NTNU

Norwegian University of
Science and Technology

Secure Termination in Real-World Security Protocols

Jonas Lunde Toreskås

Master of Science in Communication Technology
Submission date: June 2016
Supervisor: Colin Alexander Boyd, ITEM
Co-supervisor: Britta Hale, ITEM

Norwegian University of Science and Technology
Department of Telematics

Title: Secure Termination in Real-World Security Protocols

Student: Jonas Toreskås

Problem description:

Real-world security protocols such as Transport Layer Security (TLS) are very complex and can be vulnerable to many different kinds of attacks. One aspect that has not been widely explored is attacks aimed at deceiving a recipient regarding whether or not a protocol session has properly terminated. This type of deception can be achieved by truncating connections between a user and a server. In 2013 Smyth and Pironti showed that TLS can be vulnerable to such attacks which can lead to serious consequences [35].

Firstly, this master thesis will aim to reproduce the attacks of Smyth and Pironti. In 2014 they reported that some of their attacks have not yet been patched. Because of this, the next task of this project is to check whether the applications tested are still vulnerable. In the case of patched solutions, an assessment of the patches will be included in this project. Then, the project will explore the possibility of adapting the attacks and applying them or similar truncation attacks to other real-world security protocols.

Responsible professor: Colin A. Boyd, ITEM

Supervisor: Britta Hale, ITEM

Abstract

As the Internet was initially invented without any security concerns, a way of secure communication over an untrusted network was nowhere to be found. After years of research, the TLS protocol became this Internet standard for secure end-to-end communication. Today, version 1.2 of TLS is the standard for web security, and the protocol provides authentication and ensures confidentiality and integrity.

However, as TLSv1.2 is the most common form of implementing web application security, new attacks are being discovered continuously in the attempt of breaking the protocol. One of these attacks is the truncation attack discovered by Smyth and Pironti in 2013 [35]. This attack was focused around truncating TLS connections between a user and a web application server. By exploiting application logic flaws found in a selection of web applications, Smyth and Pironti were able to cast votes on behalf of honest voters in an online voting system, take full control of Hotmail accounts, and gain temporary control of Google accounts.

Now, three years later, these attacks have been recreated in this report. By reviewing the sign-out procedures for these applications and reproducing the attacks, it appeared that the application logic flaw still exists in the online voting system, but the truncation attack is only possible when a user is using certain setups. Particularly, it appears that only certain web browsers allow this sort of attack.

Due to poor handling of TLS termination modes, many modern web browsers are still susceptible to truncation attacks, and it remains up to the individual web developer to thwart these types of attacks by avoiding application logic flaws that can be exploited.

Sammendrag

Da Internett først ble oppfunnet var ikke sikkerhet en bekymring. En måte å sikre kommunikasjon over et nettverk var verken oppfunnet eller påtenkt. Etter år med forskning ble TLS-protokollen en Internett-standard for å sikre ende-til-ende kommunikasjon. I dag er versjon 1.2 av TLS protokollen standarden for web sikkerhet, og protokollen tilbyr autentisering samtidig som den sikrer konfidensialitet og integritet.

Siden TLSv1.2 er den mest vanlige formen for implementering av applikasjonssikkerhet over Internett, blir nye angrep stadig oppdaget i et forsøk på å knekke protokollen. Et av disse angrepene er et avkuttingsangrep som ble oppdaget av Smyth og Pironti i 2013 [35]. Dette angrepet fokuserte på å kutte av TLS-tilkoblinger mellom en bruker og en web-applikasjonsserver. Ved å utnytte logiske feil som finnes i et utvalg av web-applikasjoner, var Smyth og Pironti i stand til å avgi stemmer på vegne av andre velgere i et online stemmegivning system, ta full kontroll over Hotmail-kontoer, og få midlertidig kontroll over Google-kontoer.

Nå, tre år senere, er disse angrepene blitt gjenskapt i denne rapporten. Ved å analysere avloggingsprosedyrene for hver av disse applikasjonene, og ved å forsøke å gjenskape angrepene, viste det seg at logiske feil fortsatt eksisterer i det elektroniske valgsystemet, men angrepet er imidlertidig bare mulig når en bruker bruker visse oppsett. Faktisk ser det ut til at bare enkelte nettlesere tillater denne typen angrep.

På grunn av dårlig håndtering av TLS' terminerings moduser, er mange moderne nettlesere fortsatt sårbare for avkuttingsangrep, og det forblir opp til den enkelte webutvikleren å hindre denne type angrep ved å unngå anvendelse av logiske feil som kan utnyttes.

Preface

This report presents the work done during a master thesis (TTM4905, spring 2016) given by the Department of Telematics (ITEM) at the Faculty of Information Technology, Mathematics, and Electrical Engineering (IME). All of the research done during the course of this thesis was done at the Norwegian University of Science and Technology (NTNU). When choosing a subject for my master thesis, it was done by looking through a series of suggestions provided by the IME faculty. This subject was chosen because of my interest in network security and my willingness to do some practical work in terms of attempting to recreate attacks.

Writing this master thesis has been both fun and challenging, and I've learned a lot about previously unknown features of TLS and how the protocol provides security for web applications. My five years at NTNU have been finalized by this master thesis, and the past semester has been a great way of doing so.

Jonas Toreskås

Trondheim, 10th June 2016

Acknowledgement

Before introducing the reader to the contents of this report, I would like to thank my responsible professor Colin Boyd and my supervisor Britta Hale for suggesting this project as a master thesis. I would also like to thank them for proofreading this report and for providing useful feedback throughout the course of this work.

Furthermore, I would also like to extend my thanks to my sister Oda for proofreading this report. Finally, I wish to thank my fellow students whom I have shared an office with during the course of this research. Without them, showing up for work would have been much more of a challenge.

Jonas Toreskås

Trondheim, 10th June 2016

Contents

List of Figures

List of Tables

Listings

List of Acronyms

AES Advanced Encryption Standard.

ARPANET Advanced Research Projects Agency Network.

BEAST Browser Exploit Against SSL/TLS.

CBC Cipher Block Chaining.

CPU Central Processing Unit.

DHE Diffie-Hellman Key Exchange.

DSA Digital Signature Algorithm.

DTLS Datagram Transport Layer Security.

FTP File Transfer Protocol.

HTML Hypertext Markup Language.

HTTP Hypertext Transfer Protocol.

HTTPS Hypertext Transfer Protocol Secure.

IE Internet Explorer.

IETF Internet Engineering Task Force.

IME Faculty of Information Technology, Mathematics, and Electrical Engineering.

IP Internet Protocol.

IPsec Internet Protocol Security.

ISO International Organization for Standardization.

ITEM Department of Telematics.

LLC Logical Link Control.

LTS Long Term Support.

MAC Message Authentication Code.

MitM Man In The Middle.

NAT Network Address Translation.

NTNU Norwegian University of Science and Technology.

OSI Open Systems Interconnection.

PGP Pretty Good Privacy.

PRF Pseudorandom Function.

RC4 Rivest Cipher 4.

RFC Request for Comments.

SHA Secure Hash Algorithm.

SSH Secure Shell.

SSL Secure Sockets Layer.

TCP Transmission Control Protocol.

TLS Transport Layer Security.

UDP User Datagram Protocol.

URI Uniform Resource Identifier.

URL Uniform Resource Locator.

VM Virtual Machine.

WWW World Wide Web.

Chapter 1
Introduction

Computer security revolves around the protection of computing systems used all around the world. Protection, in this case, can include protection against intruders like hackers or other adversaries, but the essence of computer security is described by three key objectives: *Confidentiality*, *Integrity*, and *Availability* [36]. Together, these three objectives form a triad, as illustrated in Figure 1.1, and their descriptions are defined as follows.

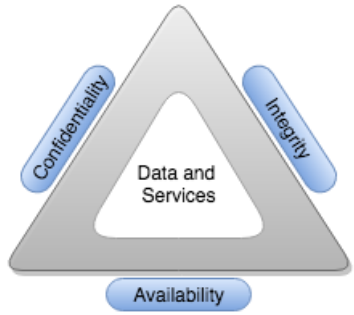

Figure 1.1: The Security Requirements Triad

- **Confidentiality:** By providing data confidentiality it is assured that information intended as confidential or private is not disclosed to users or other individuals that are not authorized to access this information. Also assured is the control an individual has over own privacy information and to whom this information may be available. Failure to provide these features will lead to a loss of confidentiality.

- **Integrity:** Data integrity ensures that the altering of information will only occur in an authorized manner. By also assuring that a system's function performs its intended task without any manipulation of the system that is

unauthorized, system integrity is assured. Any loss of or altering of information that is unauthorized will be classified as a loss of integrity.

- **Availability:** An authorized user or individual should not be denied the use of a system or service. Disruption or loss of access will indicate a loss of availability.

The Internet is a public communication network, and when web applications are executed over it, they mainly rely on Hypertext Transfer Protocol Secure (HTTPS) to be secure. To ensure *confidentiality* and *integrity* of a web application, HTTPS uses regular Hypertext Transfer Protocol (HTTP) over the TLS protocol. When a client and a server are to communicate within a web application, it is the task of HTTPS to protect this communication. Application logic does, however, also play a major part when determining the security of an application. While the TLS protocol provides secure communication, the protocol does not offer any protection against flaws in application logic. Also, TLS allows several client-server connections to be authenticated by the same session, but the protocol only provides data integrity for one of these connections at a time. This means that when a web application uses multiple TLS connections to, for instance, load content in parallel, the TLS protocol does not guarantee the ordering of the messages sent in these connections. In the case of particularly sensitive operations (like when dealing with authentication credentials), logic in web applications must be extra protective when using parallel connections. This report describes how truncation attacks can be used against such flaws to violate *integrity*.

1.1 Motivation

The TLS protocol is very complex and has been extensively researched in terms of discovering attacks against it. However, attacks aimed at compromising the session termination of the TLS protocol has not been explored extensively. Particularly, these attacks aim towards violating the beliefs a user has of a successful session termination by truncating connections between a server and a user. In 2013 such attacks were described by Smyth and Pironti [35], and this report aims to further explore the effectiveness of these truncation attacks by first finding out if they still are valid.

1.2 Problem and Scope

The scope of this report is to firstly provide background information about TLS and truncation attacks. The report will then provide a practical investigation of the effects of truncation attacks on TLS. Due to logic flaws, certain web applications are

reported to be vulnerable to truncation attacks, and this report seeks to validate these reports by recreating previously successful attacks against the latest versions of these web applications.

The report explores the effect of truncation attacks against the same type of web applications on different web browsers, rather that the initially intended goal of exploring the possibility of adapting the truncation attacks against TLS to other security protocols like Datagram Transport Layer Security (DTLS) and Secure Shell (SSH). This adjustment is made in the light of results that suggested that different web browsers have different handling of the TLS protocol.

1.3 Methodology

When first starting working on this master thesis it was important to understand how the TLS works and how the protocol is vulnerable to certain attacks. To obtain this understanding, the TLS protocol and papers discussing different attacks were analyzed in depth.

Next, the truncation attacks discovered by Smyth and Pironti in 2013 [35] were analyzed and recreated in order to discover whether or not these attacks are still valid. The results from the recreation of these attacks were then analyzed.

From the analysis conducted, the results indicated that different web browsers act differently when communication is handled by the TLS protocol. The aforementioned attacks were then recreated several times on different browsers in order to uncover if they are vulnerable to truncation attacks or not.

Finally, in an attempt to describe a way of avoiding truncation attacks from a web developer's standpoint, a web application was created with the goal of pointing out how a simple application can be changed from vulnerable to secure with just a couple of adjustments.

1.4 Outline

The chapters in this report are organized as follows:

- **Chapter 2** is an overview of both the history of the Internet and the TLS protocol. The chapter also provides the reader with theoretical background information needed to understand the features of TLS and the preliminaries of the attacks described in this report.
- **Chapter 3** is a detailed description of the most important features of TLS.

- **Chapter 4** introduces the different attacks relevant to this report. Among the attacks mentioned, special focus has been given to the different variants of truncation attacks against TLS.
- **Chapter 5** contains a detailed practical investigation of the truncation attacks discovered by Smyth and Pironti in 2013 [35]. Furthermore, this chapter describes the attempts of recreating these attacks.
- **Chapter 6** investigates further the results of the previous chapter by discovering to which degree modern web browsers are secure against truncation attacks. This chapter also includes a description of a generic method to avoid truncation attacks in a web application.
- **Chapter 7** concludes the report and discusses the results.

Chapter 2
Background

This chapter gives an overview of both the history of the Internet and the TLS protocol. The chapter also introduces the reader to the theoretical background information needed to understand the features of TLS and the preliminaries of the attacks described in this report.

2.1 History

The Internet started out as a research project called *Advanced Research Projects Agency Network (ARPANET)* which was initiated in the 1960s during the Cold War [24]. As this time period was a prime time for research it was originally intended as a means of sharing information between the United States Department of Defense, the United States military and universities. The 1970s introduced services such as electronic mail, and a commercialisation of ARPANET followed in the 1980s. The World Wide Web (WWW) as we know it was later introduced in the 1990s. At this time, the Internet was implemented with little to no security.

After the commercialisation of ARPANET and the introduction of the World Wide Web, Netscape Communication started working on employing secure communication over the existing network. In 1994 the first security protocol was created and given the name Secure Sockets Layer (SSL). This protocol, despite never being officially released, made a lot of headway. Version 2 of SSL was released officially a year later in 1995. As this version was proven to have many weaknesses [12], as later described in Section 4.2, a third version was released in 1996 with the goal of fixing weaknesses.

While the SSL protocol increased in popularity, other companies were attempting to create their own security solutions in order to compete with Netscape. The task of defining a standard protocol was given to the Internet Engineering Task Force (IETF) [1]. The goal of IETF is to be a open community dedicated to developing standards towards their goal of making the Internet work. Request for Comments (RFC) documents are the official channel used by IETF to publish Internet standards.

The first Internet security standard published by IETF was TLSv1.0 (RFC 2246 [15]). As TLSv1.0 was heavily based on SSLv3 it was ready for release already in 1999 as shown in the timeline in Figure 2.1.

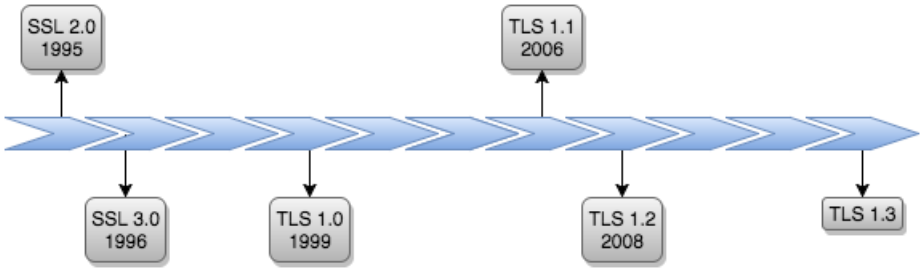

Figure 2.1: History of network security protocols

RFC 4346 [17] was released seven years later and defined as version 1.1 of TLS as it was based on TLSv1.0. The difference between these two versions is mainly the successor's support for new cryptographic algorithms and countermeasures for previously discovered attacks. In 2008, RFC 5246 [16] defined TLSv1.2 and it is this version that is currently employed in most web applications. As TLSv1.2 is the version focused on in this report, it is explained in detail in Chapter 3. Also included in the timeline in Figure 2.1 is TLSv1.3 which is a pending draft [27].

2.2 Cryptography

Cryptography are the techniques studied and implemented with the goal of ensuring secrecy and authenticity of information. A basic understanding of cryptography is important in order to understand how TLS and network security works. The area of cryptographic algorithms is large and it is commonly defined as three main areas of study: *symmetric encryption*, *asymmetric encryption* and *cryptographic hash functions* [36]. For the sake of network security relevant for this report, the next sections will introduce symmetric and asymmetric cryptography.

2.2.1 Asymmetric Cryptography

Asymmetric Cryptography, or *Public Key Cryptography*, describes encryption systems where secret keys are not shared between entities. Public key cryptography operates with key pairs consisting of one private key and one public key. In this key pair, the main feature is that when one key is used to encrypt a message into a *ciphertext*, the other key is the only key able to decrypt the message into *plaintext*.

Consider the illustration in Figure 2.2 where Alice wishes to send a secret message to Bob over a network link that is not secure. In this case, Bob has sent his public

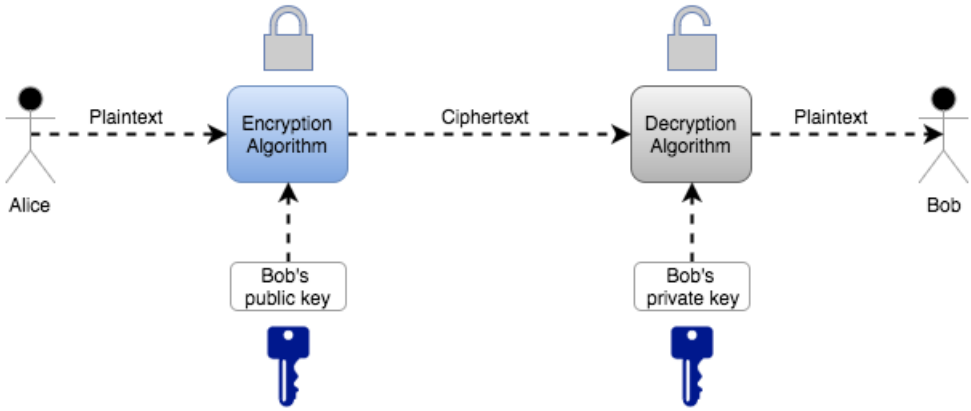

Figure 2.2: Encryption with Public Key cryptography

key to Alice, and with it, Alice encrypts the desired message. The secret message can now be transmitted to Bob as a ciphertext, and Bob will be able to decrypt it with his private key. Several different cryptosystems use asymmetric key techniques as a source of security, among these are the RSA encryption algorithm [29], the Diffie-Hellman Key Exchange (DHE) protocol [18] and the Digital Signature Algorithm (DSA) [20]. Asymmetric key algorithms are also commonly used in a selection of other protocols like Pretty Good Privacy (PGP) [21], SSH [37], and TLS [16].

2.2.2 Symmetric Cryptography

Symmetric key cryptography describes a selection of encryption methods that uses the same cryptographic key for both encryption and decryption. Compared to the public key cryptography example illustrated in Figure 2.2, Figure 2.3 shows that the key used in an asymmetric encryption algorithm is defined as a shared secret between Alice and Bob.

The main disadvantage of asymmetric cryptography is that when Alice knows the shared secret and encrypts her message with it, Bob *must* know the same shared secret in order to decrypt the ciphertext. Encryption by the use of the symmetric key techniques typically use either a stream cipher like Rivest Cipher 4 (RC4) [30], or a block cipher like the Advanced Encryption Standard (AES) [7]. Symmetric ciphers can also be used to guarantee that a message has not been tampered with during encryption. By using a Message Authentication Code (MAC) [11] generated by symmetric cipher like Cipher Block Chaining (CBC)-MAC, message integrity can be ensured.

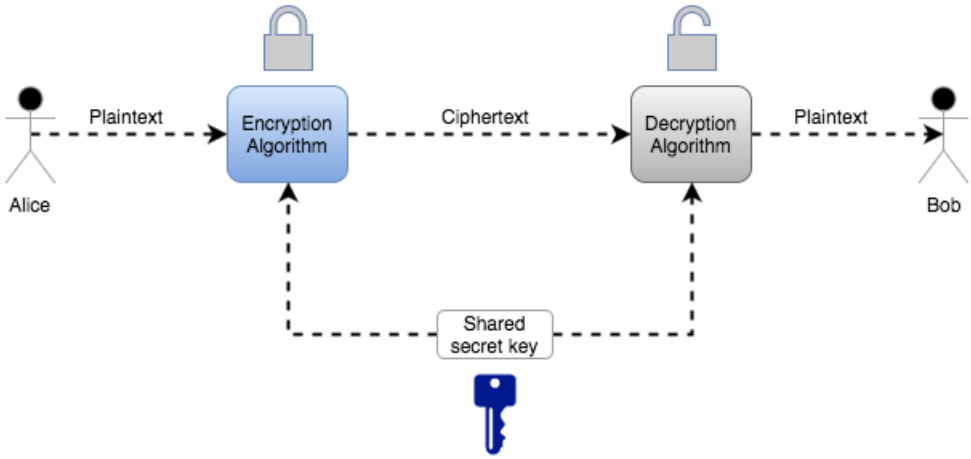

Figure 2.3: Encryption with Asymmetric cryptography

2.3 OSI Reference Model

The *Open Systems Interconnection (OSI) Reference Model* is a model created by the International Organization for Standardization (ISO) to apply a standardized architecture for communication in a computing system [38, 26]. This stacked architecture model has seven different layers: *Application Layer*, *Presentation Layer*, *Session Layer*, *Transport Layer*, *Network Layer*, *Data Link Layer*, and the *Physical Layer*. By providing a standardized layered architecture, as described by Figure 2.4, the goal of the OSI model was to achieve interoperability of different communication systems using standard protocols.

The lowest layer in the OSI model is the physical layer. This layer's task is to focus on transmitting and receiving a raw bit stream over a physical medium like cables and optical fiber. Layer number two in this model is the data link layer which task is to transfer data frames between two nodes on the physical medium. The data link layer also provides error-free transfers by the use of the Logical Link Control (LLC) protocol and the Media Access Control (MAC) sublayer. Layer number three in the OSI model is the network layer which controls the routing and switching operations of the subnet by the use of the Internet Protocol (IP). Layer number four is the transport layer which makes sure that messages are delivered in the correct sequence and without errors. A selection of transmission protocols used by the transport layer exist, but most commonly used are the Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) protocols. The session layer is layer number five in the OSI model and its purpose is to provide services to establish sessions between entities in the presentation layer. The presentation layer is like a

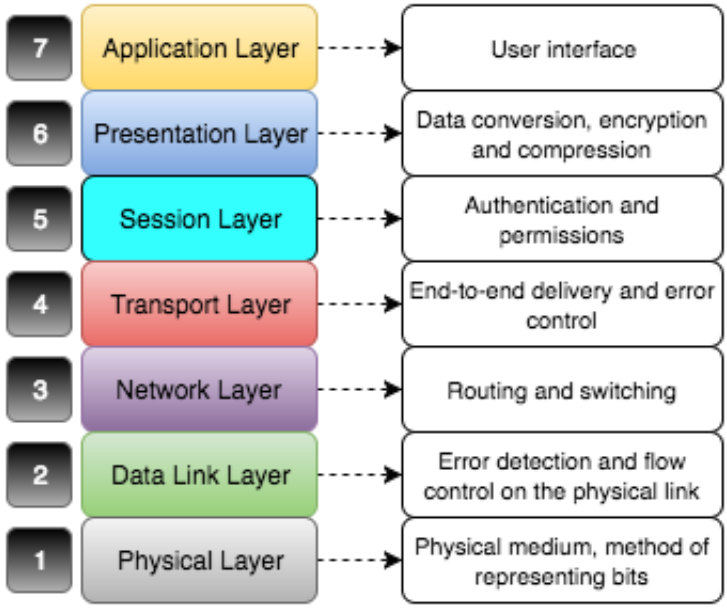

Figure 2.4: The seven layers of the OSI architecture

translator for the network as data is formatted so that it can be represented in the final layer. Layer number seven is the application layer which is also the final layer. In this layer all processed information is displayed to users. This display is typically achieved by the use of the HTTP or the File Transfer Protocol (FTP) protocols.

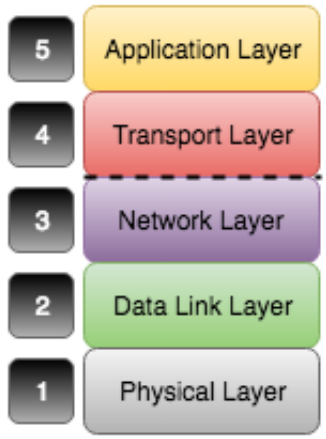

Figure 2.5: Five-layer Internet protocol stack

Derived from the OSI model is the *Five-layer Internet protocol stack* shown in Figure 2.5 [23]. A layer in these architectures serves the layer above it and is being served by the layer below it. As shown in the two figures, the Internet protocol stack is a simplified version of the OSI model, and Figure 3.1 in the next chapter shows how the TLS protocol fits between the application and the transport layer. Another security solution may be applied in the network layer in the form of the Internet Protocol Security (IPsec) protocol suite [22].

2.4 HTTP

HTTP is a generic, stateless application protocol first introduced in 1996 as HTTP/1.0 and later released as HTTP/1.1 in 1999 as RFC 2616 [19]. HTTP is implemented in a client program and a server program and these two programs communicate with each other by exchanging HTTP messages. Over the World Wide Web, HTTP is the main form of data communication and the protocol defines the structure of the messages communicated. A HTTP message can consist of a request from a client to a server or a response from a server to a client:

```
HTTP-message = Request | Response      ;HTTP/1.1 messages
```

2.4.1 HTTP Messages

As previously implied, HTTP is structured as a protocol based on requests and responses. In Figure 2.6 it is illustrated that a client may send a HTTP request to a server and the server will reply with a HTTP response.

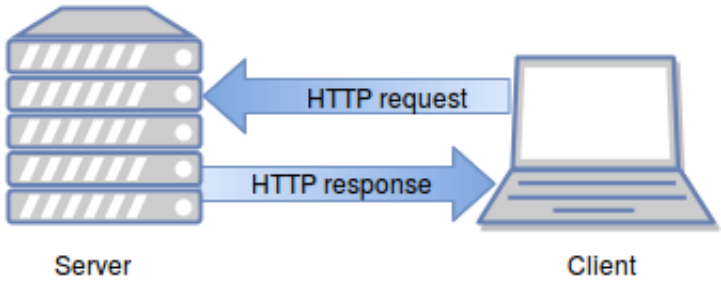

Figure 2.6: Behaviour of HTTP requests and responses

HTTP Request A HTTP request message is a message from a client to a server that indicates the action to be made on the resource identified by the protocol. In this case, a resource is typically a file or an executable implemented on the server (like a sign-out method). Defined in the specification for HTTP/1.1 are eight methods each

of which identifies the action to be made. These methods, as well as their description are listed below.

- `OPTIONS`: For a given Uniform Resource Locator (URL), the `OPTIONS` method returns the HTTP methods that the server supports. Not typically called by requesting a common resource, but rather a "*".
- `GET`: When a Request-Uniform Resource Identifier (URI) is identified, the `GET` method returns the entity information from it. This entity can typically be a simple web page.
- `HEAD`: The `HEAD` method is almost identical to the `GET` method above. The difference is, that when the server sends a response, the message-body should not be returned, only the message-headers.
- `POST`: The `POST` method requests that the server accepts the entity enclosed in the request message. This method is typically used when submitting a web form (like registering a user to a web site or casting a vote).
- `PUT`: An enclosed entity is requested to be stored under the supplied URI by the use of the `PUT` method.
- `DELETE`: A specified resource can be deleted by the use of the `DELETE` method.
- `TRACE`: When calling the `TRACE` method, a client will be able to see what the other end of the request is seeing. To achieve this, the method invokes a remote loop-back of the request message.
- `CONNECT`: HTTP uses the `CONNECT` method exclusively for proxies with the ability to switch to being a tunnel dynamically (i.e. SSL Tunneling).

HTTP Response A HTTP response message is the response a server sends after receiving and interpreting a HTTP request from a client. A server's response will contain a status code with an associated text indicating the meaning of that particular status code. All the different status codes supported by HTTP can be found in Appendix A. This report deals mostly with the `200 - OK` status code indicating that an action has been successfully received and the `302 - Found` status code indicating that a redirect has to happen before completing the request.

2.4.2 HTTP Secure

As observed in the address bar in web browsers, a secure connection is indicated with the "https" protocol identifier. HTTPS is, essentially, the HTTP protocol described above when used over TLS [28]. The secure HTTPS traffic will be distinguished from

the insecure HTTP traffic by the use of a different port. From this, using HTTP over TLS should be as simple as using HTTP over TCP.

2.5 Network Analysis

Throughout the course of this type of research, a lot of network traffic need to be recorded and analyzed. This report refers to basic network analysis techniques when explaining how certain types of information have been obtained. The following sections will provide a brief description of these techniques.

2.5.1 Wireshark

Wireshark is a tool used to analyze network protocols by capturing packets over a network and displaying the packet data in detail [6]. The tool is free, open source, and is available on most platforms and what differentiates Wireshark from other network packet analyzers is its graphical front-end. The uses for Wireshark are many as it can be used for troubleshooting network problems, debugging protocol implementations, learning details of network protocols, and examining security problems.

Wireshark is able to capture all packets traversing a network, but it is the tool's ability to *Filter* packets on defined criteria that makes it very useful when information needed in this report is obtained. For the purposes of examining packets encrypted with the TLS protocol later in this report, a `ssl` filter can be applied to a Wireshark capture and the tool will obey by only displaying packets using the SSL/TLS protocol. In addition to this protocol filter, the `ip.src` filter can be used to only display outgoing traffic. By using these two filters, TLS traffic sent from the Virtual Machine (VM) in Section 5.2 could be analyzed with ease by reviewing the displayed packets that pass through these filters.

2.5.2 iptables

iptables is a Linux administration tool used for filtering network packets and Network Address Translation (NAT) [2]. It is controlled as a command line program and allows system administrators to configure tables provided by the firewall in the Linux kernel. By using iptables, a system administrator is allowed to define tables and populate them with chains of rules describing how to handle network packets. A rule in iptables specifies what to do with a packet that matches this particular rule by jumping to a "target". iptables allows four different targets: `ACCEPT`, `REJECT`, `LOG` and `DROP`.

In this report iptables is mainly used as a way of dropping certain packets (like sign-out requests) sent and received over the network. As an example of how iptables is used to do this, the command below can be observed.

```
iptables -A OUTPUT -m length --length <<size of packets in bytes>>
  -j DROP
```

The iptables program will interpret the command above in the following way. The `-A` flag will inform iptables that the incoming rule is to be appended to the current rule chain. `OUTPUT` indicates that the new rule only applies to outgoing traffic. Next is the `-m` packet matching module. This module is followed by a module name and in this case it indicates that the rule should match depending on the `length` of the packets captured. `-length` followed by a chosen number indicating the packet size further specifies more precisely which packets that will match the new rule. Finally, the `-j` flag specifies which *target* iptables should jump to if the new rule receives a match. In cases where certain packets are to be blocked from the network the `DROP` target is used, and with it, iptables will ignore the packet and stop processing the rules in the current chain.

2.6 Truncation

This report explores, in depth, the variation of truncation attacks against TLS that can be attempted in order to exploit certain logic flaws in web applications and web browsers. To provide the reader with a more general understanding of how these attacks work, this section aims to give a brief introduction to the meaning of *truncation* in the field of computer science.

By definition, truncation involves limiting the number of digits on the right side of a decimal point. However, when talking about truncation of a message or a HTTP request, truncation means specifically leaving out the last part of the message or request. As a typical example of how truncation is often done automatically in computers and applications, consider a field in a registration form that only allows 128 characters. If a user types in a message larger than the 128 allocated characters, an application will typically ignore the last part of this message and take the remaining 128 characters as valid input. This report will look into various attacks where the principle of truncation can be used to drop parts of messages or dropping entire messages altogether when an adversary has full control over a network.

Chapter 3
Transport Layer Security

TLS is a protocol that provides secure communication over the Internet. Secure communication in this case means that the protocol allows a client and a server to communicate without the interference from a third party as it is designed to prevent message tampering or forgery as well as eavesdropping. This chapter, in its entirety, is based on the protocol documentation for TLSv1.2 as specified by RFC 5246 [16].

While providing privacy and data integrity between two communicating applications remains the primary goal of the TLS protocol, RFC 5246 describes TLS as a protocol with four goals. These protocol goals are described below as defined in RFC 5246 [16, Section 2]:

1. Cryptographic security: Two parties will be able to establish a secure connection by using TLS.
2. Interoperability: Without knowing each others code, two independent programmers should still be able to implement applications with TLS and still successfully negotiate cryptographic parameters.
3. Extensibility: By providing a framework that accepts new public keys and encryption methods, TLS accomplishes two new sub-goals: the need for a new protocol is prevented and the need for a new security library is avoided.
4. Relative efficiency: The TLS protocol includes an optional session caching scheme with the goal of reducing the number of connections established from scratch. This scheme is included because public key operations and other cryptographic operations tend to be highly Central Processing Unit (CPU) intensive. In addition to this, TLS is implemented to avoid unnecessary network activity.

To achieve these goals, the TLS protocol is composed of two layers: the *Record Protocol* and the *Handshake Protocol*. Figure 3.1 describes the manner in which the

TLS protocol's layers operates between the application layer and transport layer as they are described by the OSI model in Section 2.3.

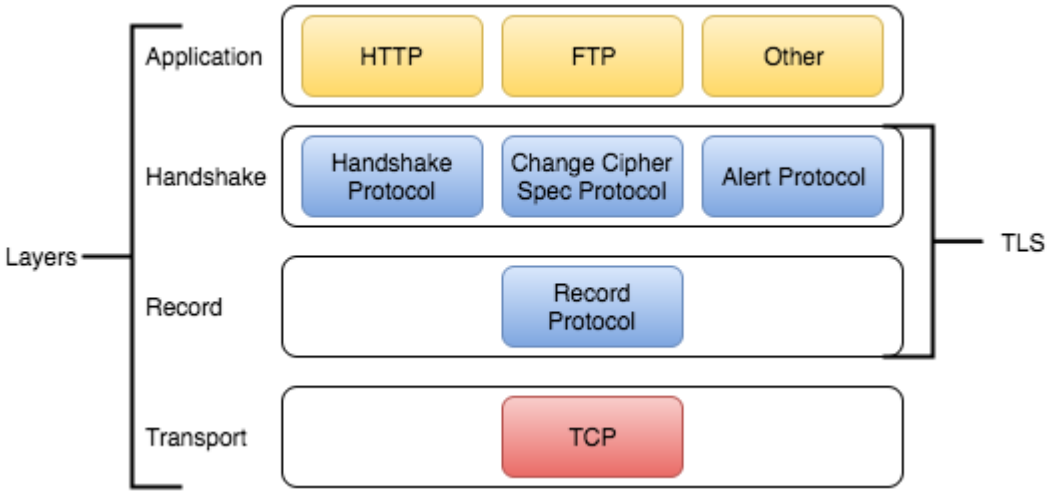

Figure 3.1: Layering of the TLS subprotocols in the OSI model

The TLS protocol comes with one very particular advantage in that it doesn't require the application protocol running on top of it to be of any particular kind. As an example, both the HTTP and the FTP protocols may lay on top of the TLS protocol without any problems. Security can be added to each of these protocols by using the TLS standard. This standard does not, however, specify how said security should be added. It is the job of a protocol designer to make the correct choices when implementing security.

3.1 TLS Record Protocol

Figure 3.1 show the TLS Record Protocol on top of the transport protocol TCP. The TLS Record Protocol applies two basic properties to achieve TLS' goal of connection security:

- *Private connection:* Even though the Record protocol can be used without encryption, data is in most cases encrypted by using symmetric cryptography (See Section 2.2.2).
- *Reliable connection:* By using a keyed MAC, a message integrity check is included in the message transport. This MAC is computed by some secure hash function like one of the Secure Hash Algorithm (SHA) variations.

From an application layer protocol, like the HTTP, data is received by the Record Protocol and transmitted to the transport layer (TCP). Before data is transmitted to TCP, the Record Protocol applies its basic properties to ensure connection encryption and reliability.

3.1.1 Connection States

TLS operates with *Connection States*. Such a state describes the operating environment of the TLS Record Protocol. The connection states specify a compression algorithm, an encryption algorithm and a MAC algorithm. In theory, there exist four different connection states: current read and write states, and pending read and write states. When a record is processed (as described in the section 3.1.2), it is done under the two current states. The pending states are set by the TLS Handshake Protocols described in section 3.2.

3.1.2 TLS Record Layer Process

The first step of the Record Layers process is to fragment blocks of information received into *TLSPlaintext* records of 2^{14} bytes or less in length. All of these records are then compressed by using the compression algorithm specified in the current session state (see section 3.1.1). If a compression algorithm is not defined, it is defined as `CompressionMethod.null` by default as one compression algorithm must always be active. The compression of a TLSPlaintext structure into a *TLSCommpressed* structure may not increase the total content length by more than 1024 bytes. If a larger fragment of TLSCompressed than 2^{14} bytes if found during decrompression, a *fatal decrompression error* is reported by the Record layer.

The next step in the Record Layer process is translating the TLSCompressed structure into *TLSCiphertext*. This is done by using encryption and MAC functions. A record's MAC counteracts the possibly missing, extra or repeated messages by including a sequence number, and the whole encryption process can be reversed by a decryption function.

The record is now almost ready to be transmitted to an underlying protocol. Finally, this process appends a Record Protocol header to the TLSCiphertext record. Included in this header are the protocol version, content type and the total length of the TLSCiphertext. These fields describe which SSL/TLS version employed, the protocol delivering data to the Record Protocol and the size of the data fragment, accordingly. The understanding of the process described in this section is simplified by Figure 3.2.

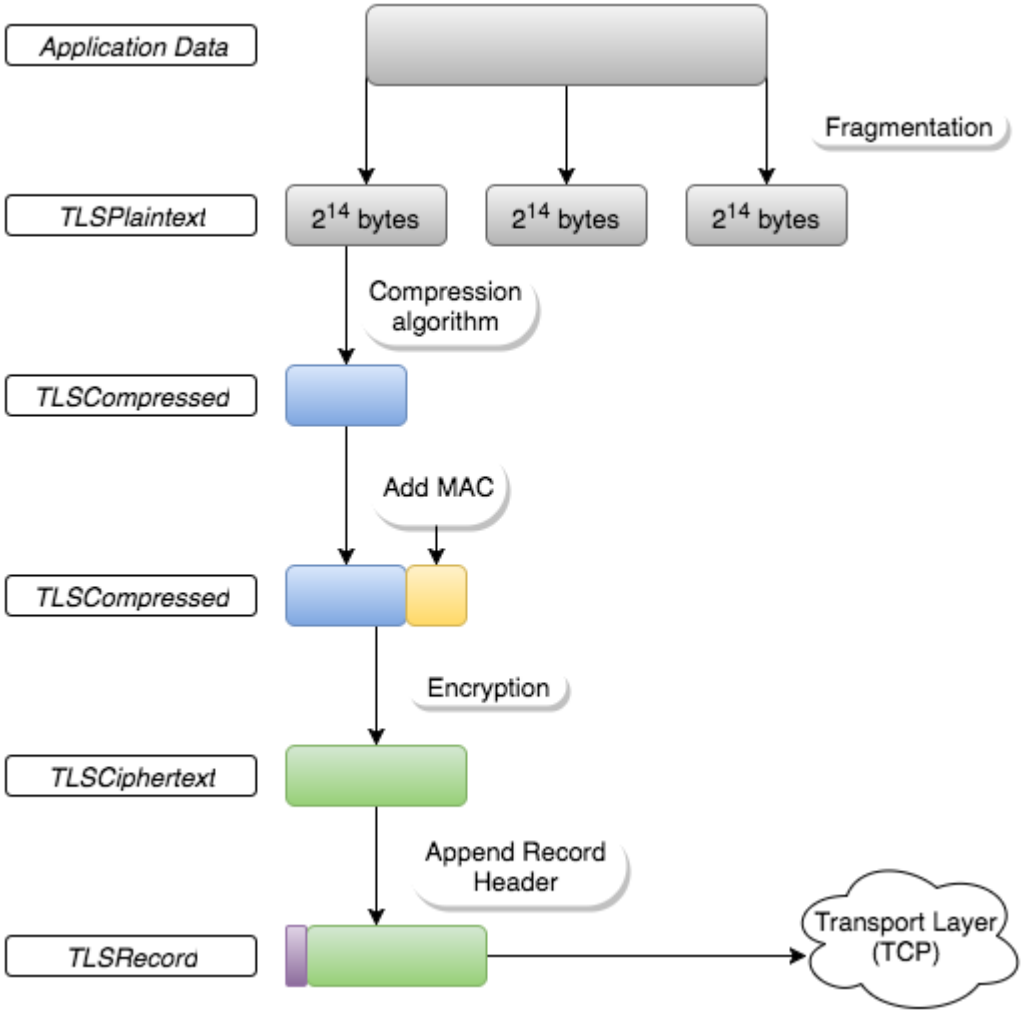

Figure 3.2: TLS Record Layer process

3.2 TLS Handshaking Protocols

The TLS Handshaking Protocols describes three subprotocols (management protocols) used by TLS to allow the agreement of security details between peers. Among other things, these subprotocols allow peers to authenticate themselves and report errors to each other. When negotiated, these security details forms the security parameters used by the TLS Record Layer when protecting application data. A list of these session items and their description can be found in Appendix B.1. The following

sections describe these three subprotocols.

3.2.1 Change Cipher Spec Protocol

The first subprotocol is the *Change Cipher Spec Protocol*. This protocol includes a single byte message of value "1" that exists with the purpose of signaling changes in ciphering strategies.

```
struct {
enum { change_cipher_spec(1), (255) } type;
} ChangeSipherSec;
```

If a change in such strategies is imminent, the receiving party is notified by the aforementioned message, for both the server and the client. When the message is received, the receiver instructs the record layer to replace the read current state with the read pending state. In the same way the sender copies the write pending state into the write current state (See Section 3.1.1).

3.2.2 Alert Protocol

The second subprotocol is the Alert Protocol that concerns alert type messages supported by the TLS Record Layer. These messages consist of a description and a severity assessment of the occurring alert. An alert can either be of *warning* or of *fatal* severity. In the case of a fatal level alert message, an immediate termination of the TLS connection will occur. Other TLS connections residing under the same session may still continue in this case under the assumption that the session identifier is invalidated to prevent the creation of new connections under the failed session. Appendix B.2 lists all possible error messages and codes recognized by the TLS record layer [16, §7.2.1]. These messages are, like all other messages in TLS, encrypted and compressed according to the rules set by the current connection state.

Closure Alerts

To avoid the truncation attacks described in Section 4.2 in the next chapter, the client and the server sharing a TLS connection have to both acknowledge the closing of a connection. This acknowledgement may be initiated by either the client or the server, and is done by the use of the close_notify message found in Appendix B.2.

A `close_notify` message notifies either a client or a server that no other messages will be sent by the party initiating the `close_notify` message on their shared TLS connection. The receiver of a `close_notify` message must immediately respond with their own `close_notify` message forcing the connection between the two parties to

close at once. Any pending writes on the connection is ignored, and beyond this point all data received over the connection is ignored.

3.2.3 Handshake Protocol

Perhaps the most complex part of the TLS protocol is the Handshake Protocol. Its responsibility is to allow authentication of a server and a client, as well as negotiating an encryption algorithm and cryptographic keys before application data is exchanged.

The TLS Handshake Protocol applies three basic properties to the TLS goal of connection security:

- The identity of each peer can be authenticated by using asymmetric cryptography (See Section 2.2.1).
- *Secure negotiation of a shared secret:* when a shared secret for a connection is negotiated between two parties it cannot be obtained even by an attacker attempting eavesdropping by placing himself in the middle of an authenticated connection.
- *Reliable negotiation:* an attacker trying to modify the negotiation communication between two parties will be detected.

The session states described in Section 3.1.1 and their cryptographic parameters are produced by the TLS Handshake Protocol. A TLS Handshake process is initiated when a client and a server start their communication. This process is described below.

Establishing a Secure Session by Using TLS

When a Handshake process is initiated between two parties, there are several steps involved. All of these steps form a message flow between the two parties including only the handshake messages shown in Appendix B.3 [16, §7.4]. This message flow is described in Figure 3.3 where "*" denotes optional messages. Also included in the message flow in Figure 3.3 are *Change Cipher Spec* protocol messages, as they play an important role in the TLS handshake.

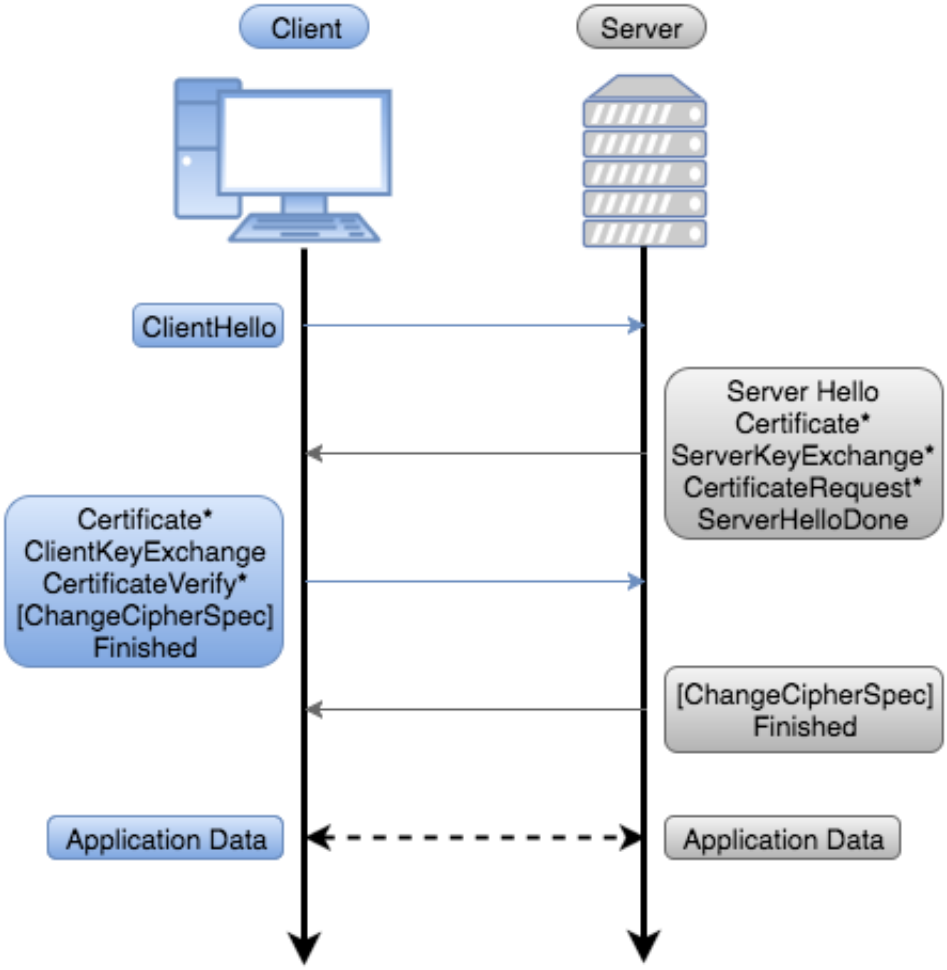

Figure 3.3: Message flow for a full TLS handshake

The following steps explain the TLS Handshake further:

1. The first necessary step of the handshake is that the client is required to send a *ClientHello* message. This message includes a random value associated with the client (*ClientHello.random*), as well as a list of the client's supported compression algorithms.

2. The response given by the server must be a *ServerHello* message. If this it not the case, a *fatal* error will occur. This message is sent on the premise that a set of algorithms are agreed upon, and it includes the server's random value (*ServerHello.random*).

3. At this point the server may send additional messages to the client. The first of these messages is the server's *Certificate* message. This message is sent if the server is to be authenticated, and it includes the server's certificate. This certificate must, unless explicitly specified otherwise, be of type X.509v3 [14]. Another applying rule to the server's certificate is that it must be compatible with the agreed upon key exchange algorithm.

4. If a server sends a *Certificate* message to the client, a *ServerKeyExchange* message may also immediately be sent. Such a message may only be sent if the previous message contains insufficient data to allow the client to exchange a premaster secret. This case is only caused by certain key exchange methods.

5. When a server has been authenticated by its certificate, it may send a *CertificateRequest* message prompting the client to provide his certificate. The *CertificateRequest* message is sent immediately after either the *Certificate* or *ServerKeyExchange* message, and depends on the selected cipher suite.

6. After the optional messages have been sent, or not sent, by the server, it will send a *ServerHelloDone* message. This message indicates an end to the phase of the handshake that includes hello associated messages. From the server's point of view, a waiting period follows as the client's response is imminent.

7. The first part of the client's response depends on whether or not optional messages has been sent by the server. If a *CertificateRequest* message has been received by the client, he must first respond with a *Certificate* message. A client's certificate is subject to the same rules as that of a server.

8. Contrary to the optional *ServerKeyExchange* message which may be sent by the server, the client must always send a *ClientKeyExchange* message. If no client certificate is requested by the server, this is the first message a client must send upon receiving a *ServerHelloDone* message. A public key algorithm was agreed upon during the "hello" phase of the handshake, and it is based on this algorithm the *ClientKeyExchange* message gets its content. The content of this message sets the premaster secret, which is derived from the client's own public key parameters. This premaster secret may also be encrypted with the public key provided by the server's certificate.

9. When a client certificate is sent to the server during this handshake, it may be a certificate with signing ability. If this is the case, a *CertificateVerify* message must immediately follow the *ClientKeyExchange* message. This message provides explicit verification of the client sending the certificate.

10. The client's pending session state is now copied into its current session state by the use of a *ChangeCipherSpec* message, and the client sends a final *Finished*

message. The purpose of the *Finished* message is to provide verification of a successful negotiation of the new algorithms, keys and secrets.

11. In the same way as the client before, the server now responds with *ChangeCipherSpec* message and copies its pending session state into its current session state. Immediately after, the server sends a *Finished* message with same purpose as the client's *Finished* message.

3.2.4 Cryptographic Computations

Only one more step remains in order for a protected connection to be fully established. To establish this connection, the TLS Record Protocol requires the client and server's random values, specification of a suite of algorithms (`cipher_suite`) and finally a *master secret*. All of the necessary algorithms and the `cipher_suite` are selected during the Client and Server Hello messages as well as the random values. The master secret now needs to be calculated.

When a key exchange method is used to create a pre master secret, it is the same algorithm that converts the pre master secret into the master secret. When this conversion takes place, the pre master secret is deleted from memory. TLS uses a Pseudorandom Function (PRF) to calculate the `master_secret` from the `pre_master_secret` and the client and server's random values as described below.

```
master_secret = PRF(pre_master_secret, "master secret",
                    ClientHello.random + ServerHello.random)
                    [0..47];
```

A master secret used in TLS will always be 48 bytes in length, and when all this is done, the client and the server may begin to communicate data over the application layer. All *Application Data* between the server and the client is now sent over a newly established secure channel as denoted by the dotted line in Figure 3.3.

Chapter 4
Attacking TLS

As the Transport Layer Security protocol is by far the most common source of application security today, it has been extensively researched in order to uncover design flaws [31]. This chapter aims to introduce some of the different attacks aimed towards breaking TLS.

4.1 Summarizing Known Attacks on TLS

When developers implement application security, the TLS protocol remains the supplier of this security. Several attacks on TLS have been discovered over the last years, and many of them have proven to be of serious threat. Although this section is not intended as a very detailed list of some of the many attacks made in an attempt to break TLS, it is meant to be an introduction to the many issues associated with TLS before looking further into attacking the termination process of the protocol. Below the reader will find a list of some of the other well-known attacks against TLS as described by RFC 7457 [31].

- **SSL Stripping:** describes a selection of attacks aimed towards modifying unencrypted protocols that requests the use of TLS. These modifications are made with the goal of removing the use of SSL/TLS.
- **BEAST:** the Browser Exploit Against SSL/TLS (BEAST) attack is an attack that successfully proved the possibility of decrypting HTTP cookies when HTTP was running over TLSv.1.0.
- **Padding Oracle Attacks:** is a selection of attacks that relies on the way TLS adds a MAC before encryption. Among others, this selection of attacks include:
  - Lucky Thirteen Attack
  - POODLE Attack

- **Attacks on the RC4 encryption algorithm:** describes several attacks attempting to exploit cryptographic weaknesses in the RC4 algorithm.
- **Compression Attacks:** is the common name given as a description to attacks that focus on the data compression in TLS. These attacks include:
  - CRIME Attack
  - TIME Attack
  - BREACH Attack
- **Renegotiation Attack:** describes an attack on the mechanism that handles session renegotiation in TLS.
- **Triple Handshake Attack:** is an attack focused around enabling an attacker to cause shared keying material between two TLS connections.
- **Denial of Service:** is an attack aimed towards exhausting computing or network resources by for example sending a massive amount of requests towards these resources.

This selection of attacks have been around for a varied amount of time. From the perspective of the authors of RFC 7457, the attacks on TLS are only getting more and more complex. However, many of these attacks have already been addressed, and several more will be, pending the publication of TLSv1.3 [27]. TLSv1.3 is still a working progress draft, and it is, therefore, the responsibility of a web application designer to be aware of which necessary steps that need to be taken to counteract many of these attacks.

4.2 Truncation Attack

When a TLS connection is being torn down, a truncation attack can be attempted. Historically, the truncation attack started by exploiting a design flaw in version 2 of the SSL protocol. SSLv2 allowed either side of a connection to send a `TCP FIN` flag, initiating a termination of the session [12]. In a truncation attack against SSLv2, an attacker can, by forging a `TCP FIN`, make it appear as though a message is received shorter than intended by the sender. As an example of this, imagine an application using SSLv2 and that fragments data records into blocks. Suppose further that this application handles wire transfers and that a user wishes to initiate a wire transfer to *Charlie's angels*. Wire transfers in this application come in the form of a HTTP request, and may look like the following.

```
POST /wire_transfer.php HTTP/1.1
Host: mybank.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 40

amount=1000%recipient=Charlie%25_Angels
```

Let us now say that TLS fragments this request in the following way: "`POST [...] recipient=Charlie`" and "`%27s_Angels`". In this case, an attacker drops the last fragment of the message by forging a `TCP FIN` and thereby closes the underlying TCP connection. By forging a `TCP FIN` after the first fragment has been received, the receiver (or server) will now only receive the part of the recipient field that says *Charlie*. The result of this is that the wire transfer will end up there rather than with the intended *Charlie's Angels* [35].

This example application ignores both the connection termination status and the `Content-Length` field, meaning that the application accepts all payload shorter than the specified length [35]. This type of attack works on this application because it does not include in its implementation any way for the receiver to know what message size to expect, and thus the receiver believes that the message received is the correct one.

To counteract this obvious design flaw, the design of SSLv3/TLSv1 introduced an alert message called `close_notify` (see Section 3.2.2). This `close_notify` message makes all the difference in the example above. When the user sends his message, the receiver will wait for a `close_notify` message to indicate that the message is finished. If the last part of the message is then deleted by an adversary, the `close_notify` will not arrive either. From this, the receiver recognizes that the message is incomplete.

4.2.1 Termination Modes

When defining the guarantees of TLS security it is done with respect to two termination modes. The two of termination modes in TLS are *graceful* connection closure and *fatal* closure. A graceful closure occurs in the event of a successful end to a connection, and with it, TLS guarantees that *all* messages are received as sent. A fatal closure occurs when something goes wrong when ending a connection, and with it, TLS only guarantees that a *prefix* of all messages is received as sent. As described in Section 3.1.2, TLS allows fragmentation of messages. In a similar way as with messages, TLS guarantees the delivery of *all* fragments in order upon graceful closure, and a *prefix* of all fragments upon fatal closure. Ignoring these termination modes has been the case for most web browsers in the past [35, 13], and there are

still many web browsers which do not distinguish between these two termination modes resulting in them being vulnerable to truncation attacks.

Chapter 5 describes several cases where web browsers inability to distinguish between these modes may lead to serious consequences. However, while the aforementioned example focused on truncating TLS fragments, the attacks in Chapter 5 focus on truncating entire messages.

4.2.2 The Cookie Cutter Attack

Truncating TLS messages and fragments, as described in Section 4.2, is not the only way truncation attacks may affect users. Still exploiting many browsers' lack of distinction between TLS termination modes, a new type of truncation attack was presented in 2014, called the *Cookie Cutter* attack [13]. While the truncation attack described above is focused on the termination procedure of SSL/TLS connections, the cookie cutter attack can be applied when a login form is posted by truncating *headers* of HTTP messages.

HTTP Cookies are small pieces of data stored in a user's web browser while browsing different websites [10]. These cookies are sent from the website a user is visiting, and their goal is to store information about the user. Such information may include stateful information (i.e. shopping carts etc.) and browsing activity as well as previously entered usernames and passwords related to the user. When a server wishes to send cookies to a user, it is done by using a `Set-Cookie` HTTP response header. After this header, a `secure` flag can be appended in order to protect the cookie's confidentiality [10]. Note that this flag is appended *after* the cookie has been set, making it possible to truncate. By truncating this flag and redirecting the user to an unencrypted URL, an attacker can recover the information stored in the cookie.

Let's review an example web application located at `https://x.com/` which uses a login form at `https://x.com/login?go=P` to set a session cookie and to further redirect the user to `https://x.com/P`. The response header from this login form will then look something like the following.

```
HTTP/1.1 302 Redirect
Location: https://x.com/P
Set-Cookie: SID=[AuthenticationToken]; secure
Content-Length: 0
```

After a successful login to a web application, a redirection often occurs. In the response header after such a redirection, a `Location` field is included, which again

includes parameters taken from the request sent. These parameters often describe the page a user accessed before trying to log in, and they can often be controlled by an attacker if he controls the network. An attacker who controls the network, can, by using different techniques, control the TLS fragmentation explained in Section 3.1.2. These techniques involve triggering a request with a selected path and injecting data to this request's `Cookie` header. In the header above, an attacker can choose a value for `P` that causes the fragment to end just before the "`;`" character. By now truncating the second fragment, the cookie will be stored without its intended secure flag, making it available for an attacker to review [13].

As mentioned on several occasions in this chapter, these attacks are possible due to web browser's poor handling of TLS termination modes. Concretely, the web browsers Chrome, Opera, and Safari all accepted incomplete HTTP headers when the Cookie Cutter attack was discovered. Bhargavan et.al. [13] summarized the possible truncations in modern browsers. Due to the increased likeliness that mobile versions of browsers connect to untrusted networks, they received more focus as Table 4.1 suggests.

Browser	In-Header truncation	Content-Length ignored	Missing last chunked fragment ignored
Android 4.2.2 Browser	✓	✓	✓
Android Chrome 27	✓	✓	✓
Android Chrome 28	✗	✗	✓
Android Firefox 24	✗	✓	✓
Safari Mobile 7.0.2	✓	✓	✓
Opera Classic 12.1	✓	✓	✓
Internet Explorer 10	✗	✓	✓

Table 4.1: Successful TLS truncations in mobile browsers [13]. ✓indicates that the truncation was successful, and ✗ indicates that the truncation failed.

Table 4.1 show a selection of mobile browsers and their vulnerability to different types of truncation. As indicated by ✓in Table 4.1, a web browser is only susceptible to the Cookie Cutter attack if *In-Header* truncation is allowed, the *Content-Length* field is ignored, and if the browser ignores cases where the last TLS fragment is *Missing*. The research done by Bhargavan et.al. allowed them to successfully launch the Cookie Cutter attack against Google Accounts in 2014.

Chapter 5
Truncating TLS Connections

In a paper published in 2013 by Ben Smyth and Alfredo Pironti [35], it was shown that certain web application flaws could be exploited by performing truncation attacks, as described in section 4.2, on TLS connections. The following chapter will provide a practical investigation of how the authors were able to cast votes on behalf of honest users in an electronic voting system (Helios), gain temporary access to Google accounts as well as gain full access to Microsoft Live (Hotmail) accounts. As the paper by Smyth and Pironti was published three years ago, changes may have been made in the logic of these applications. Because of this, this chapter will also include an assessment of the effects the same attacks have on the same applications three years later.

5.1 Violating Beliefs

There exists no guarantee from TLS that an application is protected from logical flaws. As stated in Chapter 3, it is the job of the designer to make the correct choices when implementing application security. Furthermore, TLS does not guarantee the ordering of messages in multiple connections within the same session. Only the correct ordering of a single connection is ensured, and a web application normally notifies a user of a server's state by using some form of mechanism to provide the user with either positive or negative feedback. While negative feedback typically comes in the form of an error message, positive feedback provides the user with information about a successful state change. A successful state change can be in the form of a "You have been logged out" message, and providing a user with this type of feedback at the wrong time can be described as a logical application flaw. Smyth and Pironti noticed that these types of logical flaws were not uncommon, and by focusing on web applications providing users with positive feedback *before* the actual state change occurs, they were able to attack the sign-out procedures of real-world web applications. During the course of the attacks described in this chapter a fair amount of information is obtained by using some basic traffic analysis techniques.

The reader is in these cases referred to Section 2.5 which provides an introduction to these network analysis techniques.

5.2 Setup

The three attacks following this section are made possible when making a selection of assumptions and using the setup described in this section. It is assumed during the course of these attacks that an adversary has full control of the network and that the web applications are able to achieve their objectives despite of this. An adversary's complete control of the network will allow reading, deleting and injecting messages. An honest user is using a shared computer (i.e. public library computer) which the adversary also has access to but cannot tamper with. The shared computer was modeled as a VM running Ubuntu 10.04.4 Long Term Support (LTS) and the network was modeled as the host computer controlled by an adversary. During the recreation process of imitating the attacks described by Smyth and Pironti, the shared computer was modeled as another VM running Ubuntu 14.04.3 LTS with the newest version of Firefox, and the network was modeled as before. This LTS Ubuntu version comes with five years of security and maintenance updates [4]. During the course of recreating the attacks described by Smyth and Pironti, experiments were made using a selection of other setups. These experiments are explained further in Chapter 6.

5.3 Helios Electronic Voting System

The Helios Electronic Voting System is an open source, end-to-end verifiable electronic voting terminal created to provide an online version of secure ballot casting in elections [8]. After an analysis of the authentication logic of the Helios release from 2012, Smyth and Pironti discovered a flaw in the authentication logic. Upon casting a vote in Helios, users are being automatically signed out. However, Smyth and Pironti discovered that voters are given feedback saying that they have been signed out *before* the actual log off request is made.

The trace shown in Listing 5.1 describes the procedure of requests made by a browser when a user chooses to cast a vote. Request number one in this trace is answered by a redirect by the server as the user confirms his vote. The redirect from the first request is handled by the second, the server then responds with an Hypertext Markup Language (HTML) payload containing a ballot receipt and a notification message informing the user that a successful logout has occurred. The third and final request described in Listing 5.1 is the actual sign-out request. This request is also answered by a redirect from the Helios server. By being aware of this authentication flaw, an adversary will be able to drop log off requests if the adversary has, as described in Section 5.2, full control over the network.

```
1. POST https://vote.heliosvoting.org/helios/elections/id/
   cast_confirm
   Response: 302 - Moved Temporarily
   Location[https://vote.heliosvoting.org/helios/elections/
   id/cast_done]
2. GET https://vote.heliosvoting.org/helios/elections/id/
   cast_done
   Response: 200 - OK: HTML payload
   ...
   <p><b>For your safety, we have logged you out.</b></p>
   <iframe border="0", src="/auth/logout" frameborder="0"
   height="0" width="0">
   </iframe>
   ...
3. GET https://vote.heliosvoting.org/auth/logout
   Response: 302 - Moved Temporarily
   Location[https://vote.heliosvoting.org/]
```

Listing 5.1: Trace for voting/sign-out procedure in Helios [35]

5.3.1 The Attack

The actual attack made on the Helios voting system published in 2013, makes use of the authentication flaw described above. In a video published 14 September 2012 [32], Ben Smyth demonstrates this attack as its goal is to truncate a voter's sign-out request, request number 3 in Listing 5.1, after the feedback of a successful logout has been sent. In this case the server will never be notified of the pending logout request, and the voting terminal and the server will still have an active session.

As mentioned in Section 4.2, TLS does not provide enough protection against this kind of attacks. One TLS connection is independent from another, and the protocol does not guarantee the ordering of messages between several connections within the same session [35]. This means that when an adversary drops request number 3 in the logout procedure, he would still be able to make a future connection with the helios server. This will again give him the opportunity of casting votes on behalf of honest users. The fact that all requests described in Listing 5.1 are encrypted using TLS does not cause any major problems when carrying out this attack. The sign-out request was, after some basic traffic analysis, recognized by Smyth and Pironti by its fixed length of 701 bytes. By using *iptables*, a firewall could be setup at the host and configured to drop all packets of this size. This is done by issuing the following command in the host's terminal window:

```
iptables -A OUTPUT -m length --length 701 -j DROP
```

After the host's firewall is configured with this command, all packets of length 701 are dropped when arriving from the VM (or voting terminal). Even though this occurs without the voter noticing, the attack can be detected at a later stage given that Helios is an end-to-end verifiable system meaning that voters can review their own votes [8]. Helios does not, however, provide accountability in the event of such attacks. The reasoning behind this, is that the honest voter cannot prove that an adversary has been responsible for votes cast after a certain point in time.

Countermeasures With the discovery of this vulnerability in the Helios voting system, a few countermeasures were suggested with the publishing of the results from Section 5.3.1. An obvious patch for this logic flaw is to make sure the feedback suggesting a successful logout does not appear before the actual sign-out request. Another suggestion is to make all actions in Listing 5.1 atomic, meaning that all three requests are made simultaneously. Also worth mentioning is that closing the browser completely and or clearing the cache, cookies, and otherwise manually destroying the session, will render the attack useless.

5.3.2 Recreating the Attack

Now, three years later, the trace of requests made by the browser when a voter casts a vote is identical to the trace described in Listing 5.1. In fact, a capture made in Firefox (see Figure 5.1) during a ballot casting [1] show the sign-out procedure as identical to the one from three year ago.

Figure 5.1: Firefox capture of trace in Helios

After some network analysis it became apparent that request number 3 no longer is 701 bytes in length. Instead the sign-out request now has a size of 551 bytes. This is due to the fact that a different browser version is being used. By using *iptables*, a firewall could be setup at the host and configured to drop all packets of this size. This is done by issuing the following command in the host's terminal window:

```
iptables -A OUTPUT -m length --length 551 -j DROP
```

[1] Capture made 28 April 2016

Issuing this command will effectively drop the packet containing the sign-out request, however this attack is no longer effective on the setup described in Section 5.2. As it turns out, the issue of web browsers ignoring the termination modes described in Section 4.2.1 has been addressed. Firefox does now force the connection to terminate when it notices that something happened to the request.

Given the scenario considered when conducting these types of attacks, it would be interesting to see whether the results differ if the setup is more common. As shown in Table 5.1, when all versions of Microsoft Windows are considered, it is by far the most popular operating system to date. When considering the fact that most computers that are shared (i.e. work computers, library computers etc.) are running Windows with its default web browser Internet Explorer (IE) the attacks should be recreated by using a VM running Windows rather than Ubuntu.

Table 5.1: Most widely used operating systems [3]

Windows	Mac OS X	Linux	Other
78,9%	10,6%	5,5%	5,0%

From Microsoft's own developer site, an already configured VM can be downloaded and used for research purposes. In this next attack, the shared computer is modeled as a VM running Windows 7 using IE11 (version 11 of Internet Explorer) as its default browser. The network is again modeled as the host and controlled by an adversary. After some network analysis, it turns out that the sign-out request used by Helios when casting a vote is in this scenario is 569 bytes in length. As before, an adversary configures the host's iptables to drop packets of this size by issuing the following command:

```
iptables -A OUTPUT -m length --length 569 -j DROP
```

By using this setup the user of the shared computer is not secure. By dropping the sign-out request in this scenario, the session with Helios is never terminated. An adversary can now simply refresh the Helios page and find a still active session belonging to the honest user. The absence of distinction between the two termination modes described in Section 4.2.1 turns out to be crucial for the attacks described by Smyth and Pironti. IE11 lacks this distinction, meaning that truncation attacks on TLS connections are still a threat to shared computers.

5.4 Microsoft Services

During the analysis of several different web applications, Smyth and Pironti learned that the Microsoft authentication logic has a flaw. Exploiting this flaw could give an adversary full access to a user's account. The trace in Listing 5.2 shows the most significant details of a normal sign-out procedure as seen by a browser. This trace also shows that, when a user is logged in on his Microsoft account, he is logged into several Microsoft services, each of which stores some session information. When a user chooses to sign out, it is not a centralized action. This means that the connection to each Microsoft service is terminated separately. The procedure that is initiated when a user signs off uses HTTP redirects and HTML pages with embedded Javascript code. Javascript code, in this procedure, is loaded as shown in request 2 and 3 in Listing 5.2.

```
1. GET https://login.live.com/logout.srf?ct=1364567198&rver
   =6.1.6206.0&lc=1033&id=64855&ru=http\%2F\%2Fbay171.mail.
   live.com\%2Fhandlers\%2FSignout.mvc\%3Fservice\%3DLive.
   Mail&mkt=en-fr
   Response: 200 - OK - HTML payload
2. GET https://secure.shared.live.com/~Live.SiteContent.ID
   /~17.0.11/~/~/~/~/js/Login_Core.js
   Response: 200 - OK.
3. GET https://secure.shared.live.com/~Live.SiteContent.ID
   /~17.0.11/~/~/~/~/js/Login_Alt.js
   Response: 200 - OK.
4. GET https://account.live.com/logout.aspx?ct=1364567254
   Response: 200 - OK
5. GET https://accountservices.msn.com/LogoutMSN.srf?ct
   =1364567254
   Response: 200 - OK
6. GET http://bay171.mail.live.com/handlers/Signout.mvc?
   service=Live.Mail&lc=1033
   Response: 302 - Moved Temporarily,
   Location[http://g.live.com/9ep9nmso/so-EN-US]
7. GET http://g.live.com/9ep9nmso/so-EN-US
   Response: 301 - Moved Permanently,
   Location[https://signout.live.com/content/dam/imp/
   surfaces/mail_signout/v7/mail/en-us.html]
8. GET https://signout.live.com/content/dam/imp/surfaces/
   mail_signout/v7/mail/en-us.html
   Response: 200 - OK
```

Listing 5.2: Trace of the sign-out procedure for Microsoft Live [35]

Following this code, the user is signed out from *account.live.com* in step 4, *accountservices.msn.com* in step 5 and *mail.live.com* in step 6. After every connection the user has with Microsoft services have been terminated, the user is finally redirected to the Microsoft Live homepage where he is presented with a message telling him "You've been signed out".

From studying this decentralized sign-out procedure, Smyth and Pirtonti found that there was a flaw in the authentication logic which could be exploited in a very specific way.

5.4.1 The Attack

The exploitation of this authentication flaw comes in the form of another truncation attack. The idea behind this attack is as follows. A user first signs into his *Hotmail* account by being sent to *login.live.com* where he authenticates. The user is then redirected to a page where he can read his mail. Furthermore, as the user is already signed in to Microsoft Live, he is automatically authenticated when he visits *account.live.com* [34]. When the user is done, he sends a sign-out request to Hotmail, and Microsoft's authentication server responds by initiating a logout procedure similar to the one described in Listing 5.2 where the goal is to terminate the sessions with both Hotmail and *account.live.com*. While this is happening, an adversary with full control over the network decides to drop the request terminating the user's session with *account.live.com* (request number 4 in Listing 5.2). Given the fact that the sign-out requests 4-6 are independent from each other, the user will still receive positive feedback indicating a successful logout. An illustration of this attack is given in Figure 5.2 which indicates that the procedure proceeds as normal even though one request has not been received. When the user then proceeds to leave the computer, an adversary can use the same computer to access *account.live.com* and all account information from the previous user will be available.

The truncation attack on Microsoft services is demonstrated in a video published 16 August 2013 [34]. This demonstration follows the same setup described in Section 5.2, and the network is configured to drop packets by using the following command at the host:

```
iptables -A OUTPUT -m length --length 474:506 -j DROP
```

In advance of issuing this command, basic traffic analysis techniques were used by Smyth and Pironti to find that the request terminating a user's session with *account.live.com* (request 4 in Listing 5.2) is between 474 and 506 bytes in length. After truncating a user's TLS connection, the video demonstrates how an adversary can take full control of his account by accessing *account.live.com* as the user. When

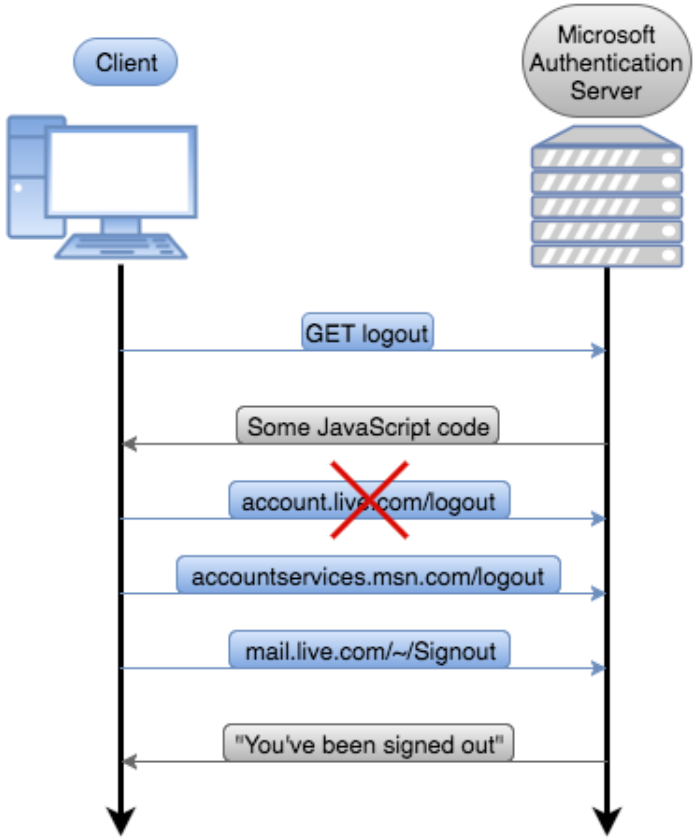

Figure 5.2: Illustration of the attack against Microsoft Live

logged in, the adversary will be able to add a recovery email to the honest user's account without having to reauthenticate. The adversary can then initiate a reset of the current account password and allow a new password to be sent to the recently added recovery email (belonging to the adversary). From this point on, the adversary has full control over the account.

Countermeasures Along with the published results from this attack, Smyth and Pironti also suggested some countermeasures to avoid this attack. Like with the countermeasure suggested for the attack on the Helios voting system (see Section 5.3.1), manually closing the browser completely and or clearing the cache, cookies, and otherwise destroying the session, will render the attack useless. Another, more complicated countermeasure, is for the Microsoft authentication server to handle the sign-out procedure centrally as well as requiring reauthentication when adding account information. The final countermeasure suggested by Smyth and Pironti

was to modify the entire sign-out procedure shown in Listing 5.2. This modification entails handling the sign-out procedure as a chain of HTTP redirects over TLS. By employing this solution, a HTML page including Javascript is never returned, and if one request is dropped, the next one will not be initiated which again will lead to a browser crash or an error message. Listing 5.3 describes such a solution.

```
1. GET https://login.live.com/<<signout>>
   Response: 302 - Moved Temporarily
   Location[https://account.live.com/<<signout>>]
2. GET https://account.live.com/<<signout>>
   Response: 302 - Moved Temporarily
   Location[https://accountservices.msn.com/<<signout>>]
3. GET https://accountservices.msn.com/<<signout>>
   Response: 302 - Moved Temporarily
   Location[https://mail.live.com/<<signout>>]
4. GET https://mail.live.com/<<signout>>
   Response: 302 - Moved Temporarily
   Location[https://signout.live.com]
```

Listing 5.3: Countermeasure for the sign-out procedure in Microsoft Live [35]

5.4.2 Recreating the Attack

Three years after the publication of the attack, the sign-out procedure for Microsoft accounts is made very different. As Figure 5.3 suggest, the countermeasures have now been taken into consideration [2]. The first detail worth mentioning is that *account.live.com* and *mail.live.com* now both reside under *outlook.live.com* suggesting a more centralized design. Also indicated by this capture is the fact that each session is terminated as a chain of HTTP redirect, as indicated by the 302 response code on the left in Figure 5.3. Upon attempting to block any of these three sign-out requests, the browser hangs — leaving the original attack from 2013 unsuccessful.

● 200	POST	vevent?e=wqT_3QLIBPB-SAIAAAMA1gAFAQiNkdG...	secure-ams.adnxs.com
● 200	GET	logoff.owa	outlook.live.com
● 200	GET	logout.srf?ct=1463044277&rver=6.6.6556.0&id=29...	login.live.com
▲ 302	GET	csignout.aspx??umkt=en-US&exch=1&lc=1033	outlook.live.com
▲ 302	GET	csignout.aspx??umkt=en-US&exch=1&lc=1033	outlook.live.com
▲ 302	GET	/?ocid=mailsignout	www.msn.com
● 200	GET	/nb-no/?ocid=mailsignout	www.msn.com

Figure 5.3: Firefox capture of the new sign-out procedure for Microsoft Live

[2] Capture made 12 May 2016

Apart from the difference in the sign-out procedure, Microsoft has made another adjustment to their services. As the attack was explained in Section 5.4.1, it relied heavily on the adversary's intention of adding a recovery email to the user's account in order to initiate a password reset. When attempting to add a recovery email to an account today, the user is shown a message indicating that at least 30 days must pass before the new security info is updated. This message appears as shown in Figure 5.4, and should give the owner of the account plenty of time to discover any malicious intentions of an adversary.

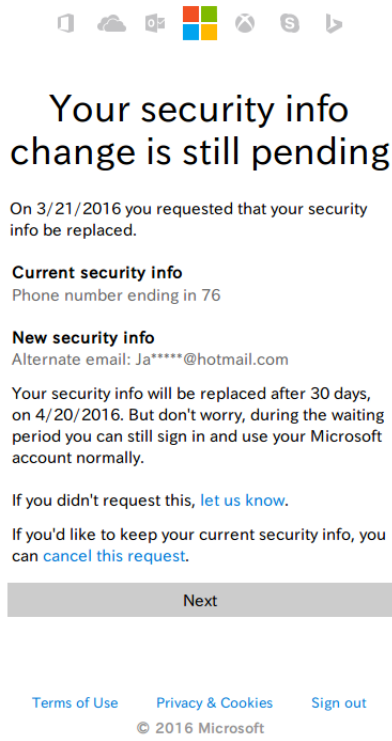

Figure 5.4: Microsoft services' 30 day rule

5.5 Google Services

The final attack described in Smyth and Pironti's paper from 2013 is one targeting Google accounts. Like the two previous attacks described in this chapter, this attack is possible due to a flaw in authentication logic. The trace in Listing 5.4 describes the sign-out procedure from the browser's point of view when a user makes a sign-out request from Google services. Although the trace does not contain every single

detail of the procedure, it can be observed that the first action is that the session with *accounts.google.com* is terminated and that a redirect follows. This redirect responds with an HTML page containing an image and some Javascript code with a `doRedirect()` function. Notice how the Javascript code is executed after the image has been loaded. The third request observed in Listing 5.4 is the browser asking the *Gmail* server for the image in request number 2. Before responding with the requested image the server terminates the user's session with *mail.google.com*. Having already loaded the `doRedirect()` function from request 2, the browser is now redirected as observed in step 4 and 5, before request number 6 loads the home page of Google.

```
1. GET https://accounts.google.com/Logout?continue=https://
   www.google.com/webhp
   Response: 302 - Moved Temporarily
   Location[http://www.google.com/accounts/Logout2?ilo=1&ils
   =mail,s.FR&ilc=0&continue=https://www.google.com/webhp?zx
   =1388193849]
2. GET http://www.google.com/accounts/Logout2?ilo=1&ils=mail
   ,s.FR&ilc=0&continue=https://www.google.com/webhp?zx
   =1388193849
   Response: 200 - OK; HTML payload:

   <body onload="doRedirect()">
   <script type="text/javascript">
     function doRedirect() {
     location.replace("http://www.google.fr/accounts/Logout2
    ?ilo=1&ils=s.FR&ilc=1&continue=https://www.google.com/
    webhp?zx=1076119961");
     }
   </script>
   <img width="0" height="0" alt="Sign Out"
     src="https://mail.google.com/mail?logout=img&zx
    =-2531125006460954395">
   </body>
3. GET https://mail.google.com/mail?logout=img&zx
   =-2531125006460954395
   Response: 200 - OK; a one pixel gif.
4. GET http://www.google.fr/accounts/Logout2?ilo=1&ils=s.FR&
    ilc=1&continue=https://www.google.com/webhp?zx=1076119961
   Response: 200 - OK; HTML payload:
   <body onload="doRedirect()">
   <script type="text/javascript">
```

```
    function doRedirect() {
     location.replace("https://www.google.com/webhp");
    }
   </script>
   <img width="0" height="0" alt="Sign Out"
     src="https://accounts.google.fr/accounts/ClearSID?zx
    =-1920517974">
   </body>
5. GET https://accounts.google.fr/accounts/ClearSID?zx
    =-1920517974
   Response: 200 - OK; a one pixel gif.
6. GET https://www.google.com/webhp
   Response: 200 - OK
```

Listing 5.4: Trace of the sign-out procedure for Google services [35]

5.5.1 The Attack

The attack on Google services was demonstrated by a video published 16. August 2013 [33]. In this demonstration it was assumed that a user has at least two active sessions with Google services (namely *Gmail* and *Google Search*) on a shared computer and that an adversary, as before, controls the network. This attack works as follows. A user is logged in to *Gmail* with an authenticated session and decides to visit *Google Search*. Upon visiting *Google Search*, a new session is seamlessly authenticated by Google. When done, the user makes a sign-out request which initiates the sign-out procedure described above. This procedure is, however, not secure because an adversary can prevent termination of the session the user has with *Gmail*. This prevention can be achieved by rejecting request number 3 containing the image from *Gmail* mentioned above. By truncation the TLS connection using a `TCP reset`, the session will not be terminated as the image will be prevented from loading. The browser will still think all content from the request is loaded and will continue to run the `doRedirect()` function. However, the image sent out to terminate the session has failed to do so, and despite an unsuccessful logout, the user has been given positive feedback indicating a successful one.

In advance of the demonstrated attack, basic network analysis methods were applied in order to find that request number 3 was between 1165 and 1195 bytes in length [33]. Furthermore, the demonstration show that request number 3 can be dropped by configuring iptables with the following command:

```
iptables -A OUTPUT -m length --length 1165:1195
  -p tcp -j REJECT --reject-with tcp-reset
```

When Google's sign-out procedure reaches request number 3, it will be rejected. Configuring the firewall to reject the said request with a `TCP reset` will cause the browser to abort loading the image and proceed to execute the rest of the Javascript code. When the user now chooses to leave the shared computer, an adversary can simply refresh the *Gmail* page and a session will still be active.

Countermeasures The same type of countermeasures recommended as a solution for the attack on Microsoft Live are recommended in order to avoid this type of attack on Google services. Smyth and Pironti stress that an entirely centralized authentication is needed in order to counteract this threat completely. Also in this case could the sign-out procedure by handled as a chain of HTTP redirects as an alternative. The last solution could be to add a `onerror` Javascript handler to all images included in the returned HTML pages during the sign-out procedure. Such a handler will return an error message when an image or gif fails to load and the user will be shown this message as the sign-out procedure is halted.

5.5.2 Recreating the Attack

Like with the other attacks described in this section, the premise of this attack has changed over the course of three years. After some research and network analysis, it is apparent that even though the sign-out procedure used by Google services remains the same, some of the requests look slightly different. For instance, the length of request number 3 in Listing 5.4 is no longer between 1165 and 1195 in bytes. After applying basic network analysis techniques now appear to be between 490 and 520 bytes in length. Like the approach three years ago, this request can be rejected using a `TCP reset` by issuing the following command at the host:

```
iptables -A OUTPUT -m length --length 490:520
  -p tcp -j REJECT --reject-with tcp-reset
```

It the same way as before, it can be observed from the capture in Figure 5.5 that the "remotelogout?zx=«...»" request is being dropped (as indicated by the grey dot on the left hand side) [3]. Furthermore, the capture shows that the procedure carries on despite the fact that the image download has been aborted. However, despite the fact that the request terminating the user's session with *Gmail* no longer arrives at its destination, an adversary would still not be able to simply continue the session with *Gmail* as before.

Figure 5.5 also shows that one other image is loaded after the image described above. After some network analysis, it turns out that the *ClearOSID* image is

[3]Capture made 12 May 2016

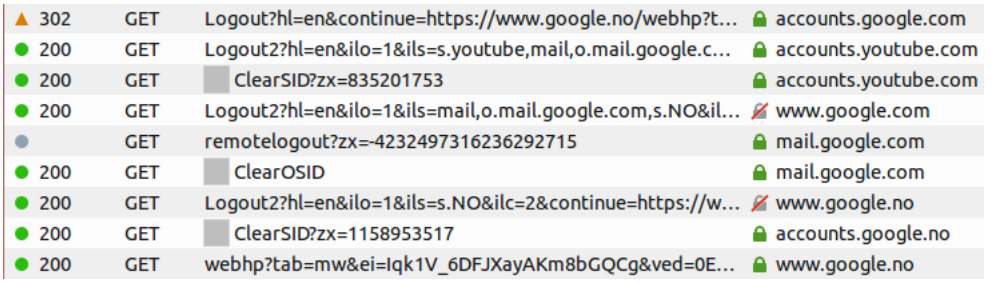

Figure 5.5: Firefox capture of trace in Google

also being loaded from the Javascript code in request number 2. When reviewing the sign-out procedure during a log out from a Windows VM running IE11, these two images appear to be between 1045 and 1085, and 923 and 953 bytes in length, respectively. Believing that both of these images are used as a termination request of the session with *mail.google.com*, they were rejected with a `TCP reset` by the use of the following two commands:

```
iptables -A OUTPUT -m length --length 1045:1085
  -p tcp -j REJECT --reject-with tcp-reset

iptables -A OUTPUT -m length --length 923:953
  -p tcp -j REJECT --reject-with tcp-reset
```

Successfully rejecting these two requests resulted in a error page with the message "This page can't be displayed". This page causes all active sessions with Google to be terminated, leaving the attack ineffective. The truncation attack against Google accounts as it was described by Smyth and Pironti was also ineffective on the different setups described in Chapter 6. Based on these results, it would appear as though Google has fixed the errors discovered in their authentication logic. Google does not, however, appear to have provided any documentation explaining these adjustments.

5.6 Summary

This chapter provided a practical investigation of a selection of web applications attacked by truncating their TLS connections, as described by Smyth and Pironti in 2013 [35]. After reviewing the approach the authors had when producing these attacks, attempts were made to check their validity three years after the initial results were published. As the TLSv1.2 protocol remains the same, the results of the recreated attacks depend on any changes in web application or web browser design.

Neither one of the original attacks was successful when using the same setup as that explained by Smith and Pironti. However, the truncation attack against the Helios Voting system proved to still be effective against a selection of other web browsers under an honest voter.

The findings made in this chapter will be further investigated, and the next chapter will provide an analysis of the extent that different web browsers are susceptible to truncation attacks as described in Section 5.3.2.

Chapter 6
Secure Termination In Web Browsers

Upon the discovery made in Chapter 5 that truncation attacks on TLS connections still work under certain conditions, it seemed apparent that the different web browsers differ from each other in a major way when it comes to interpreting the TLS protocol. Specifically, many modern web browsers seem to have poor handling of the termination modes of TLS as described in Section 4.2.1. By replicating once more the truncation attack against the authentication logic of the Helios Voting system, this chapter aims to uncover to which degree modern web browsers are vulnerable to such truncation attacks.

6.1 Handling TLS Termination Modes

Concerns about the poor handling of termination modes has been expressed on several occasions [13, 35]. Section 3.2.2 describes the use of the `close_notify` alert message, which has, since SSL3, been required in order to close a TLS connection. When a session is terminated by the use of a `close_notify`, it will indicate that a *graceful* closure has occurred, as described in Section 4.2.1. By distinguishing between a *graceful* and a *fatal* closure, applications are in theory capable of preventing truncation attacks and other Man In The Middle (MitM) attacks. By for instance recognizing a *fatal* closure, a web browser would effectively disallow the continuing of a connection. However, many applications and indeed many web browsers and HTTP servers do not distinguish between these two types of closure. Whether this is due to unawareness or if it is deliberate to increase compatibility is unclear, but in any case, it leaves the software susceptible to truncation attacks.

When the Cookie Cutter attack was discovered and published in 2014 (See Section 4.2.2), the authors summarized possible truncations in modern mobile web browsers. As shown in Table 4.1 the result of this summarization was that most mobile web browsers were susceptible to at least one variation of truncation attacks. As these results focused mainly on mobile browsers, the next section will focus on desktop

browsers and validate the success of an attempted truncation attack on the Helios Voting system, as it was described in Section 5.3.2.

6.2 Truncation in Web Browsers

A web browser is a software application intended to transfer information over the World Wide Web. All information processed by a browser can be identified by a URL/URI, and both modern mobile and desktop browsers are receiving an increasing number of responsibilities when it comes to performing operations related to security [25, 9]. Table 6.1 show the most popular desktop browsers in use, and this section will determine to which degree these modern web browsers handle the termination modes of TLS.

Table 6.1: Most widely used web browsers [5]

Google Chrome	Internet Explorer	Firefox	Safari	Opera
70,4%	5.8%	17,5%	3,7%	1.3%

6.2.1 Google Chrome

The first target in this practical investigation is the latest version (version 50 [1]) of the most popular web browser, *Google Chrome*. The setup of this attack includes a VM running Ubuntu 14.04 with the Chrome browser, and a Linux Mint machine as the host modeling the network. As before, an adversary is assumed to control the network and a user is about to cast a vote using the Helios Voting system. The network analysis techniques described in Section 2.5 were used to determine that the sign-out request (`https://vote.heliosvoting.org/auth/logout/`) from Section 5.3 had a size of 623 in bytes. Again, this request was dropped by configuring the host's iptables with the following command:

```
iptables -A OUTPUT -m length --length 623 -j DROP
```

After successfully casting a vote, the user proceeds to leave the computer believing that he has been logged out. From monitoring the network traffic during the time of the attack, no *Encrypted Alerts*, or `close_notify` alerts, can be observed. This indicates that the *fatal* closure is ignored. As it turns out, Google Chrome does *not* distinguish between the two termination modes of TLS, and by refreshing the Helios home page, an adversary will be able to cast new votes on behalf of the honest user.

[1] As of May 20, 2016

6.2.2 Opera

When moving on to analyzing the latest version (version 37 [1]) of the *Opera* web browser, the same setup as above was used. Once again, basic network analysis was initiated to find that the sign-out request from the vote casting procedure had a size of 645 bytes. The host's iptables was then configured to drop all packets of this size:

```
iptables -A OUTPUT -m length --length 645 -j DROP
```

The truncation attack was once again successful, and the Opera does also seem to handle TLS termination modes poorly.

6.2.3 Internet Explorer and Microsoft Edge

In Section 5.3.2 a truncation attack was successfully attempted on the Helios Voting system. The premise of this success was the poor handling of TLS termination modes of IE11. With the release of Microsoft's newest operating system, *Windows 10*, in 2015 a new browser was released. This next truncation attack attempt will focus on this browser, *Microsoft Edge*, as it is running on Windows 10. This time, the host's iptables was configured to drop all packets of size 574 bytes.

```
iptables -A OUTPUT -m length --length 574 -j DROP
```

With the command above, the sign-out request from Helios was successfully dropped and the truncation attack was successful on Microsoft Edge as well.

6.2.4 Results

In the paper published in 2013 by Smyth and Pironti [35], the setup described in Section 5.2 was used and the user was using a Firefox browser. Although it was not specified which version of Firefox that was used, it is safe to assume, based on Firefox's own release notes, that the version number was close to number 20 at this time. As previously stated, the attacks attempted by Smyth and Pironti were successful due to Firefox's poor handling of termination modes.

In an attempt to replicate these attacks when using newer versions of the Firefox browser, dropping the sign-out request from Helios did not prevent the user's connection from being terminated. From capturing network traffic over TLS during the vote casting procedure it can be observed that *Encrypted Alerts* are sent by both the server and the client. These encrypted alerts are indeed the `close_notify` alerts described in Section 3.2.2 and by recognizing these *fatal* alerts the client and the server terminate their TLS connection. After the connection is terminated, a new

TLS handshake (as described in Section 3.2) is initiated and the user's login is forgotten. The success of the truncation attacks against the Helios Voting system, Google Services and Microsoft Hotmail crucially rely on the lack of distinction between the termination modes of TLS. When a *fatal* closure is generated during an attack, it has to be ignored by the browser in order for these attacks to work. As described by Table 6.2 and the sections above, a large selection of modern browsers actually ignores this *fatal* closure. In this table, ✓indicates that a truncation attack was successful against the Helios Voting system on that particular version of the web browser, and ✗ indicates that the attack was unsuccessful.

Browser	TLS termination modes ignored
Firefox 46	✗
Firefox 38	✗
Internet Explorer 11	✓
Microsoft Edge	✓
Google Chrome 50	✓
Opera 37	✓

Table 6.2: Web browsers susceptible to truncation attacks. ✓indicates that the truncation attack was successful, and ✗ indicates that the attack failed.

During the course of this practical investigation, attempts have been made in order to uncover to which degree web browser developers are aware of these flaws. It would appear as though ignoring these termination modes is a trade-off made in order to improve compatibility. Prevention of truncation attacks against application flaws in sign-out procedures, such as those described in Chapter 5, does, because of these results, solely rely on the web designer implementing these applications. The next section will provide a suggestion as to how an application with authentication logic similar to Helios can be modified in order to thwart these types of attacks.

6.3 Secure Termination in a Web Application

Upon the realization that most web browsers still ignore the termination modes of TLS, web application designers and developers need to be aware of the complications that may come from authentication logic flaws. The documentation for TLSv1.2 (RFC5246 [16]) is not intended as a detailed step-by-step description of how to implement application security with the TLS protocol. Nor does the protocol guarantee security when an application is implemented poorly.

A web application does not, however, need to be vulnerable to truncation attacks. To provide an example of both an insecure and a secure sign-out procedure, a simple

web application is created and described in the next section. This application is written in *Django* which is a web framework for the *Python* programming language. This framework is both free and open source and it is designed to be a fast and reliable way of writing web applications. Django is chosen for the application described in the sections below because it is the same framework used by the developers of the Helios Voting system.

6.3.1 Insecure Web Application

Consider a simple Django application with a sign-out procedure based on the Helios Voting system. The main features of this application is that a user can log in using a personal password and send a form containing trivial information to the server. Upon posting this `send_done` form, the user receives a HTML page informing him that the form has been sent and that the application state has been changed (the user has been logged out). The procedure for sending a form in this application is illustrated by Listing 6.1 and Figure 6.1. From the trace in the listing below, it can be observed that the sign-out request is loaded from a `<iframe>` tag in the HTML page returned as a response to request number 2.

```
1. POST /send_done/
   Response: 302 - Found
   Location[/send_confirm/]
2. GET /send_confirm/
   Response: 200 OK - HTML payload
   ...
   <p><b>For your safety I've logged you out!</b></p>
   <iframe width="0" height="0" border="0" frameborder="0"
   src="/accounts/logout/"></iframe>
   ...
3. GET /accounts/logout/
   Response: 302 - Found
   Location[/]
```

Listing 6.1: Trace of send form procedure

An `<iframe>` tag in HTML indicates that another document is intended to be embedded in the HTML page when it is loaded. By embedding the source for the logout procedure, `/accounts/logout/`, a user is intended to be signed out when the `<iframe>` is processed in the HTML code. Processing the `<iframe>` in request number 2 in Listing 6.1 will create a third request that asks for the logout procedure to be initiated. Figure 6.1 illustrates a simplified representation of the trace in Listing 6.1 and from it, one can observe how the three requests occur sequentially. The sequence of HTTP requests and responses illustrated in Figure 6.1 uncovers an application

logic flaw in that the `GET /send_confirm/` request is responded by a `200 OK` and a HTML page indicating a successful logout *before* the `GET /accounts/logout/` sign-out request is made.

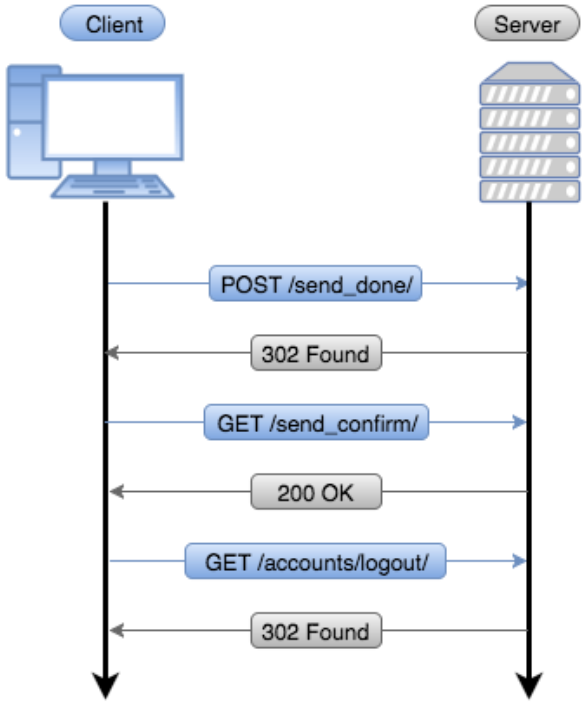

Figure 6.1: Send form procedure

As the authentication logic in this application is very much based on the Helios Voting system, it contains the same flaw and is susceptible to truncation attacks in the same way. By using the same techniques as those described in Section 5.3, an adversary can simply drop the sign-out request and thus prevent a successful logout.

6.3.2 Secure Web Application

Now imagine that a web developer is aware of the possibilities of truncation attacks compromising the application described above. The main flaw in the authentication logic from Listing 6.1 is that the user receives positive feedback of a successful logout *before* the sign-out request is made. Since this particular web application is written in Python with Django as a framework, HTTP requests can easily be created by using a sequence of `HttpRedirects`. By using such a redirect rather than the `iframe` tag from the example above to navigate the user through the sign-out procedure, a truncation attack can fairly easily be thwarted. In the trace in Listing 6.2 and

the sequence diagram in Figure 6.2 a suggestion to a new authentication logic is described.

```
1. POST /send_done/
   Response: 302 - Found
   Location[/send_confirm/]
2. GET /send_confirm/
   Response: 302 Found
   Location [/accounts/logout/]
3. GET /accounts/logout/
   Response: 302 - Found
   Location[/]
4. GET /
   Response 200 OK - HTML payload
   ...
   self.messages.success("You've been logged out. Come back
   soon!")
   ...
```

Listing 6.2: Suggestion for new trace of send form procedure

Observed by the trace in Listing 6.2, the feedback indicating a successful logout comes *after* the actual sign-out request has been made in form of a `self.message` provided by the Django framework. This is not the only way of providing positive feedback in Django but it will suffice for this particular demonstration. The main difference, however, between these two implementations is the way request number 2 is handled.

In this case, the `GET /send_confirm/` request does not initiate a sign-out request by embedding the logout source in an `<iframe>`. Instead, the Django application is configured to, after a given number of operations is done, issue a `HttpRedirect` which redirects the user to the `/accounts/logout/` location. This application design eliminates the possibility of the positive feedback indicating a successful being loaded before the sign-out request. By studying Figure 6.2 as a simplified illustration of the HTTP requests and responses in Listing 6.2 the suggestion to a secure form sending sequence can be more easily described.

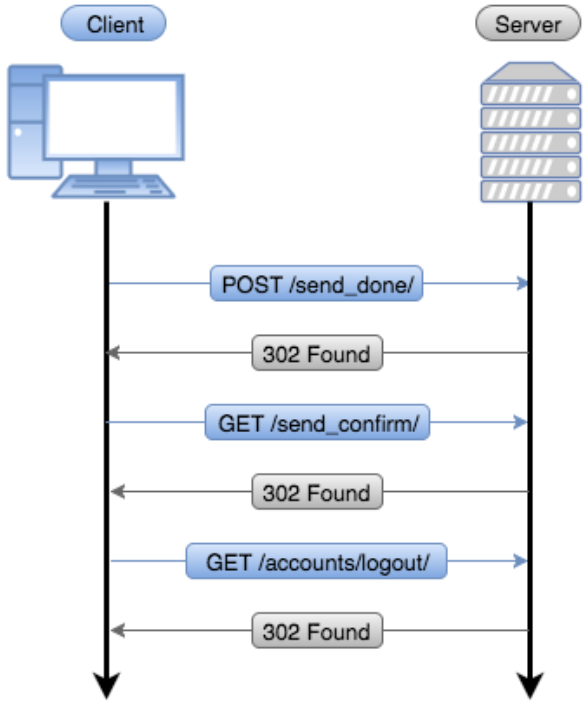

Figure 6.2: Suggestion to secure send form procedure

From comparing the two figures (6.1 and 6.2) it can be observed that the response of request number 2 is `200 OK` and `302 Found`, respectively. As described in Section 2.4.1 and Appendix A, a HTTP 302 status code indicates that a HTTP redirect will occur. In the case of this new send form procedure, an adversary dropping request number 2 will also prevent the intended redirect. By preventing the HTTP redirect the entire sign-out procedure in Listing 6.2 will be halted and feedback indicating a successful logout will never be received by the user. When this is the case, the user should eventually suspect that an error has occurred during the form posting procedure.

6.3.3 Secure Termination in a Web Application using TLS

Although the application created for the purposes of suggesting defenses against truncation attacks, it is not specified to be implemented with TLS security. The idea behind the truncation attacks described in Chapter 5 is that the only information an adversary needs is the size of the encrypted sign-out request. This type of information can be fairly easily found by using the basic network analysis techniques described in Section 2.5.

6.4 Summary

A truncation attack as it is described in this report is a fairly simple attack. The security implications of such an attack, however, can be serious depending on the purpose of the broken application. In the Django application described in this Chapter, trivial information is sent to the server by the `POST /send_done/` request, but this information could have just as well been of the more sensitive sort. It appears as though the understanding of the security guarantees provided by TLS is poor, and that web browser developers are either unaware of, or ignores, the security implications of such attacks. Regardless of this, the results from the practical investigation performed in this chapter prove that the danger of truncation attacks are still very much real, but can be fairly easily avoided.

Chapter 7
Discussion & Conclusion

7.1 Discussion

The simplicity of the truncation attacks described in this report is what makes them a serious threat. To attempt, for instance, an attack against the Helios Voting system, all an adversary needs is knowledge of how the vote casting procedure of Helios works and some basic knowledge of network analysis techniques. This is also the case for other web applications, such as for Microsoft services and Google services in 2013 [35].

When the paper discussing the truncation attacks against Helios, Microsoft, and Google was published, the vulnerabilities found were reported to their respective developers. From the results of the practical investigation made during the course of this report, we have learned that the developers of the Helios Voting system have not been able to fix the logic flaw in their application. Microsoft, however, has changed their sign-out procedure drastically since application vulnerabilities were reported to them. Whether or not these changes are due to the findings reported by Smyth and Pironti is unclear, but, as discussed in Section 5.4.2, Microsoft services is no longer vulnerable to truncation attacks against their sign-out procedure. The findings were also disclosed to Google when these attacks were discovered. While Google has made adjustments in their sign-out procedure to avoid these attacks, they reported back an assessment of the actual threat such attacks pose to their services when the results were initially reported to them. Google stated that they thought it was technically impossible to the properly defend users that use a shared computer [35].

Google has a point in that the premises of these attacks are highly specific. In the scenario described where these attacks are successful, an adversary *has* to have full control over the network and the user *has* to be using a shared computer. Furthermore, the adversary *has* to be able to access the shared computer after the user has left the computer. In addition to the claims that the scenario is too specific to be taken too seriously, Google pointed out that the attack could be thwarted by

the deletion of cookies and browser history and by closing the browser. This is also true, but it will place an unnecessary burden on the user.

7.2 Conclusion

When the Internet was created in the 1960s, security was not a concern. Over the years, however, as the Internet was made public, research was conducted in order to find a standardized protocol to provide secure communication over the open network. Today the most prominent protocol for network security is TLSv1.2. The TLS protocol is a cryptographic protocol used for web browsing, email, and many other services using the Internet. The TLS protocol has been around for decades and it is still a part of most people's everyday lives.

This thesis sought to explore the validity of truncation attacks against web applications using TLS by first recreating three attacks discovered by Smyth and Pironti [35] in 2013. These three attacks described the possibility of deceiving users into thinking a protocol session has been properly terminated by dropping sign-out requests. Due to flaws in application logic, Smyth and Pironti were able to cast votes on behalf of honest voters in the Helios voting system, obtain full control of Hotmail accounts and temporary control of Gmail accounts. In order to understand the recreation process of these attacks, background information on how and why these attacks work has been included in this report. Also included in this report is a detailed description of how the TLS protocol works.

In the process of recreating these attacks, the results indicated that a selection of popular web browsers handle the TLS termination modes differently. While this thesis initially sought to investigate to which extent it is possible to employ the Smyth and Pironti truncation attacks on security protocols other than TLS, the findings during the course of this research steered this report towards focusing on employing them in a variety of modern web browsers.

As a truncation attack was successfully attempted on the Helios Voting system in Chapter 5, this attack was replicated on a selection of modern browsers. By employing the different network analysis techniques mentioned on several occasions in this report, enough information was gathered and the attacks were made possible. As it turned out, the attack against the Helios Voting system was successful on the latest version of most of the web browsers tested in this report. The success of these attacks relies on poor handling of TLS termination modes in web browsers. As far as the author knows, this weakness has only been brought up with the publication of the paper introducing the truncation attack against TLS connections [35] and the Cookie Cutter attack [13]. While different browsers have been pointed out to have this flaw, the results of a practical investigation of modern desktop browsers'

handling of TLS termination modes have not been provided.

As the truncation attacks described in this report also rely on existing flaws in application logic, the responsibility on preventing them falls on web designers and developers. When realizing that most of these web browsers are incapable of preventing certain types of truncation attacks, the breadth of this report was widened to include a suggestion for a generic method to avoid truncation attacks from a web developer's point of view. This generic method is unique for this report in the sense that it shows how easily a very simple web application can be both insecure and secure against truncation attacks with just a couple of adjustments. All that is needed is a basic understanding of how TLS and truncation attacks on TLS connections work.

7.3 Future Work

TLS is a protocol that is under continuous development. With regards to future work, it would be interesting to investigate if the truncation attacks on TLSv1.2 described in this report would still be valid when the new version of TLS (TLSv1.3 [27]) is used for application security. TLSv1.3, however, is a pending draft, but a world wide deployment would probably occur by the end of 2016.

By recreating the attacks on the different browsers described in Section 6.2 when new versions of the most popular browsers are released, one can find out if poor handling of TLS termination modes persists. Furthermore, it would be interesting to investigate if the truncation attacks could be adapted and applied to other security protocols like DTLS and SSH.

References

[1] Home Page of IETF. https://www.ietf.org/. Accessed: 2016-05-28.

[2] iptables(8) - Linux User Manual. http://linux.die.net/man/8/iptables. Accessed: 2016-05-28.

[3] Operating System Platform Statistics. http://www.w3schools.com/browsers/browsers_os.asp. Accessed: 2016-04-12.

[4] Ubuntu 14.04 Release Notes. https://wiki.ubuntu.com/TrustyTahr/ReleaseNotes. Accessed: 2016-04-20.

[5] Web Browser Statistics. http://www.w3schools.com/browsers/browsers_stats.asp. Accessed: 2016-04-20.

[6] The Wireshark Wiki. wiki.wireshark.org. Accessed: 2016-05-28.

[7] Advanced Encryption Standard. In E. Biham (Ed.), *Fast Software Encryption, 4th International Workshop, FSE '97, Proceedings*, pp. 83–87 (1997).

[8] Adida, B. Helios: Web-based Open-Audit Voting. In *Proceedings of the 17th USENIX Security Symposium*, pp. 335–348 (2008).

[9] Amrutkar, C., P. Traynor, and P. C. van Oorschot. An Empirical Evaluation of Security Indicators in Mobile Web Browsers. *IEEE Transactions on Mobile Computing 14*(5), 889–903 (2015).

[10] Barth, A. HTTP State Management Mechanism. RFC 6265, Internet Engineering Task Force (IETF), https://tools.ietf.org/html/rfc6265 (2011).

[11] Bellare, M., R. Canetti, and H. Krawczyk. Keying Hash Functions for Message Authentication. In *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Proceedings*, pp. 1–15 (1996).

[12] Berbecaru, D. and A. Lioy. On the Robustness of Applications Based on the SSL and TLS Security Protocols. In *Public Key Infrastructure, 4th European PKI Workshop: Theory and Practice, EuroPKI 2007, Proceedings*, pp. 248–264 (2007).

[13] Bhargavan, K., A. Delignat-Lavaud, C. Fournet, A. Pironti, and P. Strub. Triple Handshakes and Cookie Cutters: Breaking and Fixing Authentication over TLS. In *2014 IEEE Symposium on Security and Privacy, SP 2014*, pp. 98–113 (2014).

[14] Cooper, D., S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. Internet X. 509 Public Key Infrastructure Certificate and CRL Profile. RFC 5280, Internet Engineering Task Force (IETF), https://tools.ietf.org/html/rfc5280 (2008).

[15] Dierks, T. and C. Allen. The TLS Protocol Version 1.0. RFC 2246, Internet Engineering Task Force (IETF), https://tools.ietf.org/html/rfc2246 (1999).

[16] Dierks, T. and E. Rescola. Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246, Internet Engineering Task Force (IETF), https://tools.ietf.org/html/rfc5246 (2008).

[17] Dierks, T. and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.1. RFC 4346, Internet Engineering Task Force (IETF), https://tools.ietf.org/html/rfc4346 (2006).

[18] Diffie, W. and M. E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory 22*(6), 644–654 (1976).

[19] Fielding, R., J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, Internet Engineering Task Force (IETF), https://tools.ietf.org/html/rfc2616 (1999).

[20] Gallagher, P. Digital Signature Standard (DSS). *Federal Information Processing Standards Publications, volume FIPS*, 186–3 (2013).

[21] Garfinkel, S. L. *PGP - Pretty Good Privacy: Encryption for Everyone (2. ed.)*. O'Reilly (1995).

[22] Kent, S. IP Encapsulating Security Payload (ESP). RFC 4303, Internet Engineering Task Force (IETF), https://tools.ietf.org/html/rfc4303 (2005).

[23] Kurose, J. F. and K. W. Ross. *Computer Networking: A Top-Down Approach Featuring the Internet* (6 ed.). Addison-Wesley (2013).

[24] Leiner, B. M., V. G. Cerf, D. D. Clark, R. E. Kahn, L. Kleinrock, D. C. Lynch, J. B. Postel, L. G. Roberts, and S. S. Wolff. A Brief History of the Internet. *Computer Communication Review 39*(5), 22–31 (2009).

[25] Mylonas, A., N. Tsalis, and D. Gritzalis. Evaluating the Manageability of Web Browsers Controls. In *Security and Trust Management - 9th International Workshop, STM 2013, Proceedings*, pp. 82–98 (2013).

[26] Recommendation, I. 200 (1994)| ISO/IEC 7498-1: 1994. *Information Technology – Open Systems Interconnection – Basic Reference Model: The Basic Model* (1994).

[27] Rescola, E. Transport Layer Security (TLS) Protocol Version 1.3. DRAFT. Technical report, Internet Engineering Task Force (IETF), https://tools.ietf.org/html/draft-ietf-tls-tls13-13. Accessed: 2016-05-28.

[28] Rescola, E. HTTP Over TLS. RFC 2818, Internet Engineering Task Force (IETF), https://tools.ietf.org/html/rfc2818 (2000).

[29] Rivest, R. L., A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-key Cryptosystems. *Commun. ACM 21*(2), 120–126 (1978, February).

[30] Schneier, B. *Applied Cryptography - Protocols, Algorithms, and Source Code in C (2. ed.)*. Wiley (1996).

[31] Sheffer, Y. and R. Holz. Summarizing Known Attacks on Transport Layer Security (TLS) and Datagram TLS (DTLS). RFC 7457, Internet Engineering Task Force (IETF), https://tools.ietf.org/html/rfc7457 (2015).

[32] Smyth, B. and A. Pironti. Attacking Helios: An authentication bug. https://www.youtube.com/watch?v=BsdjoVZ8xQA. Youtube video created 14. September 2012.

[33] Smyth, B. and A. Pironti. Truncating TLS connections to access GMail accounts. https://www.youtube.com/watch?v=Ux_gE_6RuvU. Youtube video created 6. December 2012.

[34] Smyth, B. and A. Pironti. Truncating TLS connections to steal Hotmail accounts. https://www.youtube.com/watch?v=nyx8FnhCq7g. Youtube video created 19. January 2013.

[35] Smyth, B. and A. Pironti. Truncating TLS Connections to Violate Beliefs in Web Applications. In *7th USENIX Workshop on Offensive Technologies, WOOT '13* (2013).

[36] Stallings, W. *Cryptography and Network Security* (5 ed.). Pearson Education India (2007).

[37] Ylonen, T. The Secure Shell Layer (SSH) Protocol Architecture. RFC 4251, Internet Engineering Task Force (IETF), https://tools.ietf.org/html/rfc4251 (2006).

[38] Zimmermann, H. OSI Reference Model – The ISO Model of Architecture for Open Systems Interconnection. *Communications, IEEE Transactions on Cryptography 28*(4), 425–432 (1980).

Appendix A

HTTP Status Codes

Informational

```
100  -- Continue
101  --  Switching Protocols
```

Success

```
200 -- OK
201 -- Created
202 -- Accepted
203 -- Non-Authoritative Information
204 -- No Content
205 -- Reset Content
206 -- Partial Content
```

Redirection

```
300 -- Multiple Choices
301 -- Moved Permanently
302 -- Found
303 -- See Other
304 -- Not Modified
305 -- Use Proxy
307 -- Temporary Redirect
```

Client Error

```
400 -- Bad Request
401 -- Unauthorized
402 -- Payment Required
403 -- Forbidden
404 -- Not Found
405 -- Method Not Allowed
406 -- Not Acceptable
407 -- Proxy Authentication Required
408 -- Request Time-out
409 -- Conflict
410 -- Gone
411 -- Length Required
412 -- Precondition Failed
413 -- Request Entity Too Large
414 -- Request-URI Too Large
415 -- Unsupported Media Type
416 -- Requested range not satisfiable
417 -- Expectation Failed
```

Server Error

```
500 -- Internal Server Error
501 -- Not Implemented
502 -- Bad Gateway
503 -- Service Unavailable
504 -- Gateway Time-out
505 -- HTTP Version not supported
```

Appendix B
TLS Protocol Data

B.1 Items Included in a TLS Session

session identifier:
Chosen by the server in order to identify an active session. Comes in form of an arbitrary byte sequence, and can also identify a resumable session state.

peer certificate:
May be null. If not, it will be a X509v3 certificate of the peer.

compression method:
Specifies which algorithm that will be used to compress data before encryption.

cipher spec:
Identifies the PRF that is being used to generate keys as well as the encryption and MAC algorithms.

master secret:
The server and the client share a 48-byte master secret.

is resumable:
If a session is allowed to initiate new connection, a flag will indicate this.

B.2 TLS Alert Messages

enum { warning(1), fatal(2), (255) } AlertLevel;
enum {

```
        close_notify(0),
        unexpected_message(10),
        bad_record_mac(20),
        decryption_failed_RESERVED(21),
        record_overflow(22),
        decompression_failure(30),
        handshake_failure(40),
        no_certificate_RESERVED(41),
        bad_certificate(42),
        unsupported_certificate(43),
        certificate_revoked(44),
        certificate_expired(45),
        certificate_unknown(46),
        illegal_parameter(47),
        unknown_ca(48),
        access_denied(49),
        decode_error(50),
        decrypt_error(51),
        export_restriction_RESERVED(60),
        protocol_version(70),
        insufficient_security(71),
        internal_error(80),
        user_canceled(90),
        no_renegotiation(100),
        unsupported_extension(110),
        (255)
    } AlertDescription;

    struct {
        AlertLevel level;
        AlertDescription description;
    } Alert;
```

B.3 TLS Handshake Messages

```
    enum {
        hello_request(0), client_hello(1), server_hello(2),
        certificate(11), server_key_exchange (12),
        certificate_request(13), server_hello_done(14),
        certificate_verify(15), client_key_exchange(16),
        finished(20), (255)
```

```
} HandshakeType;

struct {
    HandshakeType msg_type;    /* handshake type */
    uint24 length;             /* bytes in message */
    select (HandshakeType) {
        case hello_request:       HelloRequest;
        case client_hello:        ClientHello;
        case server_hello:        ServerHello;
        case certificate:         Certificate;
        case server_key_exchange: ServerKeyExchange;
        case certificate_request: CertificateRequest;
        case server_hello_done:   ServerHelloDone;
        case certificate_verify:  CertificateVerify;
        case client_key_exchange: ClientKeyExchange;
        case finished:            Finished;
    } body;
} Handshake;
```