
```

```
# The original protocol used when the request was made, e.g. "https" (after
# travelling through a proxy
'X-Forwarded-Proto',

# Used by a proxy server to include information that would otherwise be
# lost at lower levels in the protocol stack
'Forwarded',

# The email address of the user making the request, most often used by
# robots as contact information for the robot administrator
'From',

# Settings for protocol-upgrade with an HTTP/2 capable host
'HTTP2-Settings',

# Another protocol, to which the agent wishes to switch, e.g. HTTP/2.0
'Upgrade',

# Credentials request by a proxy in the request chain. Consumed by the
# first proxy requesting authentication.
'Proxy-Authorization',

# List of conditions for a resource to meet for a response to be requested
'If',

# An entity tag that must match the resource for a response to be requested
'If-Match',

# Combination of If-Match and If-Unmodified-Since for a range request
'If-Range',

# A timestamp. A response is requested if the entity has not been modified
# since this time.
'If-Unmodified-Since',

# An integer. Used with TRACE or OPTIONS requests to limit forwarding by
# proxies
'Max-Forwards',

# Preferences requested of the server, examples include: asynchronous
# response, relative priority, response verbosity
'Prefer',

# A list of proxies through which the request was sent
'Via',

# Protocol stack that that the client would like to tunnel via HTTP
'ALPN',

# Expected response from the server, usually HTTP 100 (Continue). In this
```

```
# case the client wishes to know if a request body is acceptable before
# sending it to the server.
'Expect',

# Alternative host that the client selected for a request
'Alt-Used',

# Client indicating whether or not it would like timezones on calendars
'CalDAV-Timezones',

# A boolean, indicates if a client will attend a CalDAV calendar event
'Schedule-Reply',

# A CalDAV opaque token for a calendar schedule. A response is requested
# if the resource matches the schedule
'If-Schedule-Tag-Match',

# COPY or MOVE request destination for a WebDAV request
'Destination',

# A URL to a lock. Used with the UNLOCK method to remove the lock.
'Lock-Token',

# Number of seconds for which a WebDAV LOCK should be active
'Timeout',

# A WebDAV URI, indicates the request order of the requested collection.
'Ordering-Type',

# A boolean indicating if a WebDAV resource should be overwritten due to
# the request
'Overwrite',

# A string indicating the desired position at which to insert a resource in
# a WebDAV request
'Position',

# Tree or graph depth of the resource on which the request should act.
# (used by WebDAV)
'Depth',

# Arbitrary text, when present with a POST request, indicates to the server
# a desired description for the content to be used in URIs
'SLUG',

# Set of header fields that will be included with the trailer of a
# message sent using a chunked transfer encoding
'Trailer',

# The Multipurpose Internet Mail Extensions version used when constructing
# the components of the message. Optional.
```

```
    'MIME-Version'
]

# Create a dictionary of masks for the headers which can be authenticated.
# Shift them by thier (index + 1) to leave room for the nonce-based-replay
# prevention indicator.
AUTH_HEADER_MASKS = {name: (1 << (i + 1)) for (i, name)
                     in enumerate(ALL_AUTH_HEADERS)}

NONCECACHE = caches['sessionarmor']


class HmacInvalid(Exception):
    def __init__(self, message="The client's HMAC did not validate"):
        super(HmacInvalid, self).__init__()
        self.message = message

    def __str__(self):
        return self.message


class OpaqueInvalid(Exception):
    def __init__(self, message=
            "The opaque token from the client was not valid"):
        super(OpaqueInvalid, self).__init__()
        self.message = message

    def __str__(self):
        return self.message


class RequestExpired(Exception):
    def __init__(self, message="The request has expired"):
        super(RequestExpired, self).__init__()
        self.message = message

    def __str__(self):
        return self.message


class NonceInvalid(Exception):
    def __init__(self, message="The replay-prevention nonce was invalid"):
        super(NonceInvalid, self).__init__()
        self.message = message

    def __str__(self):
        return self.message


class SessionExpired(Exception):
    def __init__(self, message="The session has expired"):
```

```
        super(SessionExpired, self).__init__()
        self.message = message

    def __str__(self):
        return self.message


def gen_header_mask(auth_headers, absolute_replay_prevention):
    mask = 0

    for header in auth_headers:
        mask |= AUTH_HEADER_MASKS[header]

    if absolute_replay_prevention:
        mask |= 0x01

    return pack_mask(mask)


def parse_header_mask(header_mask):
    mask = bytes_to_int(header_mask[1:])
    headers = []
    bit_n = 0
    while mask:
        if mask & 0x01:
            headers.append(ALL_AUTH_HEADERS[bit_n - 1])
        mask >>= 1
        bit_n += 1
    return headers


def header_to_dict(header, outer_sep=';', inner_sep=':'):
    '''
    Takes a header value string of the form:
    c:<base64data0>;T_re:<base64data0>;h:<base64data1>;ignored0;
    Returns a dictionary:
    {
        'T_re': 1367448031,
        'h': <binarydata1>,
        'c': <binarydata0>,
    }
    '''
    kvs = header.split(outer_sep)
    # remove empty tokens
    kvs = (kv for kv in kvs if kv != '')
    # split key/value tokens
    kvs = (kv.split(inner_sep) for kv in kvs)
    # parse key/value tokens
    d = {kv[0]: base64.b64decode(kv[1]) for kv in kvs if len(kv) == 2}
    return d
```

```
def tuples_to_header(tuples, outer_sep=';', inner_sep=':'):
    """
    Takes a list of (k, v) string tuples and returns a string
    for the Session Armor header value
    """
    encoded_tuples = [(tup[0], base64.b64encode(tup[1])) for tup in tuples]
    return outer_sep.join([inner_sep.join(tup) for tup in encoded_tuples])


def validate_ready_header(header):
    '''
    validate that there is only one header key and it is 'r'
    '''
    return len(header) == 1 and header.keys()[0] == 'r'


def validate_signed_request(header):
    # minimal set of values needed for a signed request
    valid = (header.get('s')
             and header.get('c')
             and header.get('t')
             and header.get('h')
             and header.get('ah'))
    return valid


def get_client_state(header):
    '''
    Given a header dictionary, return the name of the client state.
    '''
    if validate_ready_header(header):
        return CLIENT_READY
    if validate_signed_request(header):
        return CLIENT_SIGNED_REQUEST


def pack_mask(mask):
    '''
    pack an integer as a byte string with this format
    bit length of mask cannot exceed 256

    <num_bytes> <little-endian bytestring>
    byte0       byte1, byte2 ...
    '''
    data = int_to_bytes(mask)
    data = chr(len(data)) + data
    return data


def unpack_mask(data):
```

```
    '''
    unpack a byte string as an integer with this format

    <num_bytes> <little-endian bytestring>
    byte0       byte1, byte2 ...
    '''
    mask = bytes_to_int(data[1:])
    return mask


def using_nonce_replay_prevention(data):
    mask = bytes_to_int(data[-1])
    return mask & 0x01


def int_to_bytes(i):
    '''
    convert an integer to a litte-endian byte string
    '''
    if i == 0:
        return '\x00'
    res = []
    while i:
        res.append(chr(i & 0xFF))
        i >>= 8
    res.reverse()
    return ''.join(res)


def bytes_to_int(bstr):
    '''
    convert a byte string into an integer
    Input: '\x9e\x2c'
    Output: 40492
    bin(Output): '0b1001111000101100'
    '''
    vector = 0
    for i, _ in enumerate(bstr):
        vector += ord(bstr[-(i + 1)]) * (256 ** i)
    return vector


def select_hash_module(packed_hash_mask):
    '''
    Given a bit vector indicating supported hash algorithms, return a Python
    hash module for the strongest digest algorithm
    '''
    hash_mask = unpack_mask(packed_hash_mask)
    for bitmask in HASH_ALGO_MASKS:
        if bitmask[0] & hash_mask:
            return bitmask[1]
```

```
    raise HmacInvalid(
        'HMAC algorithm bitmask did not match any hash implementations.')


def select_hash_mask(packed_hash_mask):
    '''
    Given a header dictionary, select a hash function supported by the client.

    Return a bitmask denoting the selected module.

    1. Decode base64 value of ready header
    2. Parse into bit vector
    3. Select a hash algorithm supported by the client using the bit vector
    4. Return the bitmask for the selected hash module
    '''
    # base64 decode the value of the ready key into a byte string
    hash_mask = unpack_mask(packed_hash_mask)
    # store the bit vector as an integer
    for bitmask in HASH_ALGO_MASKS:
        if bitmask[0] & hash_mask:
            return pack_mask(bitmask[0])
    raise HmacInvalid(
        'Client ready header bitmask did not match any hash algorithms.')


def is_modifying_session(response):
    """
    Is this response creating a new session?
    """
    sessionid = response.cookies.get(settings.SESSION_COOKIE_NAME, None)
    sessionid = getattr(sessionid, 'value', None)
    return bool(sessionid is not None)


def extract_session_id(response):
    """
    Remove a sessionid from a response and return it as a string
    """
    sessionid = response.cookies[settings.SESSION_COOKIE_NAME]
    del response.cookies[settings.SESSION_COOKIE_NAME]
    return sessionid.value


def get_expiration_second():
    """
    Get expiration time for a new Session Armor session as seconds since epoch
    """
    session_duration_seconds = get_setting(
        'S_ARMOR_SESSION_VALID_SECONDS', SECONDS_14_DAYS)
    return str(int(time.time() + session_duration_seconds))
```

```
def generate_hmac_key():
    """
    Generate a new key for use by the client and server to sign requests
    """
    return os.urandom(16)


def encrypt_opaque(sessionid, hmac_key, expiration_time,
                   hash_mask, auth_headers, extra_auth_headers):
    aesgcm = AESGCM(SECRET_KEY)
    # start the nonce off with the current epoch time
    nonce = struct.pack('>I', int(time.time()))
    # add 8 random bytes for a total of 128 bits
    nonce += os.urandom(8)

    plaintext = '|'.join((sessionid, hmac_key, expiration_time))
    auth_data = '|'.join((hash_mask, auth_headers, extra_auth_headers))

    ciphertext = aesgcm.encrypt(nonce, plaintext, auth_data)
    ciphertext, tag = ciphertext[:-16], ciphertext[-16:]
    return ciphertext, nonce, tag


def decrypt_opaque(opaque, nonce, tag,
                   hash_mask, auth_headers, extra_auth_headers):
    aesgcm = AESGCM(SECRET_KEY)

    auth_data = '|'.join((hash_mask, auth_headers, extra_auth_headers))

    try:
        plaintext = aesgcm.decrypt(nonce, opaque + tag, auth_data)
    except InvalidTag:
        raise OpaqueInvalid(
            "Opaque token from the client failed to authenticate")

    try:
        sessionid, remainder = plaintext.split('|', 1)
        hmac_key, expiration_time = (remainder[:16], remainder[17:])
    except ValueError:
        raise OpaqueInvalid(
            "Plaintext from opaque token didn't have required fields")

    return sessionid, hmac_key, int(expiration_time)


def begin_session(header, sessionid, packed_header_mask):
    '''
    Input: client Session Armor headers when in a valid ready state
    Output: server Session Armor headers for a new session
    Side Effects: A nonce-based replay vector persisted externally
```

```
    '''
    # Create opaque token
    # Components: Session ID, HMAC Key, Expiration Time
    hmac_key = generate_hmac_key()
    expiration_time = get_expiration_second()
    packed_hash_mask = select_hash_mask(header['r'])

    eah = get_setting('S_ARMOR_EXTRA_AUTHENTICATED_HEADERS', [])
    if eah:
        eah = ','.join(eah)

    opaque, iv, tag = encrypt_opaque(sessionid, hmac_key, expiration_time,
                                     packed_hash_mask, packed_header_mask, eah)

    kvs = [
        ('s', opaque),
        ('iv', iv),
        ('tag', tag),
        ('kh', hmac_key),
        ('h', packed_hash_mask),
        ('ah', packed_header_mask)
    ]

    if eah:
        kvs.append(('eah', eah))

    if using_nonce_replay_prevention(packed_header_mask):
        n = os.urandom(4)
        kvs.append(('n', n))

    return tuples_to_header(kvs)


def get_setting(attribute, default):
    """
    Returns the value of a Django setting named by the string, attribute, or
    a default value
    """
    try:
        return settings.__getattr__(attribute)
    except AttributeError:
        return default


def auth_header_values(request, header_mask, extra_headers):
    '''
    Returns array of header values in order based on request bitmask

    If headers are not present in the request they are not included in the list
    '''
    headers = parse_header_mask(header_mask)
```

```
    headers = headers + extra_headers
    values = []
    for header in headers:
        if header == 'Host':
            values.append(request.get_host())
            continue

        value = (request.META.get('HTTP_' + header.upper().replace('-', '_'),
                                  None))
        if value:
            values.append(value)

    return values


def server_hmac(algo_mask, key, string):
    digestmod = select_hash_module(algo_mask)
    mac = hmac.new(key, string, digestmod)
    return mac.digest()


def validate_nonce(request_nonce, sessionid):
    request_nonce = bytes_to_int(request_nonce)
    nonce_tup = NONCECACHE.get(sessionid)
    if nonce_tup:
        latest_nonce, reciept_vector = nonce_tup
    else:
        latest_nonce, reciept_vector = (request_nonce,
                                        INITIAL_RECIEPT_VECTOR)

    delta = latest_nonce - request_nonce

    if delta < 0:
        # This a "future" nonce
        latest_nonce = request_nonce
        # Shift our current vector to the left
        reciept_vector <<= -delta
        # And set that this new nonce has been seen
        reciept_vector |= 0x01
    elif delta >= 0 and delta < RECIEPT_VECTOR_BITS:
        # This is a "past" nonce that we have the ability to check
        if reciept_vector & (1 << delta):
            message = "Request nonce has been seen before"
            raise NonceInvalid(message)
        else:
            # Set the bit in the bit vector
            reciept_vector |= 1 << delta
    elif delta > 0 and delta >= RECIEPT_VECTOR_BITS:
        # This is "past" nonce that we don't have the ability to check
        message = "Nonce is too old to validate"
        raise NonceInvalid(message)
```

```
    # Clamp to RECIEPT_VECTOR_BITS because otherwise Python will gladly shift
    # our vector into a bigint
    reciept_vector &= RECIEPT_VECTOR_MASK
    NONCECACHE.set(sessionid, (latest_nonce, reciept_vector), None)


def validate_request(request, request_header):
    hash_mask = request_header['h']
    auth_headers = request_header['ah']
    extra_headers = request_header.get('eah', None)

    sessionid, hmac_key, expiration_time = decrypt_opaque(
        request_header['s'], request_header['iv'], request_header['tag'],
        hash_mask, auth_headers, extra_headers)

    # Session expiration check
    if expiration_time <= int(time.time()):
        raise SessionExpired('Session expired due to absolute expiration time')

    # HMAC validation
    # Performs time-based and nonce-based replay prevention if present

    # Rebuild HMAC input
    using_nonce = using_nonce_replay_prevention(request_header['ah'])
    hmac_input = [request_header['n'], '+'] if using_nonce else ['+']
    hmac_input.append(request_header['t'])
    hmac_input.append(request_header['lt'])
    extra_headers = extra_headers.split(',') if extra_headers else []
    hmac_input += auth_header_values(request, request_header['ah'],
                                     extra_headers)
    hmac_input.append(request.get_full_path())
    hmac_input.append(request.body or '')
    # unicode objects to bytestring for ordinals greater than 128
    hmac_input = [x.decode('latin1').encode('latin1') for x in hmac_input]
    hmac_input = '|'.join(hmac_input)

    # Perform HMAC validation
    our_mac = server_hmac(request_header['h'], hmac_key, hmac_input)
    hmac_valid = hmac.compare_digest(our_mac, request_header['c'])

    if not hmac_valid:
        raise HmacInvalid()

    # If the request is valid, but too much time has elapsed since the prior
    # request, expire the session. Note that it's fine to do this before replay
    # prevetion, because even if an attacker were trying to maliciously replay
    # the request, the authenticated request embeds information that will
    # always expire the session, namely, the request time and the prior request
    # time. Both of these are included in the HMAC.
    inactivity_timeout_seconds = get_setting(
```

```
        'S_ARMOR_INACTIVITY_TIMEOUT_SECONDS', SECONDS_30_MINUTES)
    if (int(request_header['t']) - int(request_header['lt']) >=
            inactivity_timeout_seconds):
        raise SessionExpired('Session expired due to inactivity')

    # Validate that the request has not expired (time-based replay prevention)
    # NB: This is done after HMAC validation
    request_duration_seconds = get_setting(
        'S_ARMOR_REQUEST_VALID_SECONDS', SECONDS_5_MINUTES)
    if time.time() - int(request_header['t']) >= request_duration_seconds:
        raise RequestExpired()

    # Validate that nonce has not been used before (absolute replay prevention)
    if using_nonce:
        validate_nonce(request_header['n'], sessionid)

    return sessionid


def invalidate_session(request_header):
    hash_mask = request_header['h']
    auth_headers = request_header['ah']
    extra_headers = request_header.get('eah', None)
    _, hmac_key, _ = decrypt_opaque(
        request_header['s'], request_header['iv'], request_header['tag'],
        hash_mask, auth_headers, extra_headers)
    mac = server_hmac(request_header['h'], hmac_key, 'Session Expired')
    return tuples_to_header((('i', mac),))


class SessionArmorMiddleware(object):
    '''
    Implementation of the Session Armor protocol.

    Session Armor is an HTTP session authentication protocol hardened against
    request replay and request forgery.
    '''

    def __init__(self):
        self.strict = get_setting('S_ARMOR_STRICT', False)

        auth_headers = get_setting(
                'S_ARMOR_AUTH_HEADERS', DEFAULT_AUTH_HEADERS)
        nonce_replay_prevention = get_setting(
                'S_ARMOR_NONCE_REPLAY_PREVENTION', None)
        self.packed_header_mask = gen_header_mask(
                auth_headers, nonce_replay_prevention)

    def process_request(self, request):
        '''
        Process states of the Session Armor protocol for incoming requests
```

```
'''
header_str = request.META.get('HTTP_X_S_ARMOR', None)

if not self.strict and not header_str:
    return
elif self.strict and not header_str:
    # Disallow requests from clients that do not support SessionArmor

    # If another middleware's process_request raises an Exception
    # before this one, then the following PermissionDenied exception
    # will not be raised. This would be a breach of the authentication
    # system if this pre-empting exception is handled, and a cookie or
    # other authentication credential is used to allow a privileged
    # action. Any of the exception handlers called in the lifecycle of
    # Django's BaseHandler could allow this to happen, including the
    # handle_exception of another middleware.
    #
    # NB: This applies to all PermissionDenied exceptions called from
    # the context of this middleware.
    raise PermissionDenied('Client does not support Session Armor')

request_header = header_to_dict(header_str)
state = get_client_state(request_header)

sa_sessionid = None

if state == CLIENT_READY and request.is_secure():
    try:
        select_hash_mask(request_header['r'])
    except HmacInvalid as e:
        # Client provided an invalid HMAC algo mask
        LOGGER.debug(str(e))
        # Need to raise PermissionDenied here rather than in
        # process_response, otherwise the client will get a 500 rather
        # than a 403.
        raise PermissionDenied(str(e))
elif state == CLIENT_SIGNED_REQUEST:
    try:
        sa_sessionid = validate_request(request, request_header)
    except SessionExpired as e:
        LOGGER.debug(str(e))
        # Return before injeting the session cookie. The request will
        # be processed without a user object. This allows session
        # invalidation to proceed in process_response. We add a header
        # to the request that process_response can pick up to invalidate
        # the session
        request.META['HTTP_X_S_ARMOR_INVALIDATE'] = 'True'
        return
    except OpaqueInvalid as e:
        # Client provided an invalid symmetrically encrypted token
        LOGGER.debug(str(e))
```

```
                raise PermissionDenied(str(e))
            except HmacInvalid as e:
                # Client's HMAC did not validate
                LOGGER.debug(str(e))
                raise PermissionDenied(str(e))
            except RequestExpired as e:
                # Time-based replay prevention
                LOGGER.debug(str(e))
                raise PermissionDenied(str(e))
            except NonceInvalid as e:
                # Counter-based replay prevention
                LOGGER.debug(str(e))
                raise PermissionDenied(str(e))

        if sa_sessionid:
            request.COOKIES[settings.SESSION_COOKIE_NAME] = sa_sessionid


    def process_response(self, request, response):
        '''
        Process states of the Session Armor protocol for outgoing requests
        '''
        header_str = request.META.get('HTTP_X_S_ARMOR', None)

        if not header_str:
            return response

        sessionid = None
        if request.is_secure() and is_modifying_session(response):
            sessionid = extract_session_id(response)

        request_header = header_to_dict(header_str)
        state = get_client_state(request_header)

        if state == CLIENT_READY and request.is_secure() and sessionid:
            try:
                response['X-S-Armor'] = begin_session(
                    request_header, sessionid, self.packed_header_mask)
            except HmacInvalid:
                # If the algo mask was invalid then PermissionDenied was raised
                # in ProcessRequest
                return response
        elif state == CLIENT_SIGNED_REQUEST:
            # Session invalidation
            # Check if the session has expired
            if request.META.get('HTTP_X_S_ARMOR_INVALIDATE', None):
                response['X-S-Armor'] = invalidate_session(request_header)
            # Check if the server is deleting the session, e.g. a logout
            # view has executed.
            elif sessionid == '':
                response['X-S-Armor'] = invalidate_session(request_header)
```

```
        return response
```

A.2 Client Reference Implementation, Chrome Extension Source Code

```
"use strict";
/*
Session Armor Protocol, Google Chrome Extension

Copyright (C) 2016 Andrew Sauber

This software is licensed under the AGPLv3 open source license. See
LICENSE.txt. The license can also be found at the following URL, please note
the above copyright notice. https://www.gnu.org/licenses/agpl-3.0.en.html
*/

var _ = require("underscore");
var Hashes = require("jshashes");
var compare = require("secure-compare");

require("./status-icon");

/* request related */

var hashAlgoMask = "\x01\x05";

var hashModules = [
    [1 << 0, new Hashes.SHA256({'utf8': false})],
//    [1 << 1, Hashes.SHA384],
    [1 << 2, new Hashes.SHA512({'utf8': false})],
    [1 << 3, new Hashes.RMD160({'utf8': false})]
]
hashModules = _.object(hashModules);

var bodyCache = {}

var headerChoices = [
    '', /* least-significant bit used for nonce flag */
    'Host',
    'User-Agent',
    'Accept',
    'Connection',
    'Accept-Encoding',
    'Accept-Language',
    'Referer',
    'Cookie',
```

```
    'Accept-Charset',
    'If-Modified-Since',
    'If-None-Match',
    'Range',
    'Date',
    'Authorization',
    'Cache-Control',
    'Origin',
    'Pragma',
    'DNT',
    'X-Csrf-Token',
    'Sec-WebSocket-Version',
    'Sec-WebSocket-Protocol',
    'Sec-WebSocket-Key',
    'Sec-WebSocket-Extensions',
    'TE',
    'X-Requested-With',
    'X-Forwarded-For',
    'X-Forwarded-Proto',
    'Forwarded',
    'From',
    'HTTP2-Settings',
    'Upgrade',
    'Proxy-Authorization',
    'If',
    'If-Match',
    'If-Range',
    'If-Unmodified-Since',
    'Max-Forwards',
    'Prefer',
    'Via',
    'ALPN',
    'Expect',
    'Alt-Used',
    'CalDAV-Timezones',
    'Schedule-Reply',
    'If-Schedule-Tag-Match',
    'Destination',
    'Lock-Token',
    'Timeout',
    'Ordering-Type',
    'Overwrite',
    'Position',
    'Depth',
    'SLUG',
    'Trailer',
    'MIME-Version'
];

function getHost(url) {
    // capture everything up to the first lone slash
```

```
    // excluding the scheme and port
    return url.match(/(.+:\/\/)([^/:]+)(.*)?\/([^/]|$)/)[2];
}

function getOrigin(url) {
    // capture everything up to the first lone slash
    // including the scheme, host, and port
    return url.match(/(.+:\/\/[^/]+)\/([^/]|$)/)[1];
}

function getPath(url) {
    // capture everything after the first lone slash
    var path = url.match(/(.+:\/\/[^/]+)\/(.*|$)/)[2];
    return '/' + path;
}

function domainHasSession(url) {
    return localStorage[getOrigin(url)] !== undefined;
}

function unpackMask(mask) {
    return stringToBytes(mask.slice(1));
}

function stringToBytes(str) {
    var bytes = [], charCode;
    for (var i = 0, len = str.length; i < len; ++i) {
        charCode = str.charCodeAt(i);
        if ((charCode & 0xFF00) >> 8) {
            bytes.push((charCode & 0xFF00) >> 8);
        }
        bytes.push(charCode & 0xFF);
    }
    return bytes;
}

function bytesToInt(bytes) {
    var n = 0;
    for (var i = bytes.length - 1, len = bytes.length; i >= 0; --i) {
        n |= bytes[i] << (8 * (len - 1 - i));
    }
    return n;
}

function intToBytes(i) {
    // Not using an arbitrary precision implementation of this for two reasons:
    // 1. Bitwise operators in JavaScript convert operands to 32-bit signed
    //    integers, unlike Python, which maintains arbitrary precision.
    // 2. If the input has its MSB as 1, it's treated as negative number, and
    //    gets 1-filled on the right when shifted, resulting in "negative" byte
    //    values which are not amenable to string encoding.
```

```
    // Thus, this 0x00ff mask, which is used to kill the 1-filled bits of
    // parameters which happen to be negative when coerced.
    return [
        (i >> 24 & 0x00ff),
        (i >> 16 & 0x00ff),
        (i >>  8 & 0x00ff),
        (i >>  0 & 0x00ff)
    ];
}

function bytesToString(bytes) {
    return String.fromCharCode.apply(this, bytes);
}

function objToHeaderString(obj) {
    return _.map(_.keys(obj), function (key) {
        return key + ':' + btoa(obj[key]);
    }).join(';');
}

function unpackMasks(headerValues) {
    if (headerValues.h) {
        headerValues.hashMask = unpackMask(headerValues.h)[0];
    }
    if (headerValues.ah) {
        headerValues.headerMask = unpackMask(headerValues.ah);
    }
    return headerValues;
}

function headerStringToObj(str) {
    if (!str) return {};
    var pairs = str.split(';');
    var headerValues = {};
    _.each(pairs, function(pair) {
        pair = pair.split(':');
        headerValues[pair[0]] = atob(pair[1]);
    });
    headerValues = unpackMasks(headerValues);
    return headerValues;
}

function hmac(key, hashMask, string) {
    var macObj = hashModules[hashMask];
    return macObj.b64_hmac(key, string);
}

function headerValuesToAuth(headerMask, extraHeaders, requestHeaders) {
    var selectedHeaders = [];
    for (var i = 0, len = headerMask.length; i < len; ++i) {
        var currentByte = headerMask[len - 1 - i];
```

```
        for (var j = 0; j < 8; ++j) {
            if (currentByte & (1 << j)) {
                selectedHeaders.push(headerChoices[i * 8 + j]);
            }
        }
    }

    // Append the extra authenticated headers in their order
    selectedHeaders = selectedHeaders.concat(extraHeaders);

    // These need to be appended in the bitmask order
    var authHeaderValues = [];
    for (var header of selectedHeaders) {
        for (var reqHeader of requestHeaders) {
            if (header.toLowerCase() === reqHeader.name.toLowerCase()) {
                authHeaderValues.push(reqHeader.value);
            }
        }
    }

    return authHeaderValues;
}

function stringForAuth(nonce, requestTime, lastRequestTime, authHeaderValues,
                       path, body) {
    var macTokens = ['+', requestTime, lastRequestTime];
    if (nonce !== null) {
        macTokens.unshift(nonce);
    }
    macTokens = macTokens.concat(authHeaderValues);
    macTokens = macTokens.concat(path);
    macTokens.push(body || '');
    return macTokens.join('|');
}

function genHeaderString(originValues, ourMac, requestTime, lastRequestTime,
                         nonce) {
    var requestValues = {}
    requestValues.c = ourMac;
    requestValues.t = requestTime;
    requestValues.lt = lastRequestTime;
    requestValues.s = originValues.s;
    requestValues.iv = originValues.iv;
    requestValues.tag = originValues.tag;
    requestValues.h = originValues.h;
    requestValues.ah = originValues.ah;
    if (originValues.eah) {
        requestValues.eah = originValues.eah;
    }
    if (nonce !== null) {
        requestValues.n = nonce;
```

```
    }

    return objToHeaderString(requestValues);
}

function genSignedHeader(details) {
    var originValues = JSON.parse(localStorage[getOrigin(details.url)]);
    var hmacKey = originValues['kh'];

    if (usingNonceReplayPrevention(originValues.ah)) {
        var nonce = getNonce(details.url);
    }
    nonce = nonce ? setAndIncrementNonce(details.url, nonce) : null;

    var requestTime = Math.floor(Date.now() / 1000);
    var lastRequestTime = localStorage[getOrigin(details.url) + '|lrt'];
    var path = getPath(details.url);
    var body = bodyCache[details.requestId];

    /* we use two seperate callbacks, so don't leak memory*/
    delete bodyCache[details.requestId];

    var authHeaderValues = headerValuesToAuth(originValues.headerMask,
                                              originValues.eah.split(','),
                                              details.requestHeaders);
    var authString = stringForAuth(nonce, requestTime, lastRequestTime,
                                   authHeaderValues, path, body);
    var ourMac = hmac(hmacKey, originValues.hashMask, authString);
    ourMac = atob(ourMac);

    return genHeaderString(originValues, ourMac, requestTime, lastRequestTime,
                           nonce);
}

function genReadyHeader() {
    var headerValue = objToHeaderString({
        'r': hashAlgoMask
    });
    return headerValue;
}

function usingNonceReplayPrevention(headerMask) {
    var charCode = headerMask.charCodeAt(headerMask.length - 1);
    return !!(charCode & 0x01);
}

function getNonce(url) {
    var origin = getOrigin(url);
    var nonce = localStorage[origin + '|nonce'];
    return nonce ?
        bytesToInt(stringToBytes(nonce)) :
```

```
        null;
}

function setNonce(url, nonce) {
    var origin = getOrigin(url);
    nonce = bytesToString(intToBytes(nonce));
    localStorage[origin + '|nonce'] = nonce;
    return nonce;
}

function setAndIncrementNonce(url, nonce) {
    nonce++;
    return setNonce(url, nonce);
}

function storeNewSession(url, headerValues) {
    var origin = getOrigin(url);

    if (!origin.startsWith("https")) {
        console.log("Won't store SessionArmor session delivered insecurely.");
        return;
    }

    if (usingNonceReplayPrevention(headerValues.ah)) {
        setNonce(url, bytesToInt(stringToBytes(headerValues['n'])));
    }

    localStorage[origin] = JSON.stringify(headerValues);
}

function invalidateSession(url, serverMac) {
    var origin = getOrigin(url);
    var originValues = JSON.parse(localStorage[origin]);
    var hmacKey = originValues['kh'];
    var ourMac = hmac(hmacKey, originValues.hashMask, "Session Expired");
    serverMac = btoa(serverMac);
    if (!compare(serverMac, ourMac)) return;
    localStorage.removeItem(origin);
    localStorage.removeItem(origin + '|nonce');
    localStorage.removeItem(origin + '|lrt');
}

function onHeaderReceived(details) {
    var headerValues = {};
    _.each(details.responseHeaders, function(header) {
        if (header.name !== "X-S-Armor") return;
        headerValues = headerStringToObj(header.value);
    });

    if (headerValues.hasOwnProperty('s')) {
        storeNewSession(details.url, headerValues);
```

```
    } else if (headerValues.hasOwnProperty('i')) {
        invalidateSession(details.url, headerValues['i']);
    }
}

function beforeSendHeader(details) {
    details.requestHeaders.push({
        "name": "Host",
        "value": getHost(details.url)
    });
    var headerValue =
        domainHasSession(details.url)
            ? genSignedHeader(details)
            : genReadyHeader();
    details.requestHeaders.push({
        "name": "X-S-Armor",
        "value": headerValue
    });

    // Set "last request time" for this domain to now
    var lastRequestTimeKey = getOrigin(details.url) + '|lrt';
    localStorage[lastRequestTimeKey] = Math.floor(Date.now() / 1000);
    return {requestHeaders: details.requestHeaders};
}

/* body-related */

function extendedEncodeURIComponent(s) {
    return encodeURIComponent(s).replace(/[()'!]/g, function(c) {
        return '%' + c.charCodeAt(0).toString(16);
    });
}

function formDataToString(formData) {
    return _.map(Object.keys(formData), function(key) {
        // each key has an array of values
        return _.map(formData[key], function(value) {
            return key + '=' +
                   extendedEncodeURIComponent(value).replace(/%20/g, '+');
        }).join('&');
        // forms are encoded with key=value pairs joined with '&'
        // keys can be repeated
    }).join('&');
}

function beforeRequest(details) {
    /* Skip this step if this request does not have a related, active,
     * SessionArmor session, or does not have body data */
    if (!domainHasSession(details.url) || !details.requestBody) return;

    if (details.requestBody.error) {
```

```
        /* If Chrome has a problem parsing the request body,
         * log and continue. The request will fail on the server side. */
        console.log("request body error: " + details.requestBody.error);
    } else if (details.requestBody.raw) {
        /*
        Raw body authentication requires patching Chromium as follows. This
        forces Chrome to use the "raw" presenter for both the MIME type of
        multipart/form-data _and_ the MIME type of
        application/x-www-form-urlencoded
        (as of 2016-08-24)

        diff --git
          a/extensions/browser/api/web_request/web_request_event_details.cc
          b/extensions/browser/api/web_request/web_request_event_details.cc
        index a9f2f83..835b0eb5 100644
        --- a/extensions/browser/api/web_request/web_request_event_details.cc
        +++ b/extensions/browser/api/web_request/web_request_event_details.cc
        @@ -84,7 +84,6
        @@ void WebRequestEventDetails::SetRequestBody(
               const net::URLRequest* request) {
             if (presenters[i]->Succeeded()) {
               request_body->Set(kKeys[i], presenters[i]->Result());
               some_succeeded = true;
        -      break;
             }
           }
        */
        bodyCache[details.requestId] = String.fromCharCode.apply(null,
                new Uint8Array(details.requestBody.raw[0].bytes));
    }
}

/* handle body data and store it for HMAC */
chrome.webRequest.onBeforeRequest.addListener(
    beforeRequest,
    {"urls": ["https://*/*", "http://*/*"]},
    ["blocking", "requestBody"]
);

/* prepare HMAC before requests */
chrome.webRequest.onBeforeSendHeaders.addListener(
    beforeSendHeader,
    {"urls": ["https://*/*", "http://*/*"]},
    ["blocking", "requestHeaders"]
);

/* handle Session initialization */
chrome.webRequest.onHeadersReceived.addListener(
    onHeaderReceived,
    {"urls": ["https://*/*", "http://*/*"]},
    ["blocking", "responseHeaders"]
```

```
);
```

Appendix B. SessionJack Data

Table B.1: Domain Susceptibility to Session Hijacking

Domain	Bearer Tokens	Packet Sniffing	Cross-Site Scripting	Alexa Rank
drive.google.com	Vulnerable	Protected	Protected	1
mail.google.com	Vulnerable	Protected	Protected	1
facebook.com	Vulnerable	Protected	Protected	2
youtube.com	Vulnerable	Protected	Protected	3
yahoo.com	Vulnerable	Protected	Protected	5
amazon.com	Vulnerable	Vulnerable	Vulnerable	6
console.aws.amazon.com	Vulnerable	Protected	Protected	6
en.wikipedia.org	Vulnerable	Protected	Protected	7
twitter.com	Vulnerable	Protected	Protected	9
linkedin.com	Vulnerable	Protected	Protected	14
ebay.com	Vulnerable	Vulnerable	Vulnerable	17
yandex.ru	Vulnerable	Protected	Protected	19
instagram.com	Vulnerable	Vulnerable	Protected	28
reddit.com	Vulnerable	Vulnerable	Protected	31
wordpress.com	Vulnerable	Vulnerable	Protected	40
paypal.com	Vulnerable	Protected	Protected	42
account.microsoft.com	Vulnerable	Protected	Protected	43
apple.com	Vulnerable	Protected	Protected	49
netflix.com	Vulnerable	Vulnerable	Vulnerable	53
stackoverflow.com	Vulnerable	Vulnerable	Protected	56
alibaba.com	Vulnerable	Vulnerable	Vulnerable	63
github.com	Vulnerable	Protected	Protected	91
dropbox.com	Vulnerable	Protected	Protected	98
chase.com	Vulnerable	Vulnerable	Protected	119
twitch.tv	Vulnerable	Vulnerable	Protected	139
soundcloud.com	Vulnerable	Vulnerable	Vulnerable	166
snapdeal.com	Vulnerable	Vulnerable	Protected	171
steampowered.com	Vulnerable	Vulnerable	Protected	231
sourceforge.net	Vulnerable	Protected	Protected	267
accounts.spotify.com	Vulnerable	Protected	Protected	346
kickstarter.com	Vulnerable	Protected	Protected	368
newegg.com	Vulnerable	Vulnerable	Vulnerable	374
evernote.com	Vulnerable	Protected	Protected	386
oracle.com	Vulnerable	Vulnerable	Vulnerable	425
meetup.com	Vulnerable	Vulnerable	Vulnerable	433
slack.com	Vulnerable	Protected	Protected	540
intuit.com	Vulnerable	Protected	Protected	547
ibm.com	Vulnerable	Vulnerable	Vulnerable	774
atlassian.net	Vulnerable	Protected	Protected	821
macrumors.com	Vulnerable	Vulnerable	Protected	1091
bitbucket.org	Vulnerable	Protected	Protected	1106
mint.com	Vulnerable	Protected	Protected	1198
kongregate.com	Vulnerable	Vulnerable	Vulnerable	1434

bandcamp.com	Vulnerable	Protected	Vulnerable	1444
humblebundle.com	Vulnerable	Protected	Protected	1630
cloud.digitalocean.com	Vulnerable	Protected	Protected	1713
membership.square-enix.com	Vulnerable	Vulnerable	Vulnerable	2004
news.ycombinator.com	Vulnerable	Protected	Protected	2057
tdbank.com	Vulnerable	Vulnerable	Vulnerable	2141
minecraft.net	Vulnerable	Vulnerable	Vulnerable	2449
teespring.com	Vulnerable	Vulnerable	Vulnerable	2478
amtrak.com	Vulnerable	Vulnerable	Vulnerable	3070
dramafever.com	Vulnerable	Vulnerable	Protected	3324
pcpartpicker.com	Vulnerable	Vulnerable	Protected	3743
airdroid.com	Vulnerable	Vulnerable	Vulnerable	4849
linuxquestions.org	Vulnerable	Vulnerable	Protected	4979
express.com	Vulnerable	Vulnerable	Protected	5289
microcenter.com	Vulnerable	Vulnerable	Protected	5749
bitcointalk.org	Vulnerable	Vulnerable	Vulnerable	6040
coinbase.com	Vulnerable	Protected	Protected	6167
archlinux.org	Vulnerable	Protected	Protected	6572
gumroad.com	Vulnerable	Protected	Protected	7417
hackerrank.com	Vulnerable	Protected	Protected	8264
zennioptical.com	Vulnerable	Vulnerable	Protected	8643
xmarks.com	Vulnerable	Protected	Protected	8900
hitbox.tv	Vulnerable	Vulnerable	Vulnerable	9090
unrealengine.com	Vulnerable	Protected	Protected	9718
oculus.com	Vulnerable	Protected	Protected	10462
wallhaven.cc	Vulnerable	Vulnerable	Protected	12278
codechef.com	Vulnerable	Vulnerable	Protected	13112
caremark.com	Vulnerable	Protected	Vulnerable	13202
venmo.com	Vulnerable	Protected	Protected	14225
expensify.com	Vulnerable	Protected	Protected	14249
freesound.org	Vulnerable	Vulnerable	Protected	14353
login.aessuccess.org	Vulnerable	Protected	Protected	14877
wellsfargodealerservices.com	Vulnerable	Protected	Protected	15948
loseit.com	Vulnerable	Vulnerable	Protected	16215
bluejeans.com	Vulnerable	Protected	Vulnerable	18845
coastal.com	Vulnerable	Vulnerable	Protected	21508
topcoder.com	Vulnerable	Vulnerable	Vulnerable	21551
codeforces.com	Vulnerable	Vulnerable	Protected	22019
thebodyshop-usa.com	Vulnerable	Vulnerable	Protected	24844
bayan.ir	Vulnerable	Vulnerable	Protected	27057
kanbanflow.com	Vulnerable	Vulnerable	Protected	30466
teavana.com	Vulnerable	Vulnerable	Protected	33993
projecteuler.net	Vulnerable	Protected	Protected	35078
bighugelabs.com	Vulnerable	Vulnerable	Vulnerable	38333
afraid.org	Vulnerable	Vulnerable	Vulnerable	39307
myminifactory.com	Vulnerable	Vulnerable	Vulnerable	39358
dcollege.net	Vulnerable	Protected	Protected	42970
kraken.com	Vulnerable	Protected	Protected	43088
bitcoin.cz	Vulnerable	Vulnerable	Protected	48778
typeracer.com	Vulnerable	Vulnerable	Vulnerable	50197
tdcardservices.com	Vulnerable	Protected	Protected	56458

audiotool.com	Vulnerable	Vulnerable	Vulnerable	72253
hashicorp.com	Vulnerable	Protected	Protected	72667
fourmilab.ch	Vulnerable	Vulnerable	Vulnerable	75080
ocremix.org	Vulnerable	Vulnerable	Protected	82602
unrealtournament.com	Vulnerable	Protected	Protected	105627
quakelive.com	Vulnerable	Vulnerable	Vulnerable	150405
qhimm.com	Vulnerable	Vulnerable	Vulnerable	218679
lexaloffle.com	Vulnerable	Vulnerable	Vulnerable	226464
contactlensking.com	Vulnerable	Vulnerable	Protected	236765
usaco.org	Vulnerable	Vulnerable	Vulnerable	256864
zergid.com	Vulnerable	Vulnerable	Vulnerable	285250
wvshare.com	Vulnerable	Vulnerable	Protected	294061
catzilla.com	Vulnerable	Vulnerable	Protected	387793
amarriner.com	Vulnerable	Vulnerable	Vulnerable	3489038

Appendix C. Formal Verification

C.1 Proverif Model

```
(* Copyright 2016 - present Andrew Sauber. All rights reserved.  *)
(* Formal verification of the Session Armor protocol using pi-calculus *)


(****************************************************************************)

(* Q: If the server completes the protocol, and the the client began the
    protocol exactly once, what does that prove? *)
(* A: It proves that an attacker acting as a server could not have fulfilled
    the request from the client and then acted as a man-in-the-middle to
    impersonate the client and replay the request with the legitimate server.
    In other words, a legitimate request can only be completed once. *)

(* Q: What do we want to prove? *)
(* A: We want to prove that if the client makes a request using an hmac with a
certain key, then only the client that originally received the key during the
setup phase of the SA the protocol could have made this request. We also want to
prove that an attacker cannot access the server_secret or any client_secret *)

(* Q: Can we prove that the request was made within a specified time period? *)
(* A: ProVerif does not have a notion of time, so we cannot prove that the
request was made within a specified time period. However, there was a paper
published by one of the ProVerif authors indicating that a nonce can be used to
simulate a timestamp. So we include a nonce here with the correspondence
queries. *)

(****************************************************************************)

(* Types *)
type key.
type nonce.
type timestamp.

(* Channels *)
free HTTPS: channel [private].
free HTTP:  channel.

(* Free Variables *)
free server_secret_test, session_secret_test: bitstring
[private].

(* Functions *)
```

```
fun encrypt(bitstring, key): bitstring.
reduc forall plaintext: bitstring, k: key;
   decrypt(encrypt(plaintext, k), k) = plaintext.
fun hmac(bitstring, key): bitstring.

(* Correspondence events *)
event clientRequest(key, timestamp, nonce, bitstring, bitstring).
event serverResponse(key, timestamp, nonce, bitstring, bitstring).

(* Correspondence Query *)
query session_secret: key,
     request_time:   timestamp,
     request_nonce:  nonce,
     request_url:    bitstring,
     request_data:   bitstring;
inj-event(serverResponse(session_secret,
                         request_time,
                         request_nonce,
                         request_url,
                         request_data)) ==>
inj-event(clientRequest(session_secret,
                        request_time,
                        request_nonce,
                        request_url,
                        request_data)).

(* Secrecy queries *)
query attacker(server_secret_test).
query attacker(session_secret_test).

(* Client Macros *)
let browser() =
   new user_id:        bitstring;
   new password:       bitstring;

   out(HTTPS, (user_id, password));
   in(HTTPS, (session_id:      bitstring,
              session_secret:  key,
              session_expiry:  timestamp,
              session_token:   bitstring));

   new request_time:   timestamp;
   new request_nonce:  nonce;
   new request_url:    bitstring;
   new request_data:   bitstring;

   event clientRequest(session_secret,
                       request_time,
                       request_nonce,
                       request_url,
                       request_data);
```

```
   out(HTTP, (request_time,
              request_nonce,
              hmac((request_time, request_nonce, request_url, request_data),
                  session_secret),
              session_token,
              request_url,
              request_data)).


(* Server Macros *)
let webapp() =
   in(HTTPS, (user_id: bitstring, password: bitstring));

   new server_secret:  key;
   new session_secret: key;
   new session_id:     bitstring;
   new session_expiry: timestamp;

   out(HTTP, encrypt(server_secret_test, server_secret));
   out(HTTP, encrypt(session_secret_test, session_secret));

   let session_token = encrypt((session_id,
                                user_id,
                                session_secret,
                                session_expiry), server_secret) in
   out(HTTPS, (session_id, session_secret, session_expiry, session_token));

   in(HTTP, (request_time:  timestamp,
             request_nonce: nonce,
             request_hmac:  bitstring,
             client_token:  bitstring,
             request_url:   bitstring,
             request_data:  bitstring));

   let (=session_id, =user_id, =session_secret, =session_expiry) =
       decrypt(client_token, server_secret) in
   let (server_hmac: bitstring) =
       hmac((request_time, request_nonce, request_url, request_data),
           session_secret) in
   if request_hmac = server_hmac then
       event serverResponse(session_secret,
                            request_time,
                            request_nonce,
                            request_url,
                            request_data).

(* Main *)
process
( !(browser()) | !(webapp()) )
```

C.2 Proverif Verification Output

```
Process:
{1}new server_secret: key;
{2}new user_id: bitstring;
{3}new password: bitstring;
(
    {4}!
    {5}out(HTTPS, (user_id,password));
    {6}in(HTTPS, (session_id: bitstring,session_secret: key,
        session_expiry: timestamp,session_token: bitstring));
    {7}new request_time: timestamp;
    {8}new request_nonce: nonce;
    {9}new request_url: bitstring;
    {10}new request_data: bitstring;
    {11}event clientRequest(session_secret,request_time,request_nonce,
        request_url,request_data);
    {12}out(HTTP, (request_time,request_nonce,hmac((request_time,
        request_nonce,request_url,request_data),session_secret),
        session_token,request_url,request_data))
) | (
    {13}!
    {14}in(HTTPS, (user_id_31: bitstring,password_32: bitstring));
    {15}new session_secret_33: key;
    {16}new session_id_34: bitstring;
    {17}new session_expiry_35: timestamp;
    {18}out(HTTP, encrypt(server_secret_test,server_secret));
    {19}out(HTTP, encrypt(session_secret_test,session_secret_33));
    {20}let session_token_36: bitstring = encrypt((session_id_34,user_id_31,
            session_secret_33,session_expiry_35),server_secret) in
    {21}out(HTTPS, (session_id_34,session_secret_33,session_expiry_35,
            session_token_36));
    {22}in(HTTP, (request_time_37: timestamp,request_nonce_38: nonce,
            request_hmac: bitstring,client_token: bitstring,
            request_url_39: bitstring,request_data_40: bitstring));
    {23}let (=session_id_34, =user_id_31, =session_secret_33,
=session_expiry_35) =
            decrypt(client_token,server_secret) in
    {24}let server_hmac: bitstring = hmac((request_time_37,request_nonce_38,
            request_url_39,request_data_40),session_secret_33) in
    {25}if (request_hmac = server_hmac) then
    {26}event serverResponse(session_secret_33,request_time_37,
            request_nonce_38,request_url_39,request_data_40)
)

-- Query not attacker(session_secret_test[])
Completing...
Starting query not attacker(session_secret_test[])
RESULT not attacker(session_secret_test[]) is true.
```

```
-- Query not attacker(server_secret_test[])
Completing...
Starting query not attacker(server_secret_test[])
RESULT not attacker(server_secret_test[]) is true.


-- Query
inj-event(serverResponse(session_secret_1504,request_time_1505,request_nonce_150
6,
    request_url_1507,request_data_1508))
==>
inj-event(clientRequest(session_secret_1504,request_time_1505,request_nonce_1506
,
    request_url_1507,request_data_1508))
Completing...  Starting query
inj-event(serverResponse(session_secret_1504,request_time_1505,request_nonce_150
6,
    request_url_1507,request_data_1508))
==>
inj-event(clientRequest(session_secret_1504,request_time_1505,request_nonce_1506
,
    request_url_1507,request_data_1508))
goal reachable: begin(clientRequest(session_secret_33[password_32 =
password[],user_id_31 = user_id[],!1 = endsid_2426],request_time[session_token
= encrypt((session_id_34[password_32 = password[],user_id_31 = user_id[],!1 =
endsid_2426],user_id[],session_secret_33[password_32 = password[],user_id_31 =
user_id[],!1 = endsid_2426],session_expiry_35[password_32 =
password[],user_id_31 = user_id[],!1 =
endsid_2426]),server_secret[]),session_expiry = session_expiry_35[password_32 =
password[],user_id_31 = user_id[],!1 = endsid_2426],session_secret =
session_secret_33[password_32 = password[],user_id_31 = user_id[],!1 =
endsid_2426],session_id = session_id_34[password_32 = password[],user_id_31 =
user_id[],!1 = endsid_2426],!1 = @sid_2427],request_nonce[session_token =
encrypt((session_id_34[password_32 = password[],user_id_31 = user_id[],!1 =
endsid_2426],user_id[],session_secret_33[password_32 = password[],user_id_31 =
user_id[],!1 = endsid_2426],session_expiry_35[password_32 =
password[],user_id_31 = user_id[],!1 =
endsid_2426]),server_secret[]),session_expiry = session_expiry_35[password_32 =
password[],user_id_31 = user_id[],!1 = endsid_2426],session_secret =
session_secret_33[password_32 = password[],user_id_31 = user_id[],!1 =
endsid_2426],session_id = session_id_34[password_32 = password[],user_id_31 =
user_id[],!1 = endsid_2426],!1 = @sid_2427],request_url[session_token =
encrypt((session_id_34[password_32 = password[],user_id_31 = user_id[],!1 =
endsid_2426],user_id[],session_secret_33[password_32 = password[],user_id_31 =
user_id[],!1 = endsid_2426],session_expiry_35[password_32 =
password[],user_id_31 = user_id[],!1 =
endsid_2426]),server_secret[]),session_expiry = session_expiry_35[password_32 =
password[],user_id_31 = user_id[],!1 = endsid_2426],session_secret =
session_secret_33[password_32 = password[],user_id_31 = user_id[],!1 =
```

```
endsid_2426],session_id = session_id_34[password_32 = password[],user_id_31 =
user_id[],!1 = endsid_2426],!1 = @sid_2427],request_data[session_token =
encrypt((session_id_34[password_32 = password[],user_id_31 = user_id[],!1 =
endsid_2426],user_id[],session_secret_33[password_32 = password[],user_id_31 =
user_id[],!1 = endsid_2426],session_expiry_35[password_32 =
password[],user_id_31 = user_id[],!1 =
endsid_2426]),server_secret[]),session_expiry = session_expiry_35[password_32 =
password[],user_id_31 = user_id[],!1 = endsid_2426],session_secret =
session_secret_33[password_32 = password[],user_id_31 = user_id[],!1 =
endsid_2426],session_id = session_id_34[password_32 = password[],user_id_31 =
user_id[],!1 = endsid_2426],!1 = @sid_2427]), session_token =
encrypt((session_id_34[password_32 = password[],user_id_31 = user_id[],!1 =
endsid_2426],user_id[],session_secret_33[password_32 = password[],user_id_31 =
user_id[],!1 = endsid_2426],session_expiry_35[password_32 =
password[],user_id_31 = user_id[],!1 = endsid_2426]),server_secret[]),
session_expiry = session_expiry_35[password_32 = password[],user_id_31 =
user_id[],!1 = endsid_2426], session_secret = session_secret_33[password_32 =
password[],user_id_31 = user_id[],!1 = endsid_2426], session_id =
session_id_34[password_32 = password[],user_id_31 = user_id[],!1 =
endsid_2426], @sid = @sid_2427, @occ11 = @occ_cst) ->
end(endsid_2426,serverResponse(session_secret_33[password_32 =
password[],user_id_31 = user_id[],!1 = endsid_2426],request_time[session_token
= encrypt((session_id_34[password_32 = password[],user_id_31 = user_id[],!1 =
endsid_2426],user_id[],session_secret_33[password_32 = password[],user_id_31 =
user_id[],!1 = endsid_2426],session_expiry_35[password_32 =
password[],user_id_31 = user_id[],!1 =
endsid_2426]),server_secret[]),session_expiry = session_expiry_35[password_32 =
password[],user_id_31 = user_id[],!1 = endsid_2426],session_secret =
session_secret_33[password_32 = password[],user_id_31 = user_id[],!1 =
endsid_2426],session_id = session_id_34[password_32 = password[],user_id_31 =
user_id[],!1 = endsid_2426],!1 = @sid_2427],request_nonce[session_token =
encrypt((session_id_34[password_32 = password[],user_id_31 = user_id[],!1 =
endsid_2426],user_id[],session_secret_33[password_32 = password[],user_id_31 =
user_id[],!1 = endsid_2426],session_expiry_35[password_32 =
password[],user_id_31 = user_id[],!1 =
endsid_2426]),server_secret[]),session_expiry = session_expiry_35[password_32 =
password[],user_id_31 = user_id[],!1 = endsid_2426],session_secret =
session_secret_33[password_32 = password[],user_id_31 = user_id[],!1 =
endsid_2426],session_id = session_id_34[password_32 = password[],user_id_31 =
user_id[],!1 = endsid_2426],!1 = @sid_2427],request_url[session_token =
encrypt((session_id_34[password_32 = password[],user_id_31 = user_id[],!1 =
endsid_2426],user_id[],session_secret_33[password_32 = password[],user_id_31 =
user_id[],!1 = endsid_2426],session_expiry_35[password_32 =
password[],user_id_31 = user_id[],!1 =
endsid_2426]),server_secret[]),session_expiry = session_expiry_35[password_32 =
password[],user_id_31 = user_id[],!1 = endsid_2426],session_secret =
session_secret_33[password_32 = password[],user_id_31 = user_id[],!1 =
endsid_2426],session_id = session_id_34[password_32 = password[],user_id_31 =
user_id[],!1 = endsid_2426],!1 = @sid_2427],request_data[session_token =
encrypt((session_id_34[password_32 = password[],user_id_31 = user_id[],!1 =
endsid_2426],user_id[],session_secret_33[password_32 = password[],user_id_31 =
```

```
user_id[],!1 = endsid_2426],session_expiry_35[password_32 =
password[],user_id_31 = user_id[],!1 =
endsid_2426]),server_secret[]),session_expiry = session_expiry_35[password_32 =
password[],user_id_31 = user_id[],!1 = endsid_2426],session_secret =
session_secret_33[password_32 = password[],user_id_31 = user_id[],!1 =
endsid_2426],session_id = session_id_34[password_32 = password[],user_id_31 =
user_id[],!1 = endsid_2426],!1 = @sid_2427]))

RESULT inj-event(serverResponse(session_secret_1504,request_time_1505,
    request_nonce_1506,request_url_1507,request_data_1508))
==> inj-event(clientRequest(session_secret_1504,request_time_1505,
    request_nonce_1506,request_url_1507,request_data_1508)) is true.
```

Bibliography

[1] Ben Adida. Sessionlock: securing web sessions against eavesdropping. In *Proceedings of the 17th international conference on World Wide Web*, pages 517–524. ACM, 2008.

[2] Ben Adida. Sessionlock: securing web sessions against eavesdropping. In *Proceedings of the 17th international conference on World Wide Web*, pages 517–524. ACM, 2008.

[3] Nadhem J Al Fardan and Kenneth G Paterson. Lucky thirteen: Breaking the tls and dtls record protocols. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 526–540. IEEE, 2013.

[4] Andrew W Appel. Verified software toolchain. In *ESOP*, volume 11, pages 1–17. Springer, 2011.

[5] Andrew W Appel. Verification of a cryptographic primitive: Sha-256. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 37(2):7, 2015.

[6] Adam Barth. HTTP State Management Mechanism. RFC 6265, April 2011.

[7] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying hash functions for message authentication. pages 1–15. Springer-Verlag, 1996.

[8] Tim Berners-Lee. The original http as defined in 1991. `https://www.w3.org/Protocols/HTTP/AsImplemented.html`.

[9] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Keccak sponge function family main document. *Submission to NIST (Round 2)*, 3:30, 2009.

[10] Karthikeyan Bhargavan, Cédric Fournet, Ricardo Corin, and Eugen Zălinescu. Verified cryptographic implementations for tls. *ACM Transactions on Information and System Security (TISSEC)*, 15(1):3, 2012.

[11] Bruno Blanchet. Modeling and verifying security protocols with the applied pi calculus and ProVerif. *Foundations and Trends in Privacy and Security*, 1(1–2):1–135, October 2016.

[12] Taehwan Choi and Mohamed G Gouda. Httpi: An http with integrity. In *Computer Communications and Networks (ICCCN), 2011 Proceedings of 20th International Conference on*, pages 1–6. IEEE, 2011.

[13] Tyler Close. Web-key: Mashing with permission. In *Proceedings of Web*, volume 2, 2008.

[14] Italo Dacosta, Saurabh Chakradeo, Mustaque Ahamad, and Patrick Traynor. One-time cookies: Preventing session hijacking attacks with stateless authentication tokens. *ACM Transactions on Internet Technology (TOIT)*, 12(1):1, 2012.

[15] Philippe De Ryck, Lieven Desmet, Frank Piessens, and Wouter Joosen. Secsess: keeping your session tucked away in your browser. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, pages 2171–2176. ACM, 2015.

[16] Danny Dolev and Andrew Yao. On the security of public key protocols. *IEEE Transactions on information theory*, 29(2):198–208, 1983.

[17] Nancy Durgin, Patrick Lincoln, John Mitchell, and Andre Scedrov. Multiset rewriting and the complexity of bounded security protocols. *Journal of Computer Security*, 12(2):247–311, 2004.

[18] Roy T. Fielding and Julian Reschke. Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing. RFC 7230, June 2014.

[19] Mark A. Finlayson. Mit java wordnet interface (jwi) userfhs guide.

[20] Django Software Foundation. Django session extention. `https://github.com/django/django/blob/9baf692a58de78dba13aa582098781675367c329/django/contrib/sessions/middleware.py#L48`.

[21] Kevin Fu, Emil Sit, Kendra Smith, and Nick Feamster. The dos and don'ts of client authentication on the web. In *USENIX Security Symposium*, pages 251–268, 2001.

[22] Matthew Green. How to choose an authenticated encryption mode. `https://blog.cryptographyengineering.com/2012/05/19/how-to-choose-authenticated-encryption/`.

[23] Jeremiah Grossman. Xst sorta lives! (bypassing httponly). `http://blog.jeremiahgrossman.com/2007/04/xst-lives-bypassing-httponly.html`.

[24] Phillip Hallam-Baker, Professor John Franks, Lawrence C. Stewart, Eric W. Sink, Jeffery L. Hostetler, Paul J. Leach, and Ari Luotonen. An Extension to HTTP : Digest Access Authentication. RFC 2069, January 1997.

[25] Stefan Heule, Devon Rifkin, Alejandro Russo, and Deian Stefan. The most dangerous code in the browser. In *Workshop on Hot Topics in Operating Systems, (HotOS)*. USENIX, May 2015.

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

[27] Trent Jaeger. Hughes diffie-hellman variant. 2008.

[28] Antoine Joux. Authentication failures in nist version of gcm. *NIST Comment*, page 3, 2006.

[29] John Kelsey. Sha-3 derived functions: cshake, kmac, tuplehash, and parallelhash. *NIST Special Publication*, 800:185, 2016.

[30] Mitja Kolšek. Session fixation vulnerability in web-based applications. *Acros Security*, page 7, 2002.

[31] Guillaume Lehembre. Wi-fi security – wep, wpa y wpa2. *Recuperado el*, 9(10), 2006.

[32] Alex X Liu, Jason M Kovacs, C-T Huang, and Mohamed G Gouda. A secure cookie protocol. In *Computer Communications and Networks, 2005. ICCCN 2005. Proceedings. 14th International Conference on*, pages 333–338. IEEE, 2005.

[33] Gavin Lowe. An attack on the needham-schroeder public-key authentication protocol. *Information processing letters*, 56(3):131–133, 1995.

[34] David McGrew and John Viega. The galois/counter mode of operation (gcm). *Submission to NIST Modes of Operation Process*, 20, 2004.

[35] Stephen J Minutillo. Cookieless http session persistence using the base tag.

[36] Bodo Möller, Thai Duong, and Krzysztof Kotowicz. This poodle bites: exploiting the ssl 3.0 fallback. *PDF online*, pages 1–4, 2014.

[37] Bodo Möller, Thai Duong, and Krzysztof Kotowicz. This poodle bites: exploiting the ssl 3.0 fallback. *PDF online*, pages 1–4, 2014.

[38] Magnus Nystrom. Identifiers and Test Vectors for HMAC-SHA-224, HMAC-SHA-256, HMAC-SHA-384, and HMAC-SHA-512. RFC 4231, December 2005.

[39] Kenneth G Paterson and Arnold Yau. Padding oracle attacks on the iso cbc mode encryption standard. In *CT-RSA*, volume 2964, pages 305–323. Springer, 2004.

[40] Andreas Petersson and Martin Nilsson. Forwarded HTTP Extension. RFC 7239, June 2014.

[41] Lisa Phifer. Wpa psk crackers: Loose lips sink ships. `http://www.wi-fiplanet.com/tutorials/article.php/3667586/WPA-PSK-Crackers-Loose-Lips-Sink-Ships.htm`.

[42] Joseph Salowey, Abhijit Choudhury, and David McGrew. Aes galois counter mode (gcm) cipher suites for tls. Technical report, 2008.

[43] Aditya Sood and Richard Enbody. The state of http declarative security in online banking websites. *Computer Fraud & Security*, 2011(7):11–16, 2011.

[44] Kevin Spett. Cross-site scripting. *SPI Labs*, 1:1–20, 2005.

[45] Marc Stevens, Pierre Karpman, and Thomas Peyrin. Freestart collision for full sha-1. Cryptology ePrint Archive, Report 2015/967, 2015. `http://eprint.iacr.org/2015/967`.

[46] Gary Wassermann and Zhendong Su. Static detection of cross-site scripting vulnerabilities. In *Proceedings of the 30th international conference on Software engineering*, pages 171–180. ACM, 2008.